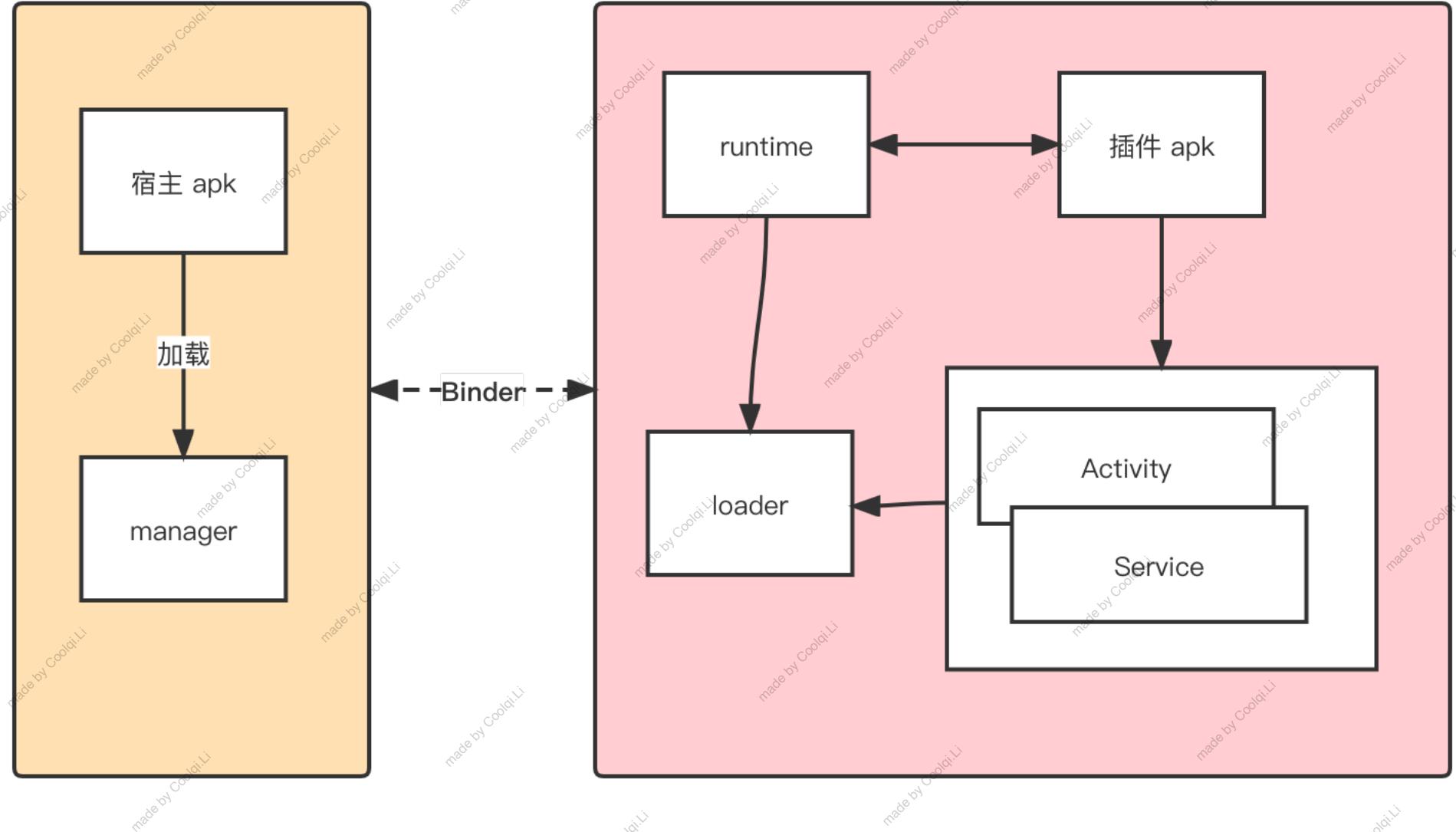


插件化 shadow 原理分享

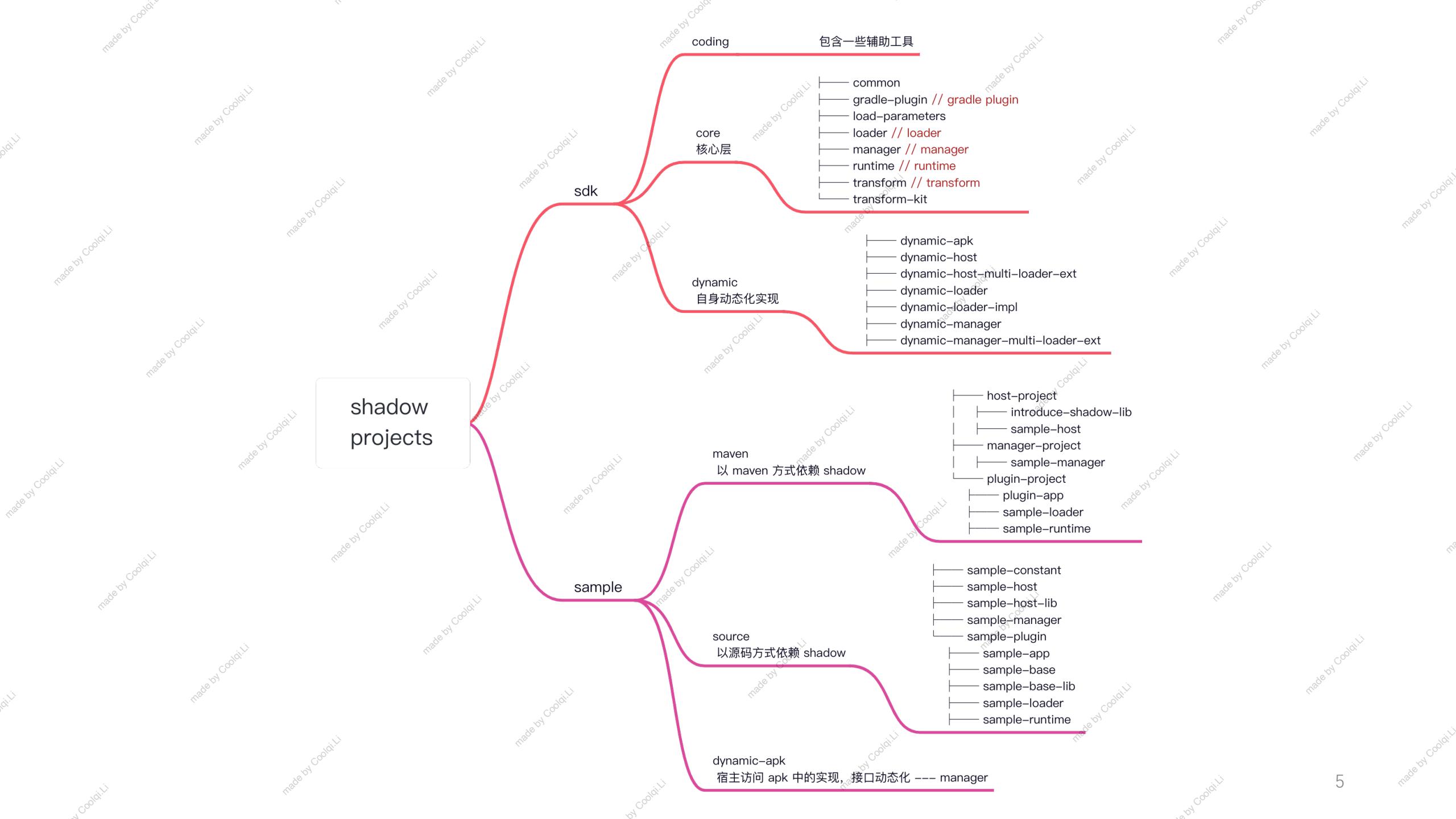
李帅哥

原理总览

in Shadow

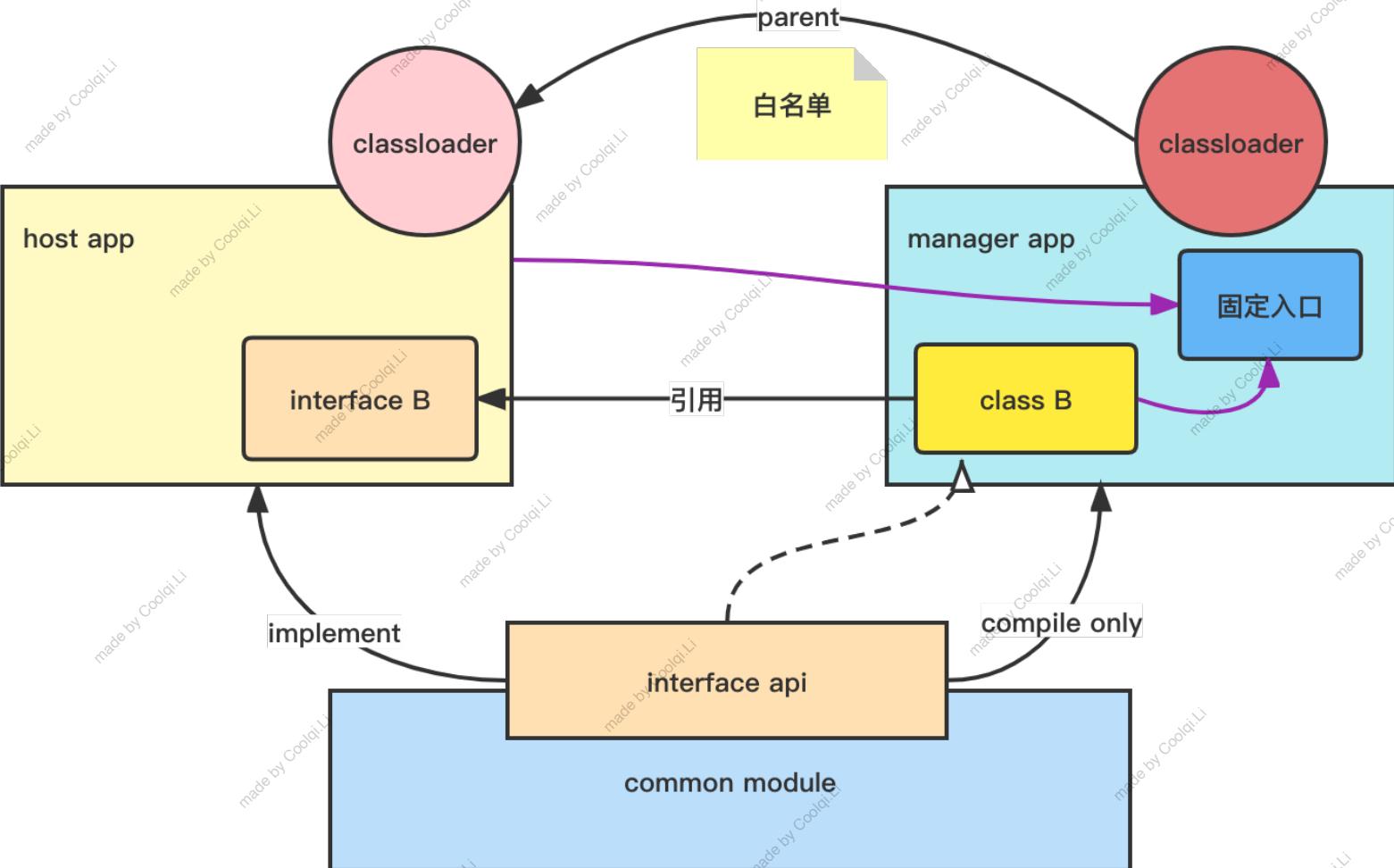


工程结构 in Shadow

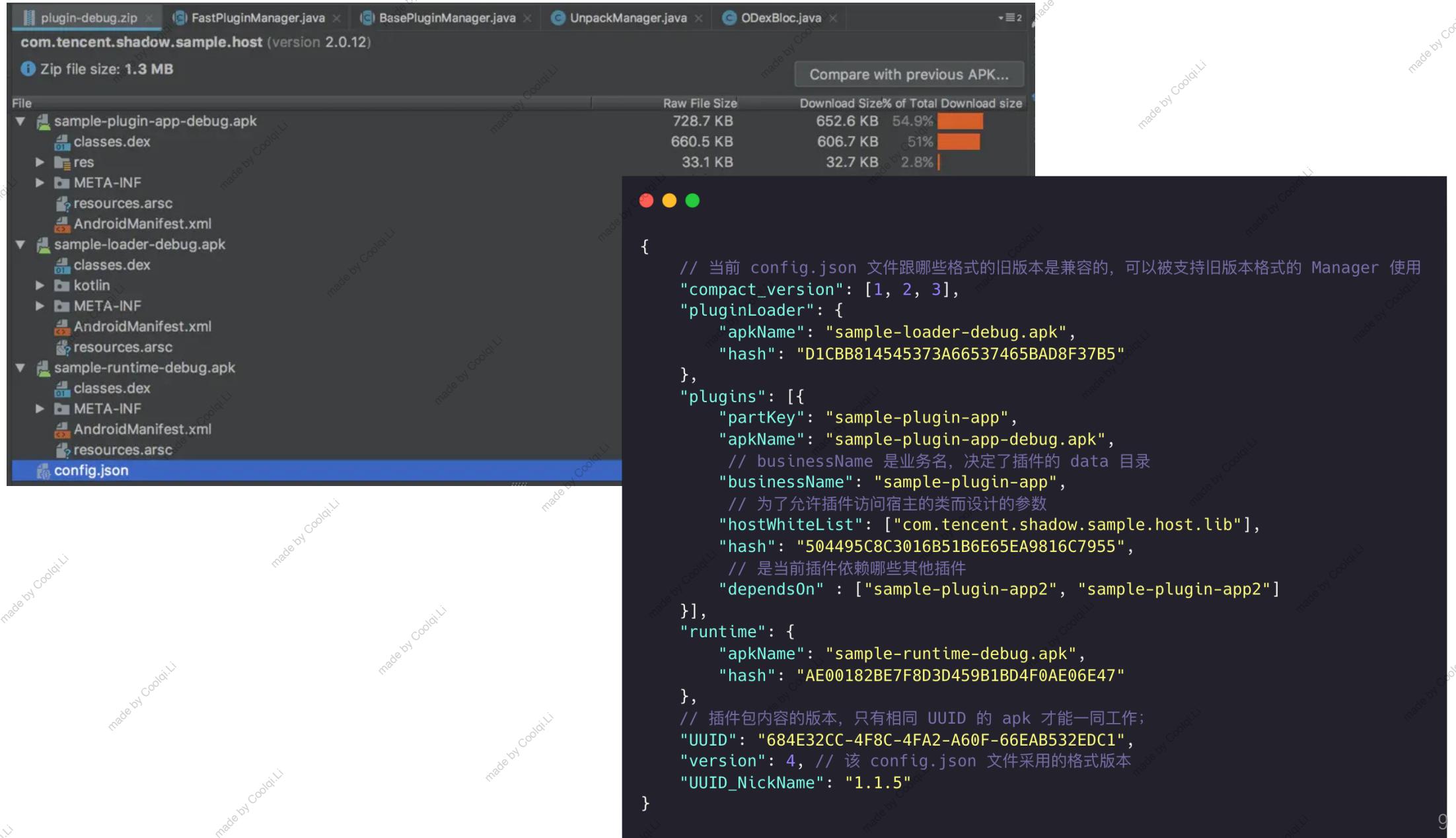


接口动态化原理

in shadow

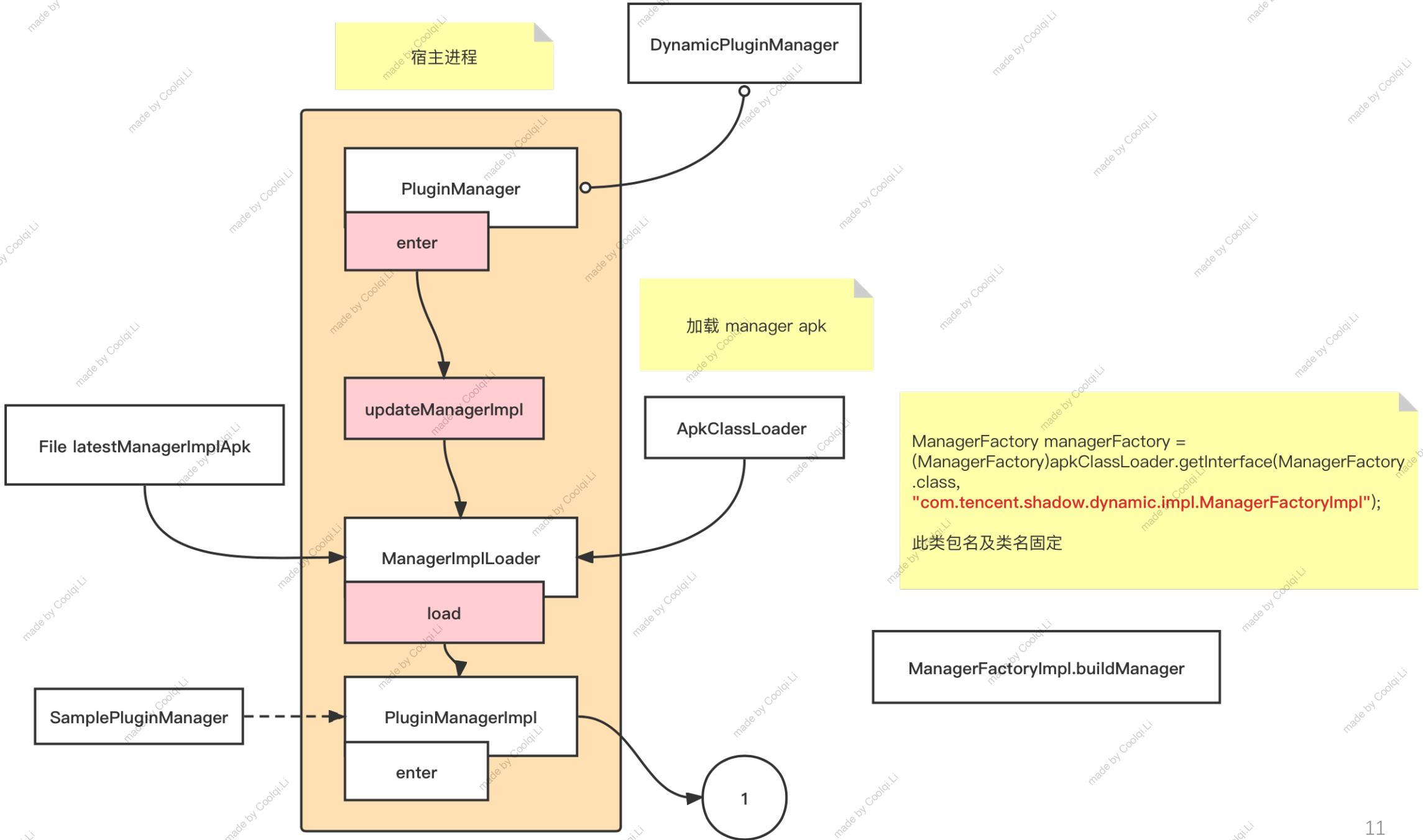


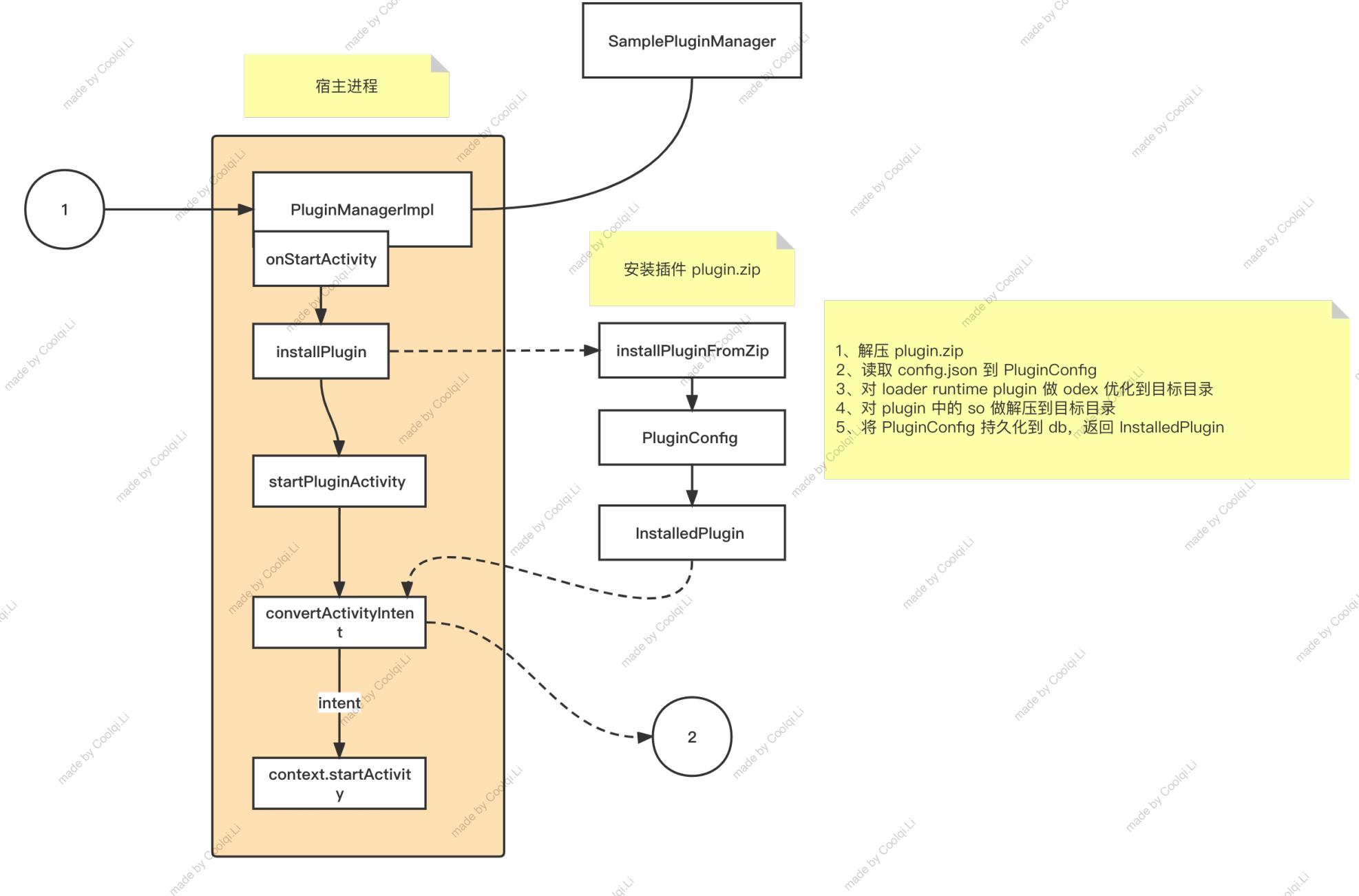
插件包的组成 in Shadow

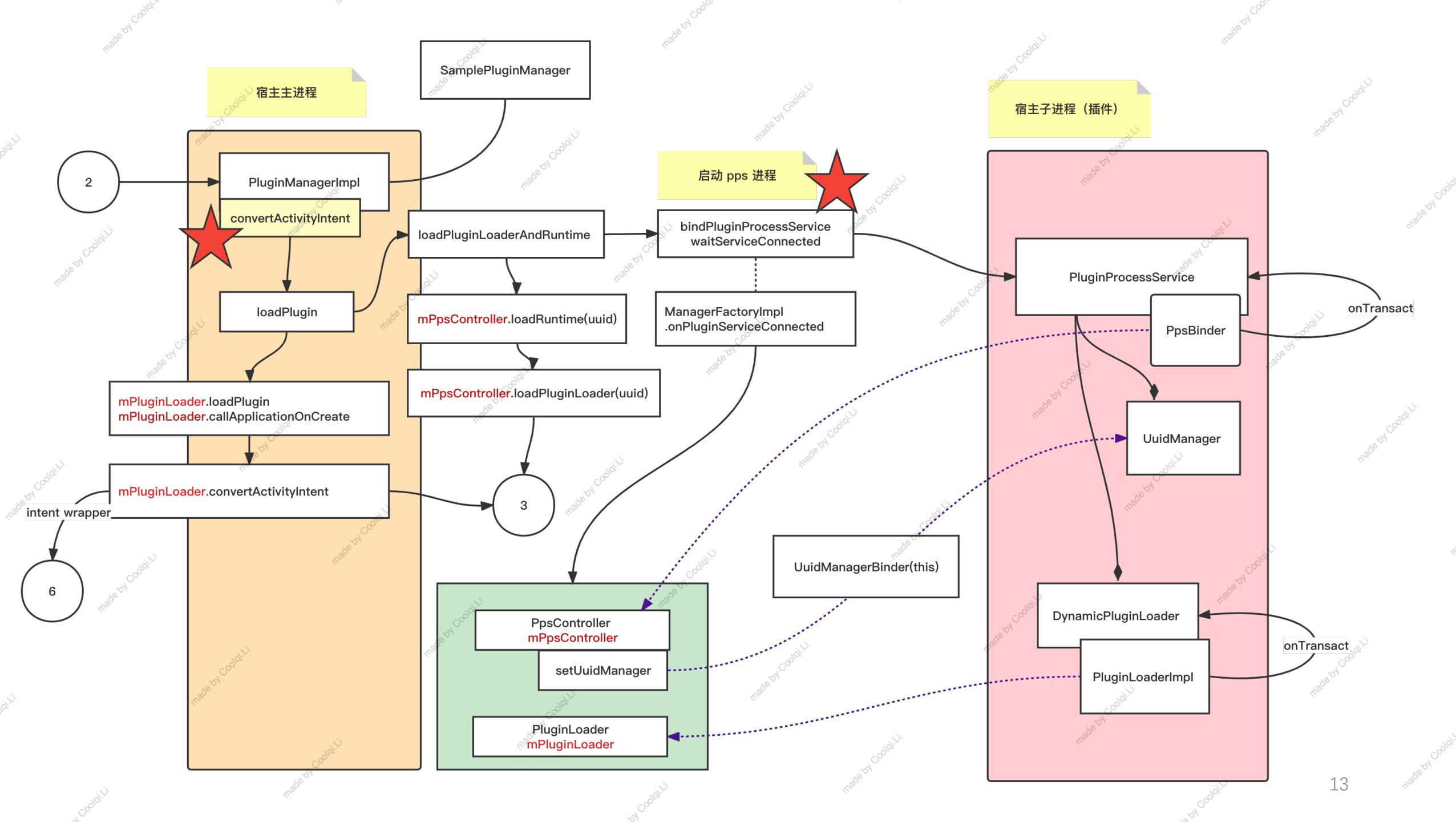


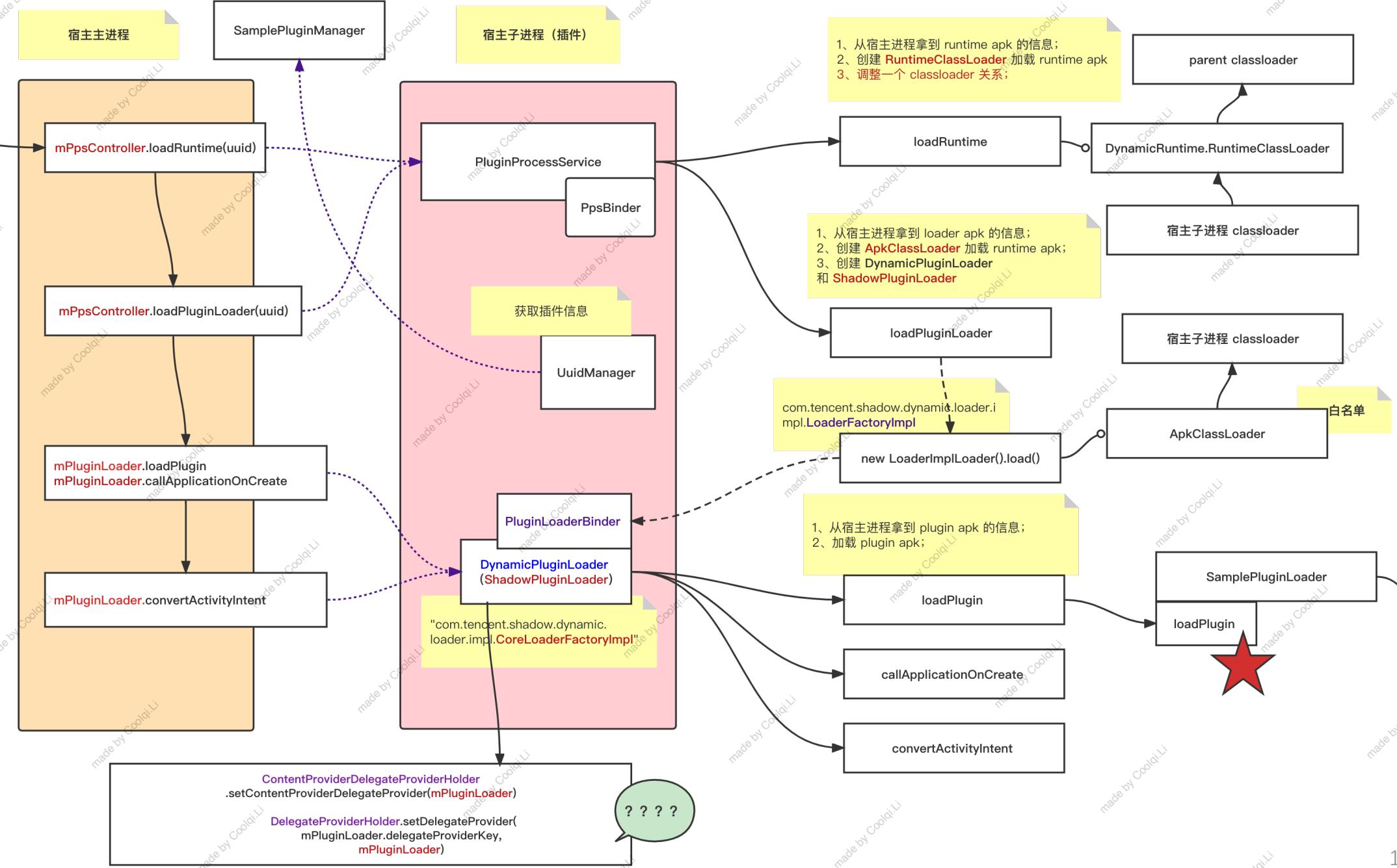
```
{  
    // 当前 config.json 文件跟哪些格式的旧版本是兼容的, 可以被支持旧版本格式的 Manager 使用  
    "compact_version": [1, 2, 3],  
    "pluginLoader": {  
        "apkName": "sample-loader-debug.apk",  
        "hash": "D1CBB814545373A66537465BAD8F37B5"  
    },  
    "plugins": [  
        "partKey": "sample-plugin-app",  
        "apkName": "sample-plugin-app-debug.apk",  
        // businessName 是业务名, 决定了插件的 data 目录  
        "businessName": "sample-plugin-app",  
        // 为了允许插件访问宿主的类而设计的参数  
        "hostWhiteList": ["com.tencent.shadow.sample.host.lib"],  
        "hash": "504495C8C3016B51B6E65EA9816C7955",  
        // 是当前插件依赖哪些其他插件  
        "dependsOn" : ["sample-plugin-app2", "sample-plugin-app2"]  
    ],  
    "runtime": {  
        "apkName": "sample-runtime-debug.apk",  
        "hash": "AE00182BE7F8D3D459B1BD4F0AE06E47"  
    },  
    // 插件包内容的版本, 只有相同 UUID 的 apk 才能一同工作;  
    "UUID": "684E32CC-4F8C-4FA2-A60F-66EAB532EDC1",  
    "version": 4, // 该 config.json 文件采用的格式版本  
    "UUID_NickName": "1.1.5"  
}
```

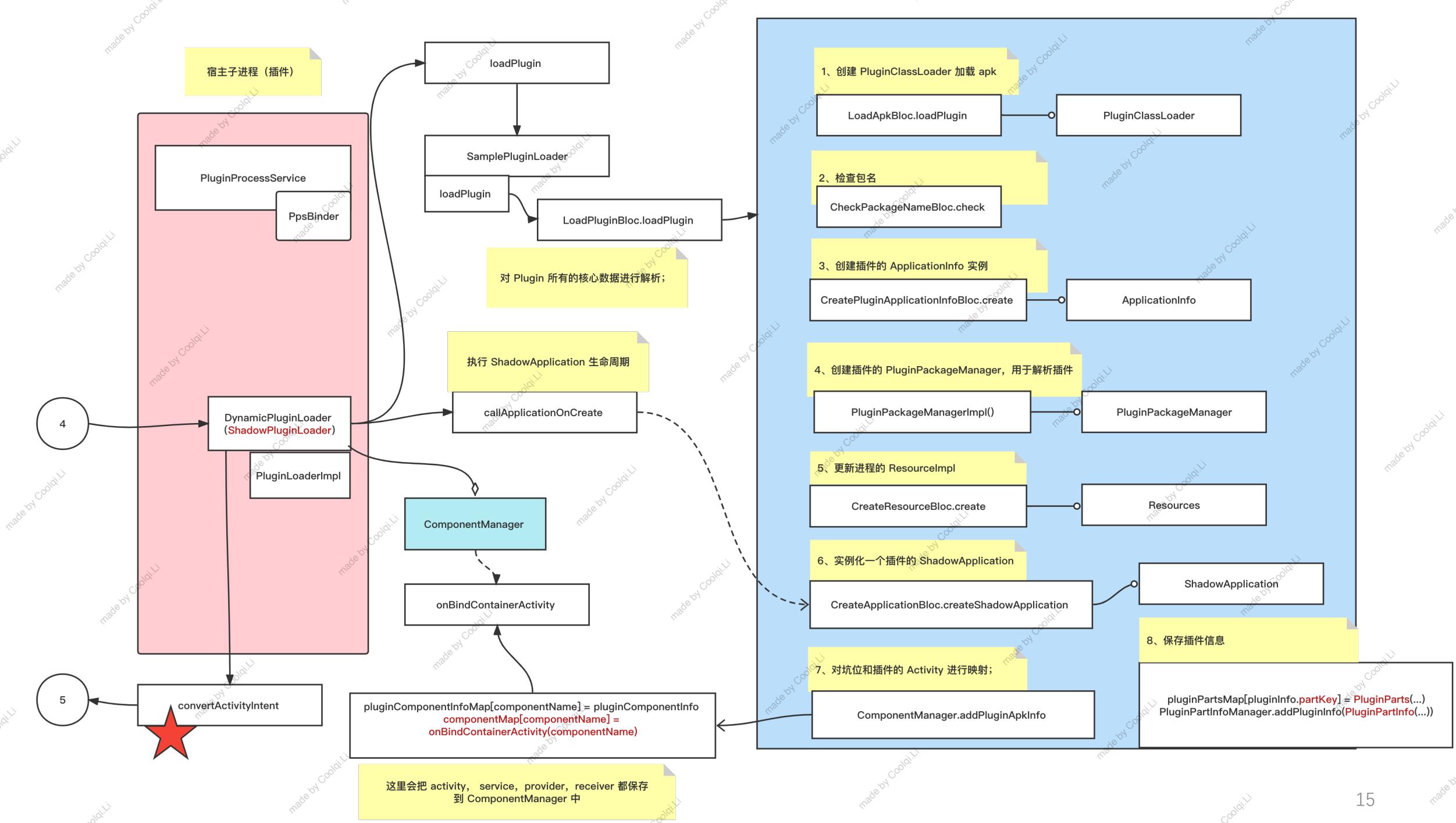
宿主如何启动插件 Activity in Shadow

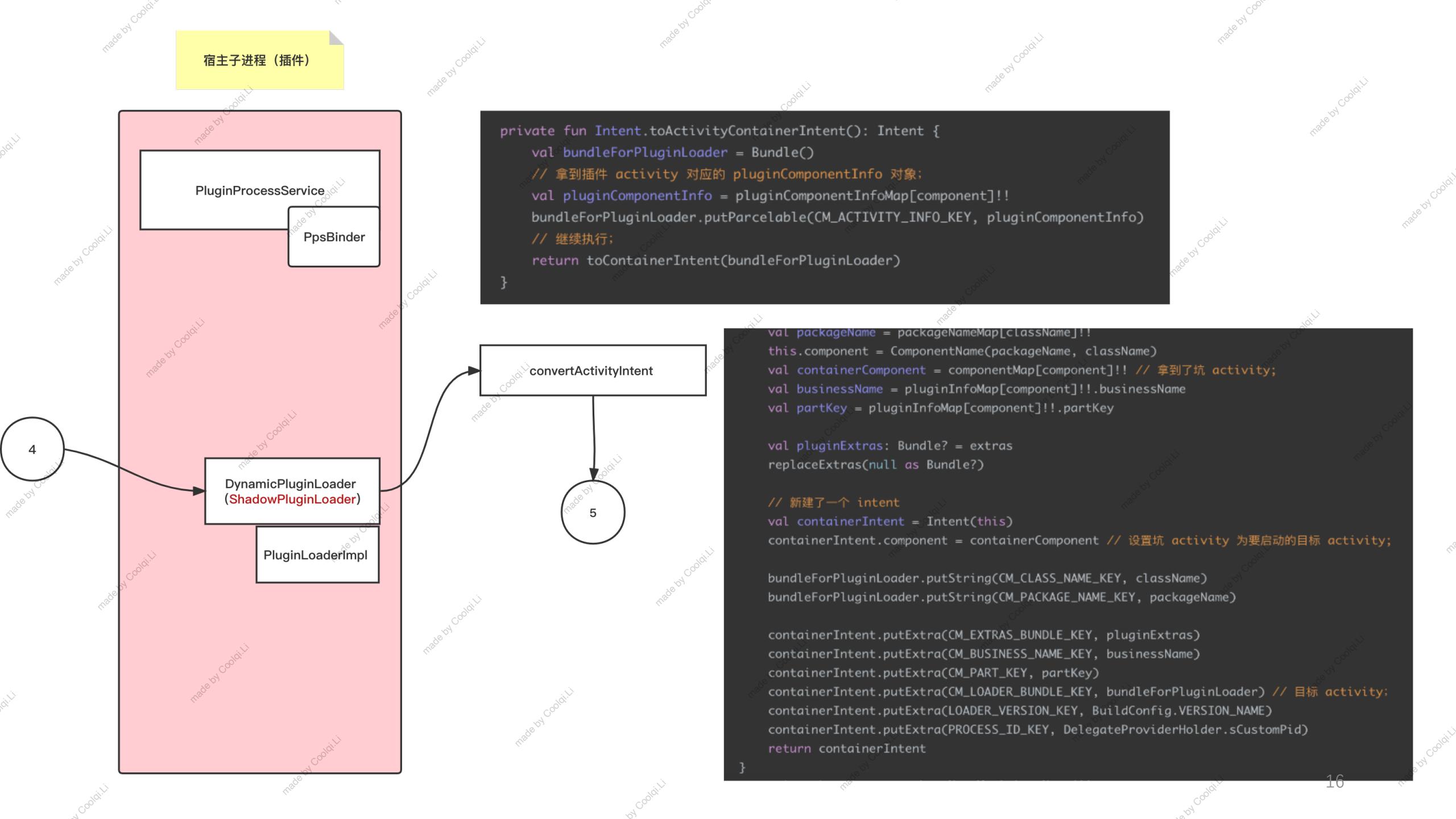


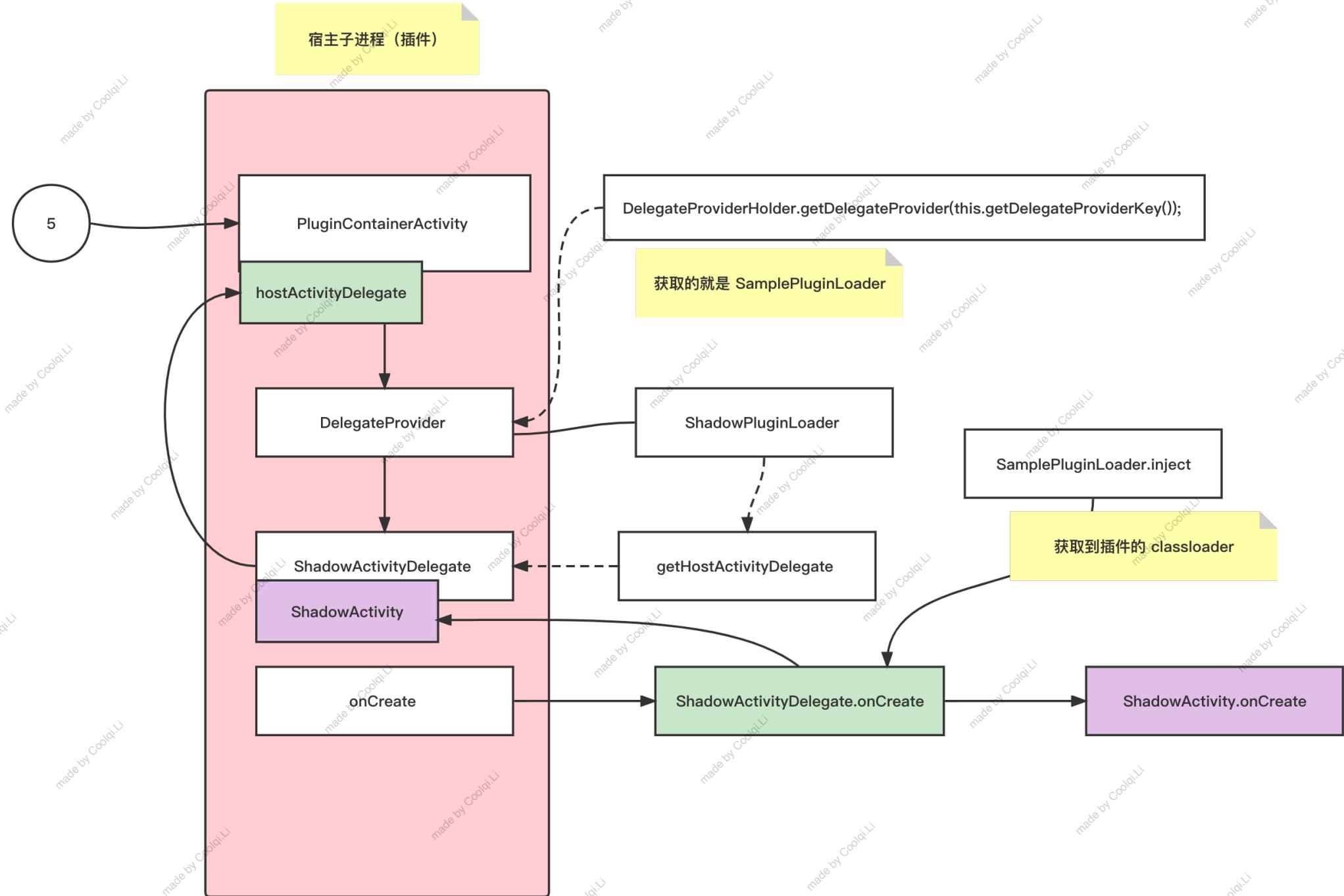












PPS 的作用 in Shadow

PPS 的作用

代表插件进程的生命周期

实现宿主和插件进程的双向通信

加载 Runtime 和 Loader

使插件中的 Service 能够跨进程工作

插件独立进程的好处 in Shadow

插件独立进程的好处

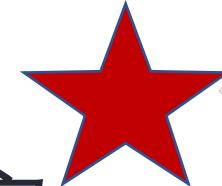
资源独立

出现 Crash 不会影响宿主进程

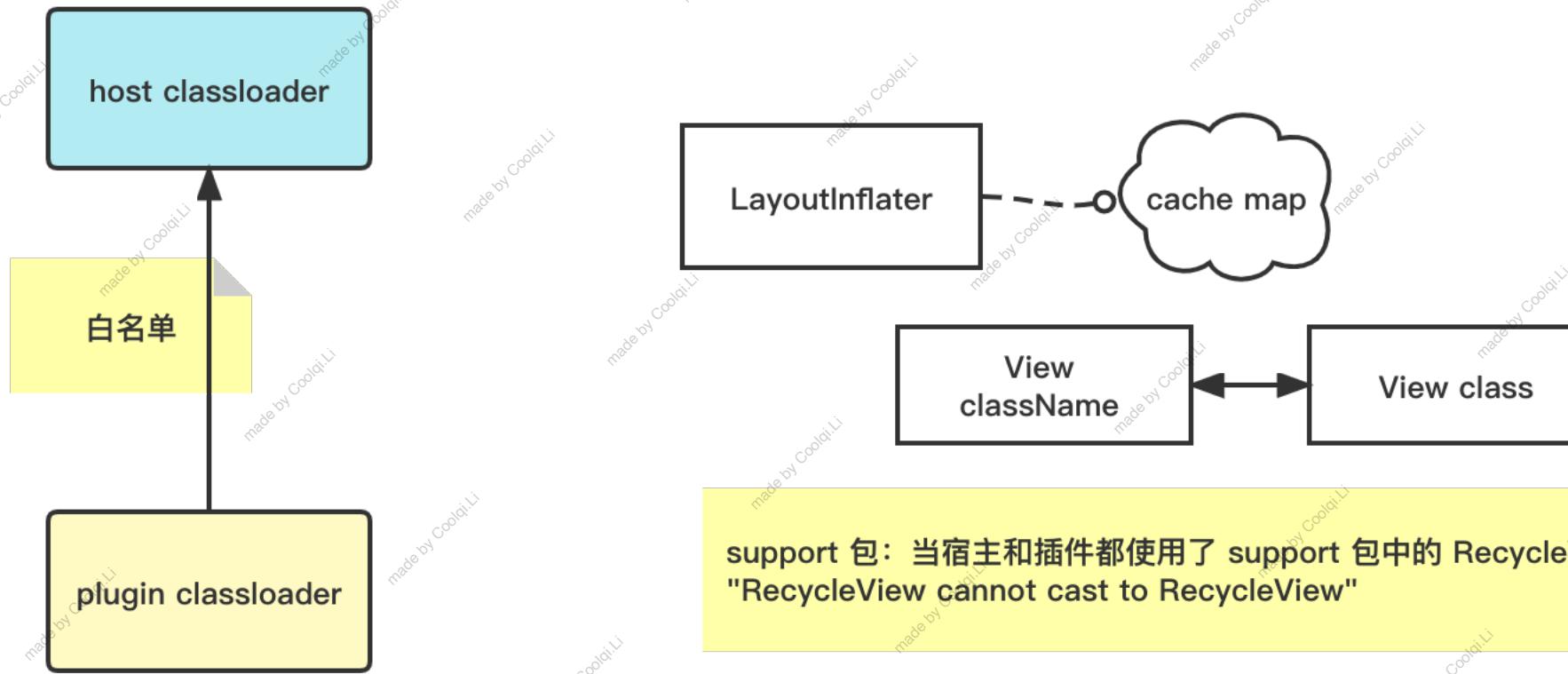
同一进程，插件和宿主存在 so 库冲突

相同 so 库的不同版本即不能同时加载，也不能换着加载

同名 View 问题和解决方案 in Shadow



插件和宿主中出现相同名字的类



传统解决方案

统
一
法

统
一
法

清空缓存法

插件直接使用宿主的 support 包

通过反射修改 LayoutInflator 的缓存，清空缓存达到目的

兼容性差

缓存被复写

Shadow 的解决方案

com.tencent.shadow.core.runtime.ShadowLayoutInflator
InnerInflater
com.tencent.shadow.core.runtime.ShadowFactory2

核心点

注入 Factory

LayoutInflater 在构造 View 时就会先试图让注入的 Factory 构造

如果 Factory 能够构造出来，就会优先使用

自定义 Factory

复制了原本内置构造逻辑的代码

缓存的 Key 添加了 partKey

一个 LayoutInflater 对象只能 set 一次 Factory
允许 clone 一个 LayoutInflater 对象，保留它的的 Factory

```
private View createCustomView(String name, Context context, AttributeSet attrs) {
    String cacheKey = mPartKey + name;
    Constructor<? extends View> constructor = sConstructorMap.get(cacheKey);
    if (constructor != null && !verifyClassLoader(context, constructor)) {
        constructor = null;
        sConstructorMap.remove(cacheKey);
    }
    Class<? extends View> clazz = null;

    try {
        if (constructor == null) {
            // Class not found in the cache, see if it's real, and try to add it
            clazz = context.getClassLoader().loadClass(name).asSubclass(View.class);
        }
    } catch (ClassNotFoundException e) {
        Log.w("View", "Failed to load class " + name + " from context " + context);
    }
}
```

ShadowFactory2

```
public static ShadowLayoutInflater build(LayoutInflater original, Context newContext, String partKey) {
    InnerInflater innerInflater = new InnerInflater(original, newContext, partKey);
    return new ShadowLayoutInflater(innerInflater, newContext, partKey);
}

private static class InnerInflater extends ShadowLayoutInflater {
    private InnerInflater(LayoutInflater original, Context newContext, String partKey) {
        super(original, newContext, partKey);
        setFactory2(new ShadowFactory2(partKey, layoutInflater: this));
    }
}

private ShadowLayoutInflater(LayoutInflater original, Context newContext, String partKey) {
    super(original, newContext);
}
```

ShadowLayoutInflater

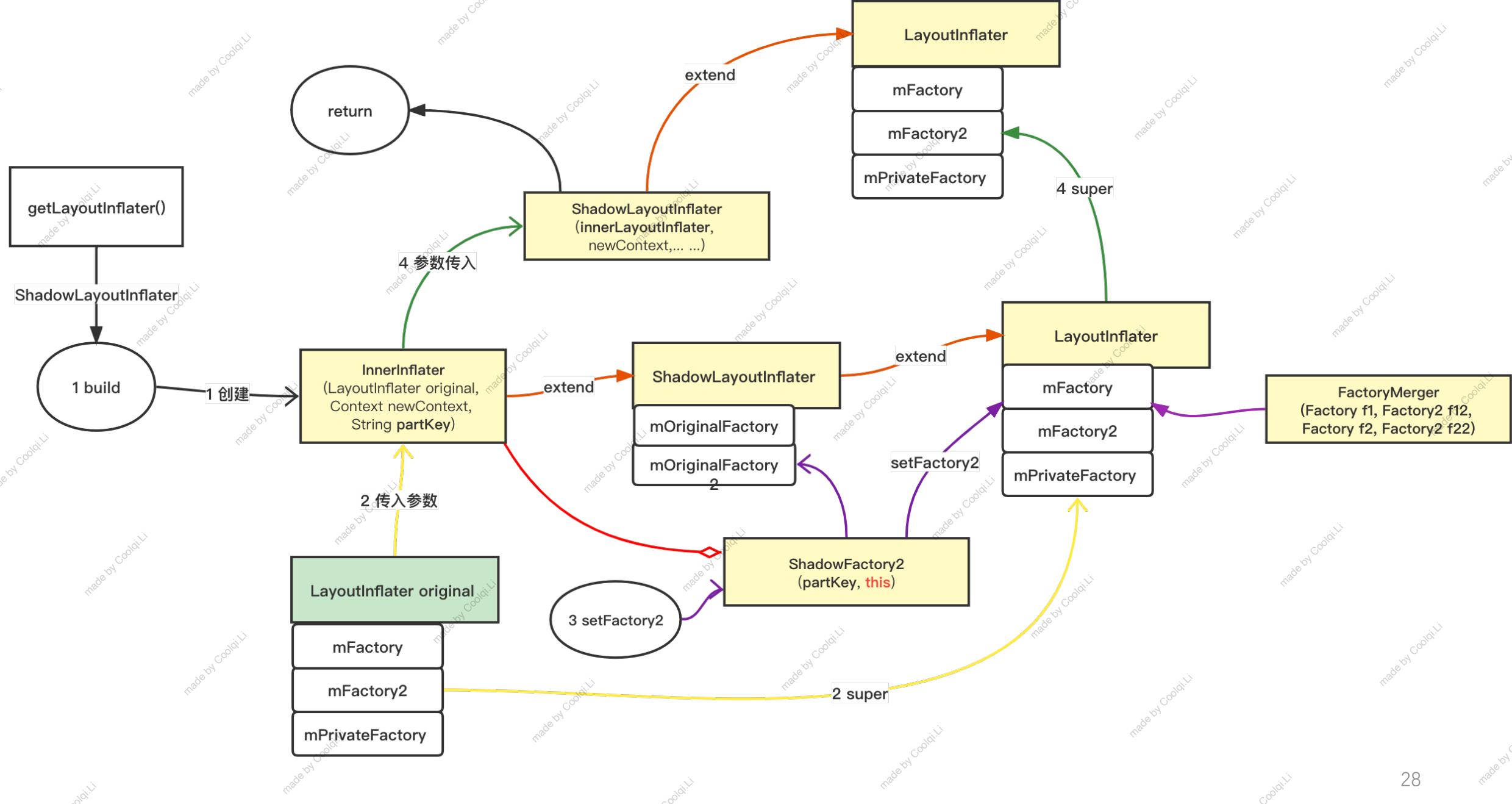
LayoutInflater

```
public final View tryCreateView(@Nullable View parent, @NonNull String name,
    @NonNull Context context,
    @NonNull AttributeSet attrs) {
    if (name.equals(TAG_1995)) {
        // Let's party like it's 1995!
        return new BlinkLayout(context, attrs);
    }

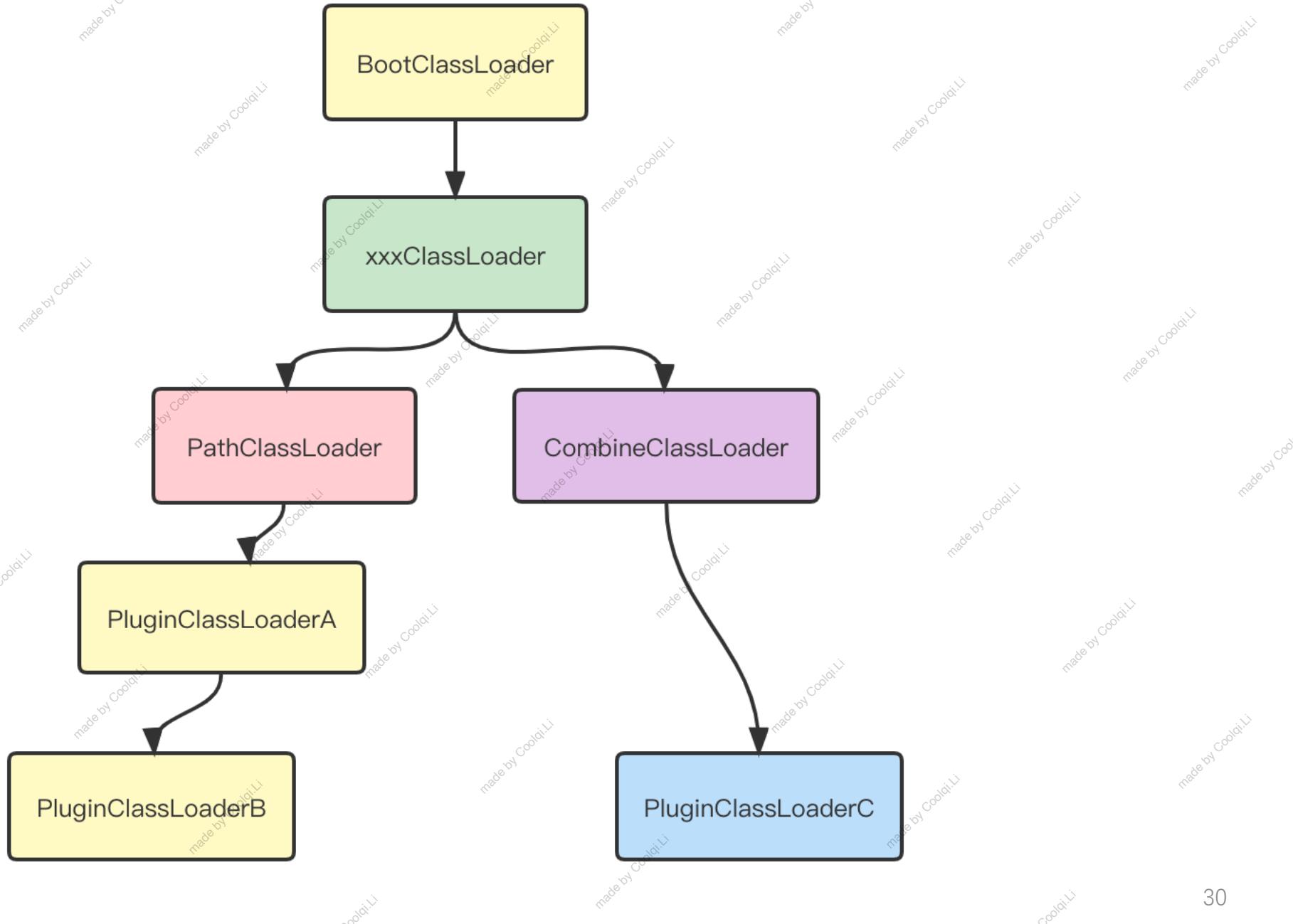
    View view;
    if (mFactory2 != null) {
        view = mFactory2.onCreateView(parent, name, context, attrs);
    } else if (mFactory != null) {
        view = mFactory.onCreateView(name, context, attrs);
    } else {
        view = null;
    }

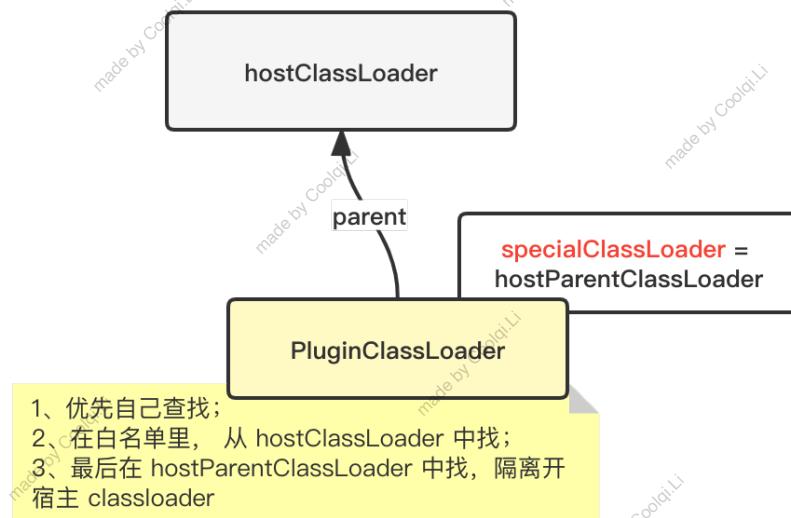
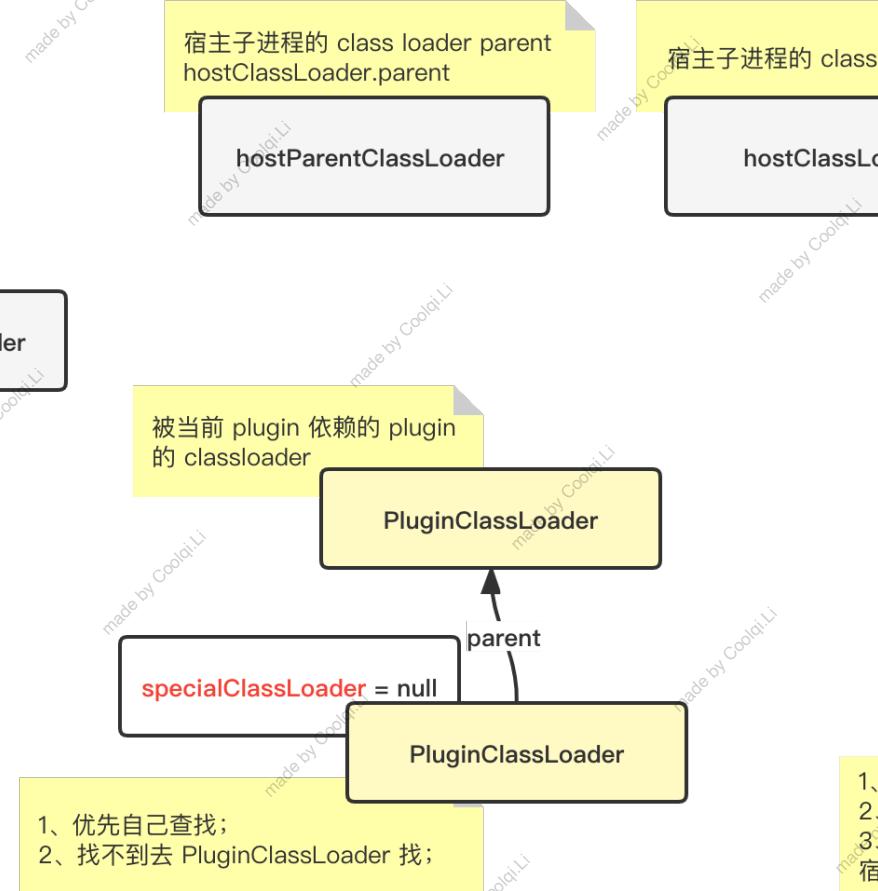
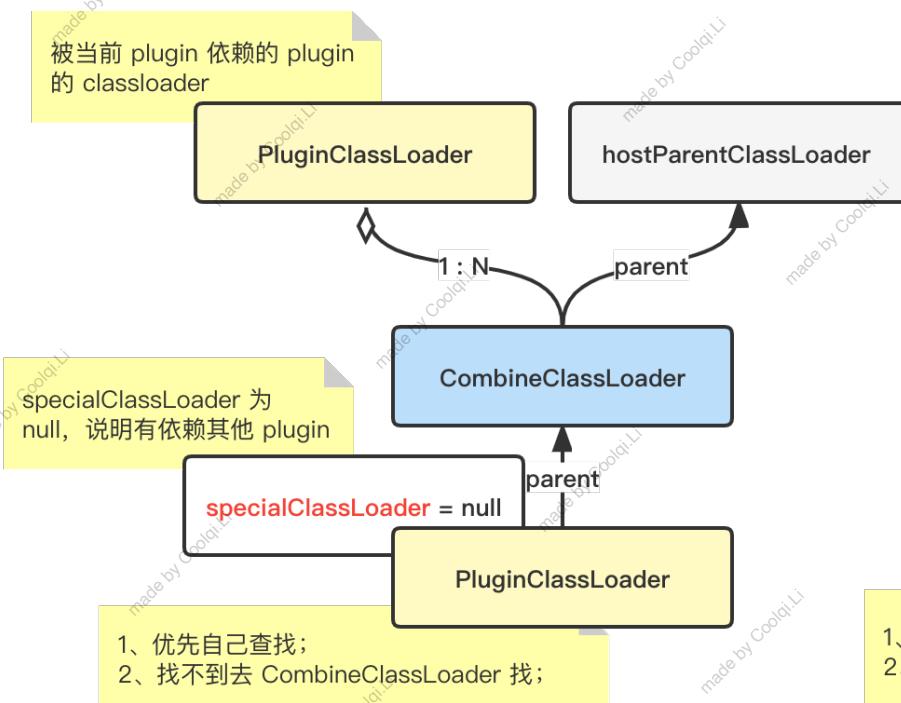
    if (view == null && mPrivateFactory != null) {
        view = mPrivateFactory.onCreateView(parent, name, context, attrs);
    }

    return view;
}
```



插件 PluginClassLoader 浅析 in Shadow





插件包名和宿主包名 in Shadow

```
object CheckPackageNameBloc {
    @Throws(ParsePluginApkException::class)
    fun check(
        pluginManifest: PluginManifest,
        hostApplicationContext: Context
    ) {
        if (pluginManifest.applicationPackageName != hostApplicationContext.packageName) {
            /*
            要求插件和宿主包名一致有两方面原因：
            1. 正常的构建过程中，aapt会将包名写入到arsc文件中。插件正常安装运行时，如果以
            android.content.Context.getPackageName为参数传给
            android.content.res.Resources.getIdentifier方法，可以正常获取到资源。但是在插件环境运行时，
            Context.getPackageName会得到宿主的packageName，则getIdentifier方法不能正常获取到资源。为此，
            一个可选的办法是继承Resources，覆盖getIdentifier方法。但是Resources的构造器已经被标记为
            @Deprecated了，未来可能会不可用，因此不首选这个方法。
            2. Android系统，更多情况下是OEM修改的Android系统，会在我们的context上调用getPackageName或者
            getOpPackageName等方法，然后将这个packageName跨进程传递做它用。系统的其他代码会以这个packageName
            去PackageManager中查询权限等信息。如果插件使用自己的包名，就需要在Context的getPackageName等实现中
            new Throwable(), 然后判断调用来源以决定返回自己的包名还是插件的包名。但是如果保持采用宿主的包名，则没有
            这个烦恼。
            我们也可以始终认为Shadow App是宿主的扩展代码，使用是宿主的一部分，那么采用宿主的包名就是理所应当的了。
            */
        throw ParsePluginApkException("插件和宿主包名不一致。宿主:${hostApplicationContext.packageName} 插件:${pluginManifest.applicationPackageName}")
    }
}
```

插件包的资源 in Shadow

```
fun create(archiveFilePath: String, hostAppContext: Context): Resources {
    triggerWebViewHookResources(hostAppContext)

    val packageManager = hostAppContext.packageManager
    val applicationInfo = ApplicationInfo()
    val hostApplicationInfo = hostAppContext.applicationInfo
    applicationInfo.packageName = hostApplicationInfo.packageName
    applicationInfo.uid = hostApplicationInfo.uid

    if (Build.VERSION.SDK_INT > MAX_API_FOR_MIX_RESOURCES) {
        fillApplicationInfoForNewerApi(applicationInfo, hostApplicationInfo, archiveFi
    } else {
        fillApplicationInfoForLowerApi(applicationInfo, hostApplicationInfo, archiveFi
    }

    try {
        val pluginResource = packageManager.getResourcesForApplication(applicationInfo
            // 利用资源分区，将宿主和插件 apk 添加到同一个 Resources 对象中;
        return if (Build.VERSION.SDK_INT > MAX_API_FOR_MIX_RESOURCES) {
            pluginResource
        } else {
            // 第一个方案是 MixResources 方案;
            val hostResources = hostAppContext.resources
            MixResources(pluginResource, hostResources)
        }
    } catch (e: PackageManager.NameNotFoundException) {
        throw RuntimeException("Plugin $archiveFilePath")
    }
}
```

ShadowContext 代理 in Shadow

```
public class ShadowContext extends SubDirContextThemeWrapper {
    PluginComponentLauncher mPluginComponentLauncher;
    ClassLoader mPluginClassLoader;
    ShadowApplication mShadowApplication;
    Resources mPluginResources;
    Resources mMixResources;
    LayoutInflater mLayoutInflater;
    ApplicationInfo mApplicationInfo;
    protected String mPartKey;
    private String mBusinessName;
    final private Map<BroadcastReceiver, BroadcastReceiverWapper> mBroadcastReceivers = new HashMap<>();

    public ShadowContext() {
    }

    public ShadowContext(Context base, int themeResId) {
        super(base, themeResId);
    }
}
```

ShadowActivity、ShadowApplication 的父类都是 ShadowContext

宿主如何启动插件 Service in Shadow

```
PluginManager pluginManager =  
    InitApplication.getPluginManager();  
pluginManager.enter(MainActivity.this,  
FROM_ID_CALL_SERVICE, null, null);
```

宿主进程

PluginManagerImpl

callPluginService

intent with data

installPlugin

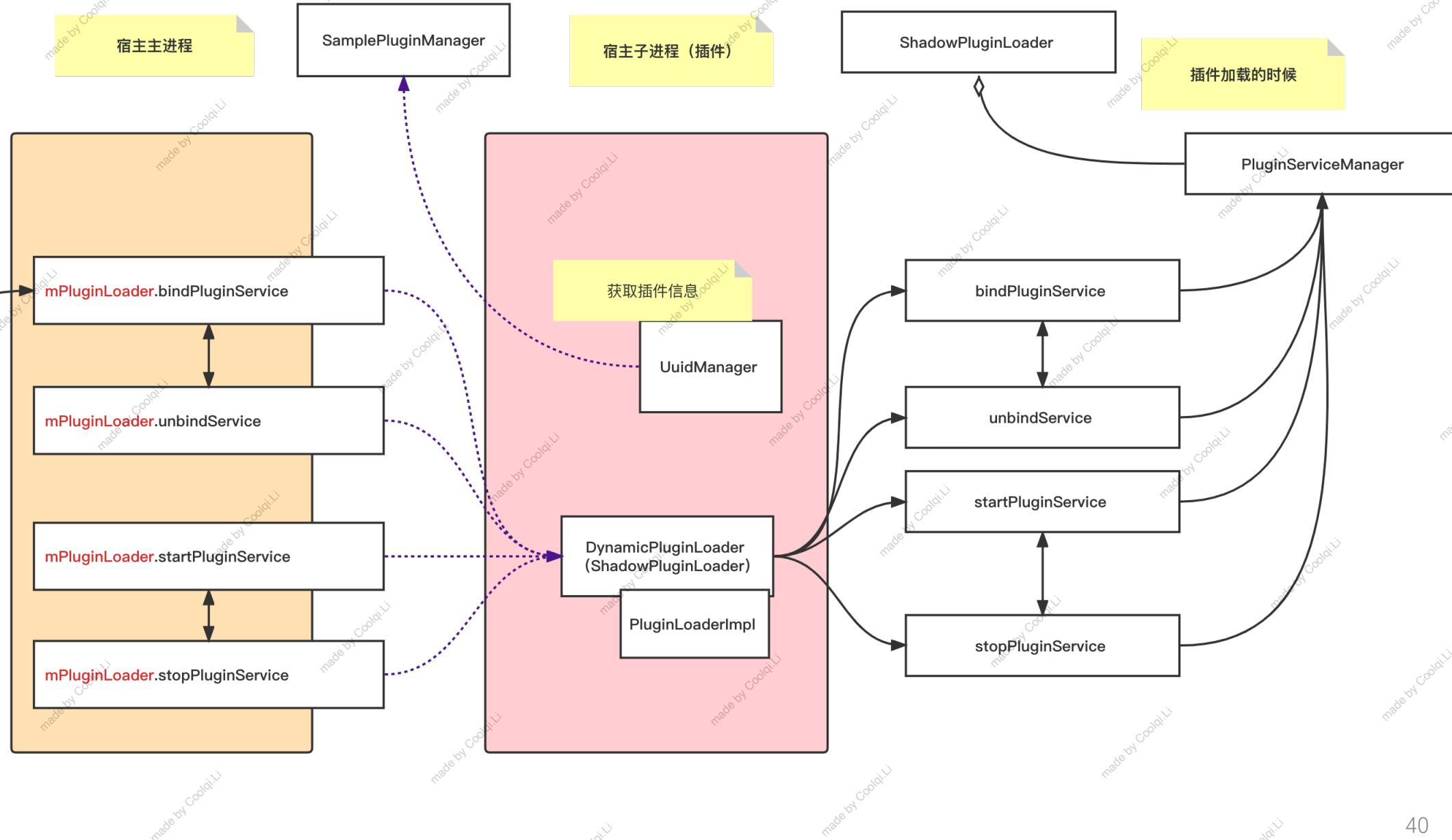
loadPlugin

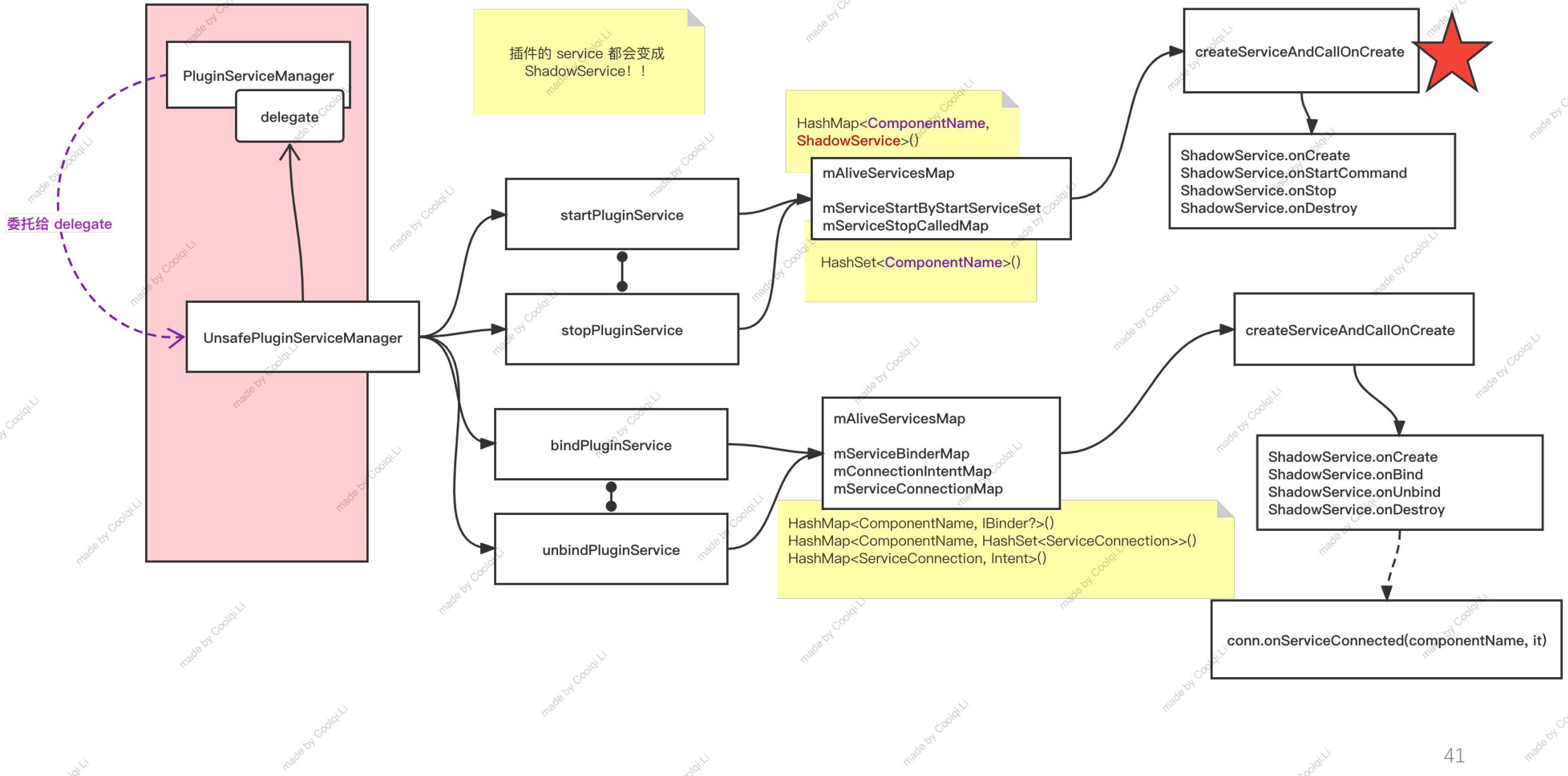
intent

mPluginLoader.bindPluginService

SamplePluginManager

2





```
private fun createServiceAndCallOnCreate(intent: Intent): ShadowService {
    val service = newServiceInstance(intent)
    service.onCreate()
    return service
}

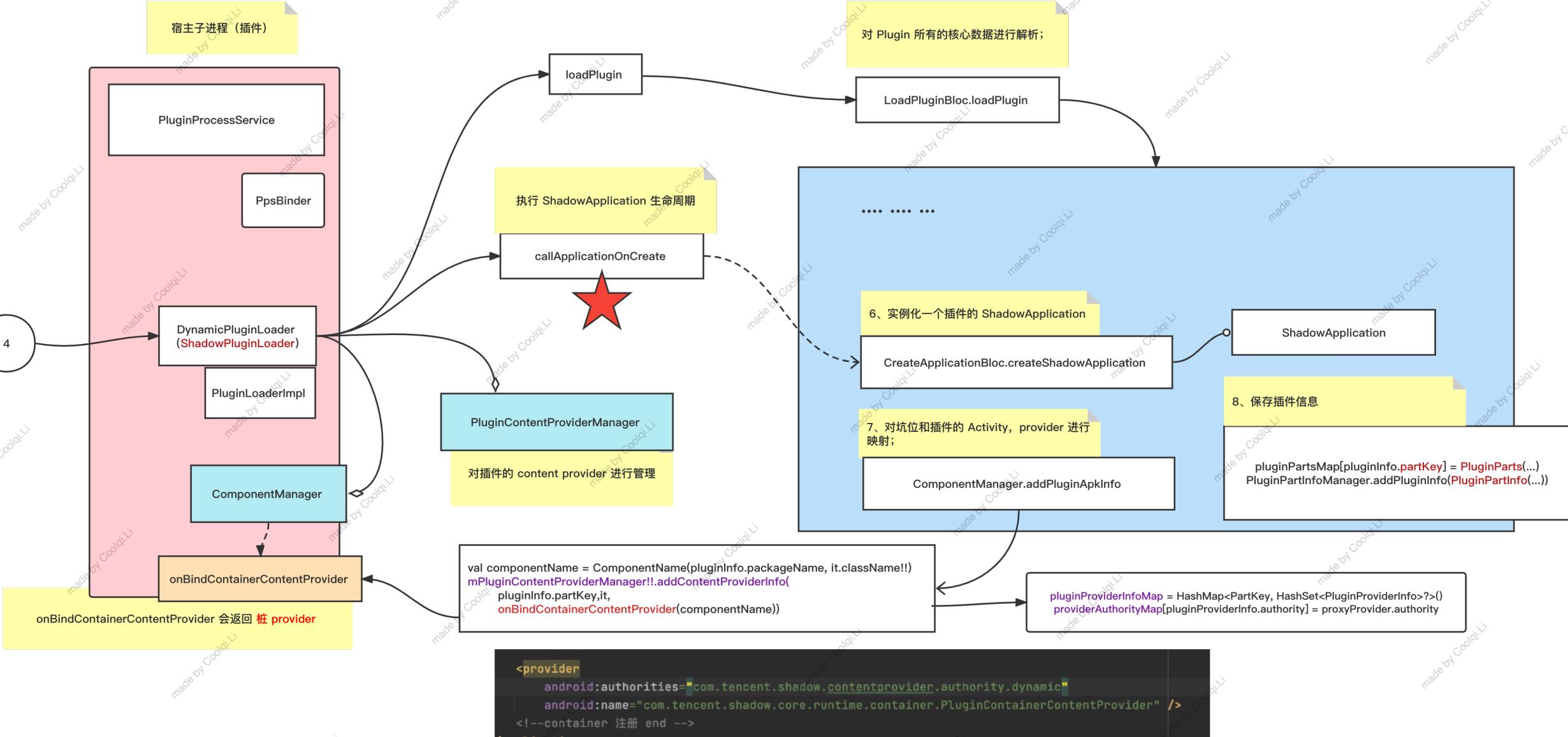
private fun newServiceInstance(intent: Intent): ShadowService {
    val componentName = intent.component!!
    val businessName = mPluginLoader.mComponentManager.getComponentBusinessName(componentName)
    val partKey = mPluginLoader.mComponentManager.getComponentPartKey(componentName)
    val className = componentName.className

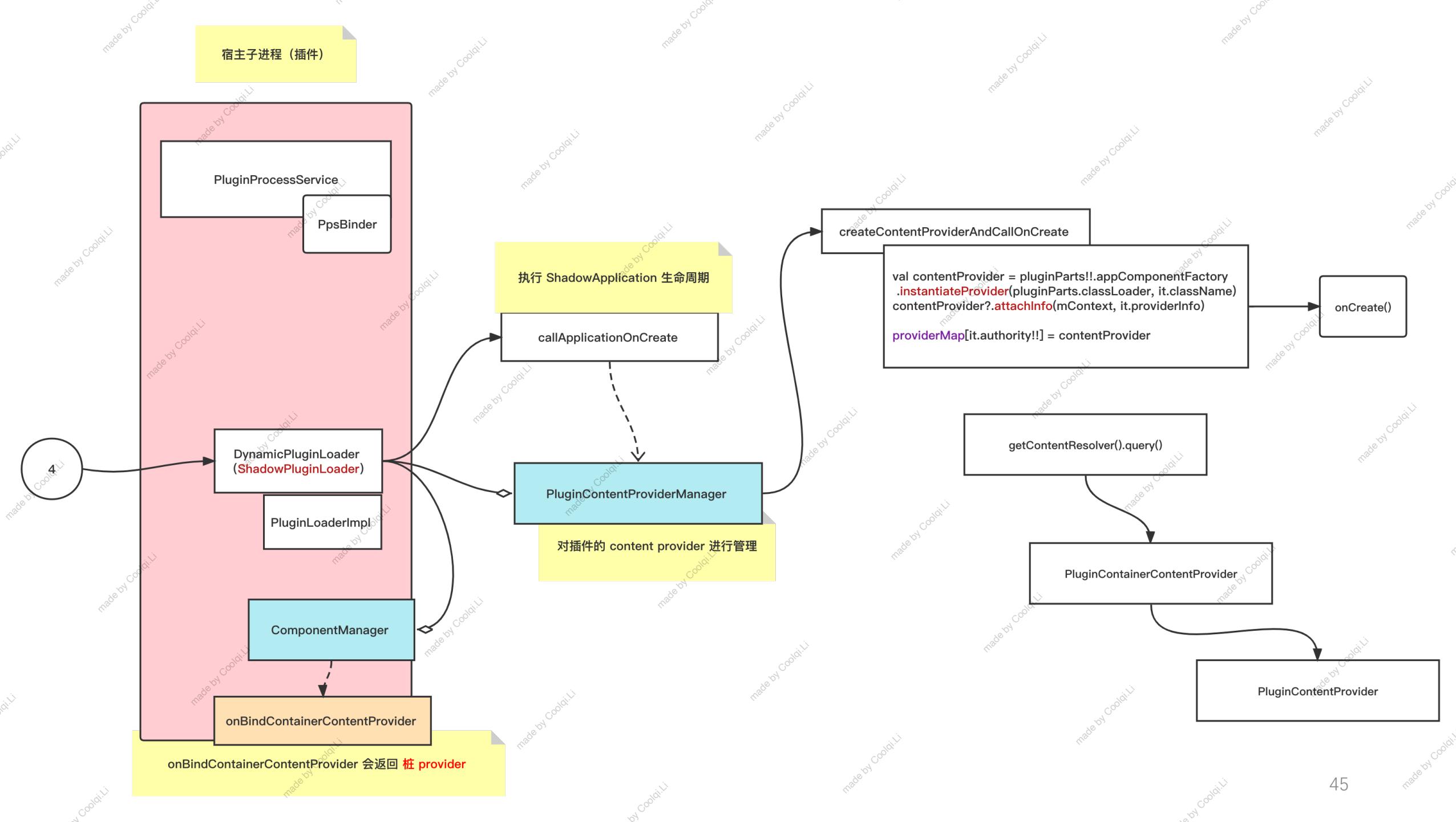
    val tmpShadowDelegate = TmpShadowDelegate()
    mPluginLoader.inject(tmpShadowDelegate, partKey!!)
    val service = tmpShadowDelegate.getAppComponentFactory()
        .instantiateService(tmpShadowDelegate.getPluginClassLoader(), className, intent)

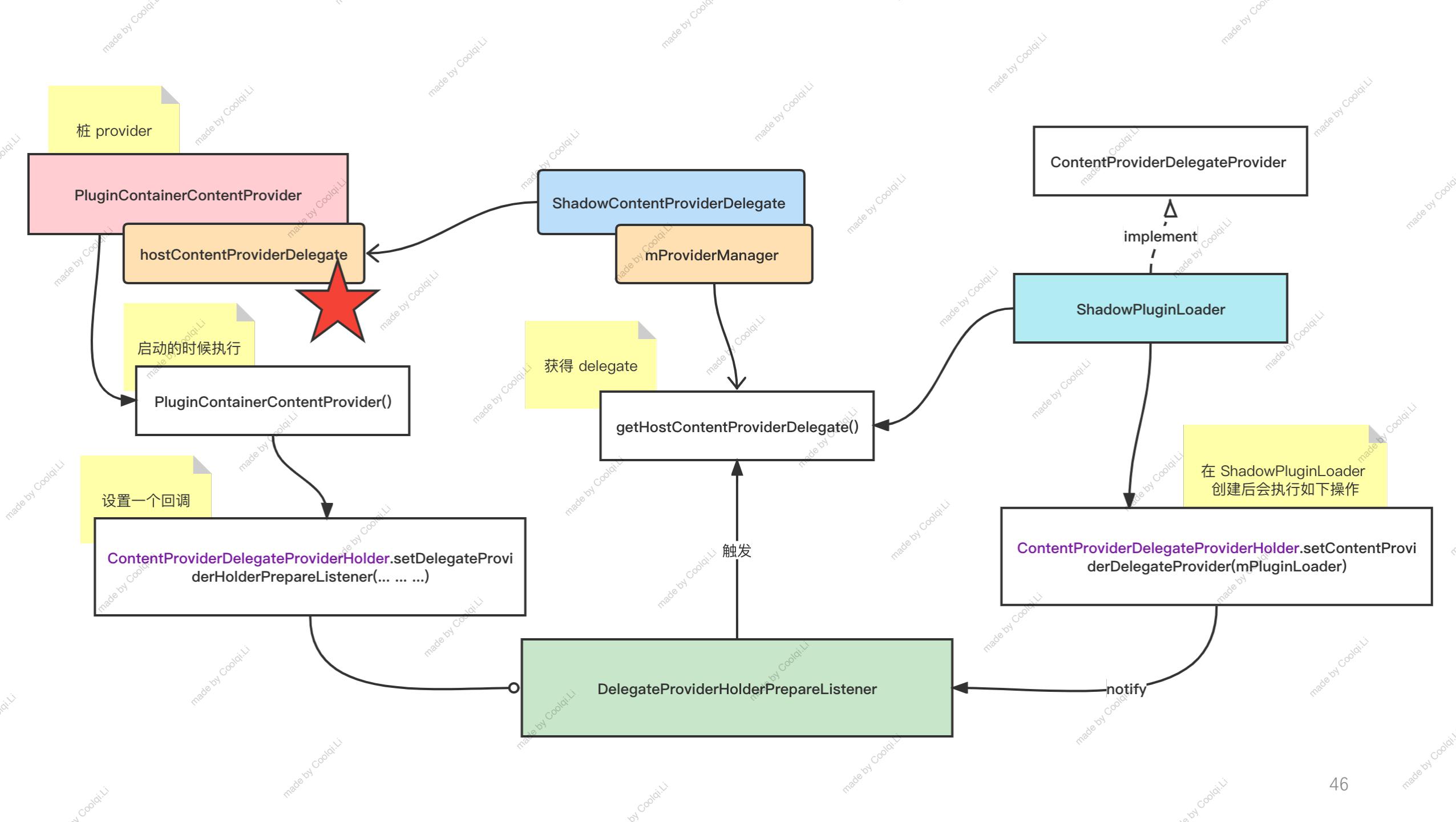
    service.setPluginResources(tmpShadowDelegate.getPluginResources())
    service.setPluginClassLoader(tmpShadowDelegate.getPluginClassLoader())
    service.setShadowApplication(tmpShadowDelegate.getPluginApplication())
    service.setPluginComponentLauncher(tmpShadowDelegate.getComponentManager())
    service.applicationInfo = tmpShadowDelegate.getPluginApplication().applicationInfo
    service.setBusinessName(businessName)
    service.setPluginPartKey(partKey)

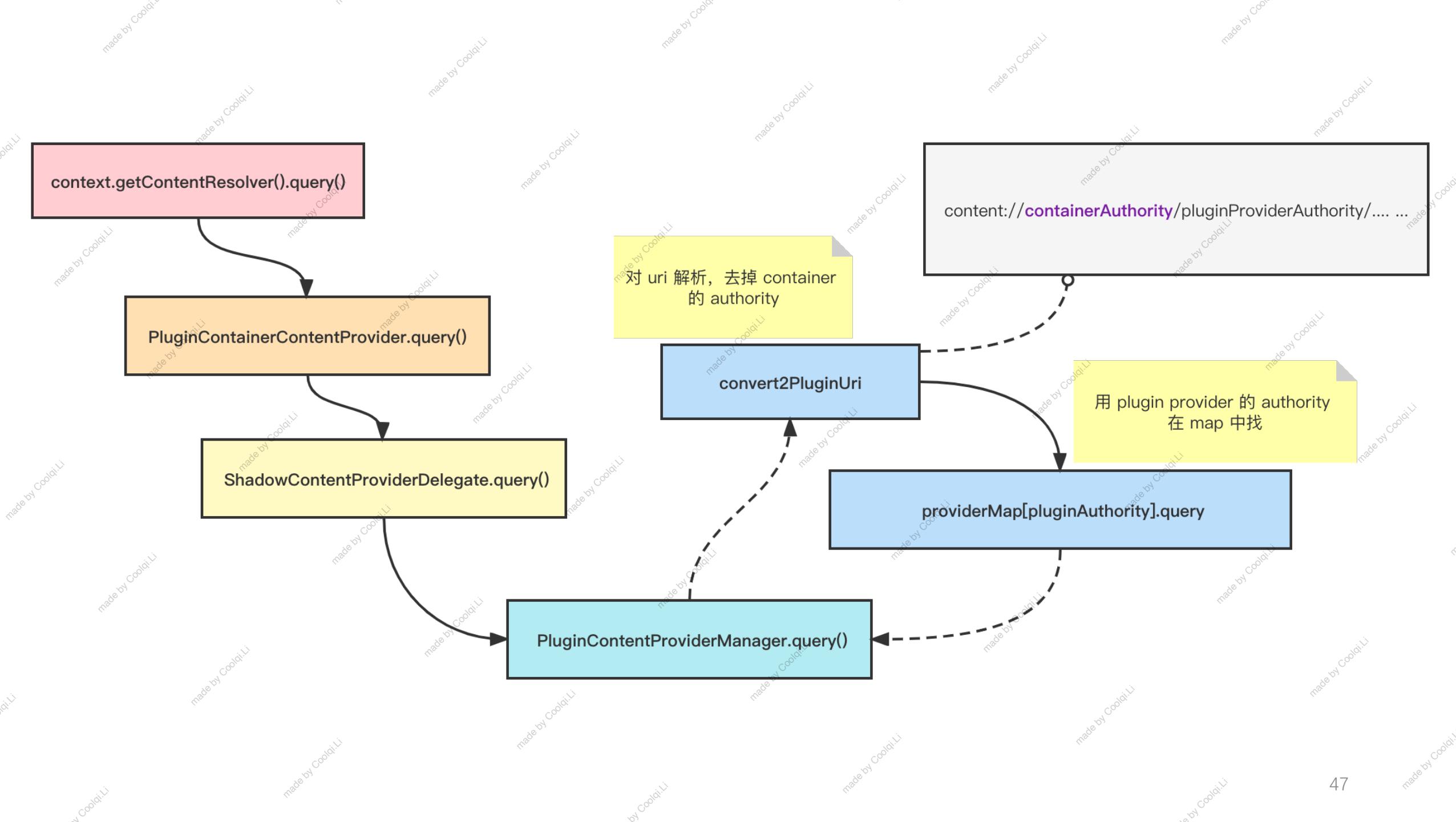
    //和ShadowActivityDelegate.initPluginActivity一样，attachBaseContext放到最后
    service.setHostContextAsBase(mHostContext)
    return service
}
```

宿主如何启动和访问插件 provider in Shadow

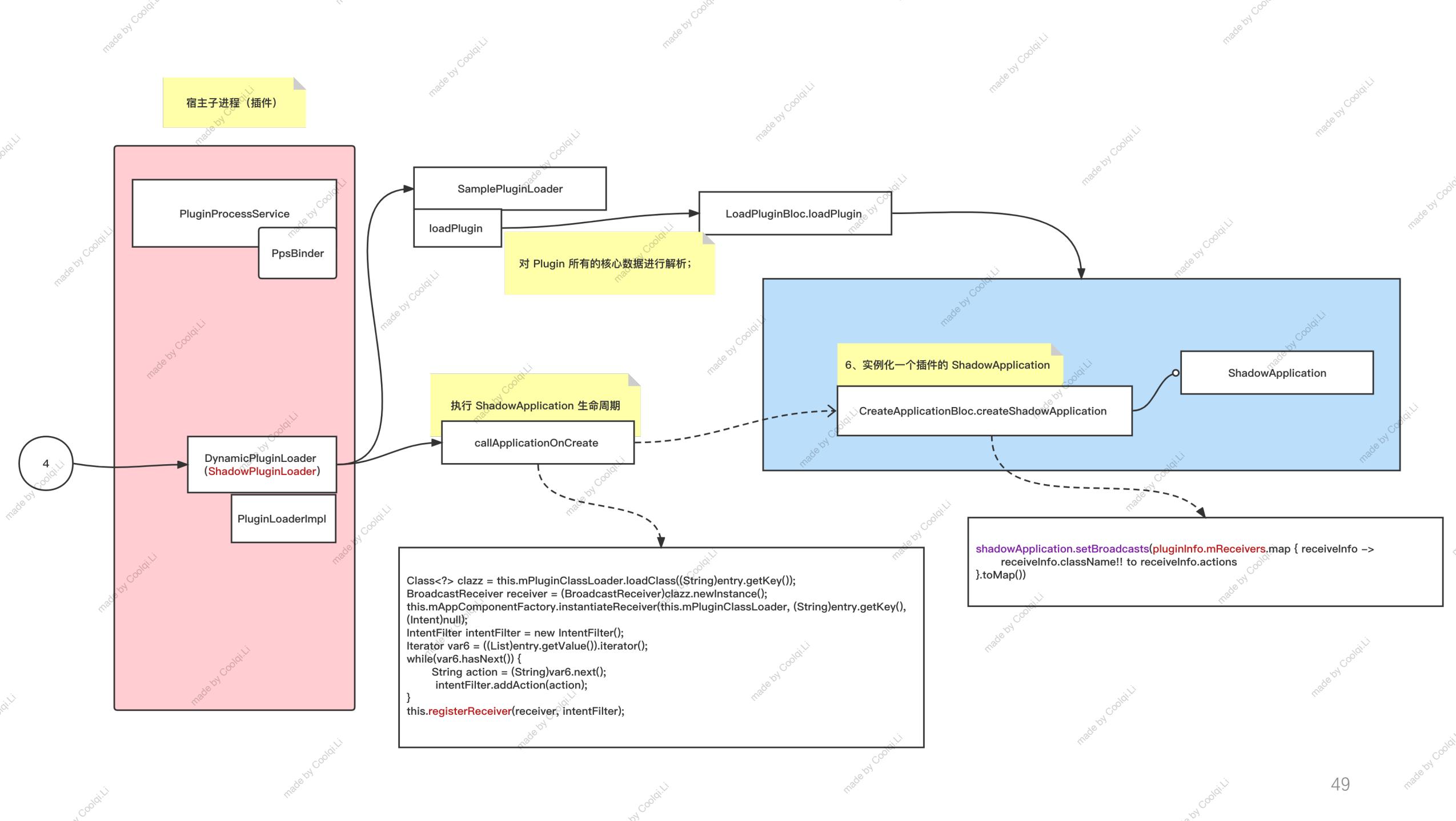








宿主如何启动和访问插件 receiver in Shadow



Gradle Plugin 简析-配置阶段 in Shadow

```
shadow {
    transform {
        useHostContext = ['abc']
    }
}

packagePlugin {
    pluginTypes {
        debug {
            loaderApkConfig = new Tuple2('sample-loader-debug.apk',
                                         ':sample-loader:assembleDebug')
            runtimeApkConfig = new Tuple2('sample-runtime-debug.apk',
                                         ':sample-runtime:assembleDebug')
        }
        pluginApks {
            pluginApk1 {
                businessName = 'sample-plugin-app'
                partKey = 'sample-plugin-app'
                buildTask = ':sample-app:assemblePluginDebug'
                apkPath = 'projects/sample/source/sample-plugin/sample-app
                           /build/outputs/apk/plugin/debug/sample-app-plugin-debug.apk'
                hostWhiteList = ["com.tencent.shadow.sample.host.lib"]
                dependsOn = ['sample-base']
            }
            sampleBase {
                businessName = 'sample-plugin-app'
                partKey = 'sample-base'
                buildTask = ':sample-base:assemblePluginDebug'
                apkPath = 'projects/sample/source/sample-plugin/sample-base
                           /build/outputs/apk/plugin/debug/sample-base-plugin-debug.apk'
                hostWhiteList = ["com.tencent.shadow.sample.host.lib"]
            }
        }
    }
    release {}
}

loaderApkProjectPath = 'projects/sample/source/sample-plugin/sample-loader'
runtimeApkProjectPath = 'projects/sample/source/sample-plugin/sample-runtime'

archiveSuffix = System.getenv("PluginSuffix") ?: ""
archivePrefix = 'plugin'
destinationDir = "${getRootProject().getBuildDir()}""

version = 4
compactVersion = [1, 2, 3]
uuidNickName = "1.1.5"
}
```

```
// ShadowPlugin
override fun apply(project: Project) {
    val shadowExtension = project.extensions
        .create("shadow", ShadowExtension::class.java)
    if (!project.hasProperty("disable_shadow_transform")) {
        baseExtension.registerTransform(ShadowTransform(
            project,
            lateInitBuilder,
            { shadowExtension.transformConfig.useHostContext })
    )
}

addFlavorForTransform(baseExtension)

project.extensions.create("packagePlugin",
    PackagePluginExtension::class.java, project)
}
```

```
// ShadowPlugin
override fun apply(project: Project) {
    project.afterEvaluate {
        initAndroidClassPoolBuilder(baseExtension, project)
        // 1、创建 plugin 的编译 task;;
        createPackagePluginTasks(project)
        // 2、遍历每个变体;;
        onEachPluginVariant(project) { pluginVariant ->
            // 3、检查插件是否修改了资源 ID 分区，不能让宿主和插件的资源 ID 冲突
            checkAaptPackageNameConfig(pluginVariant)

            val appExtension: AppExtension =
                project.extensions.getByType(AppExtension::class.java)

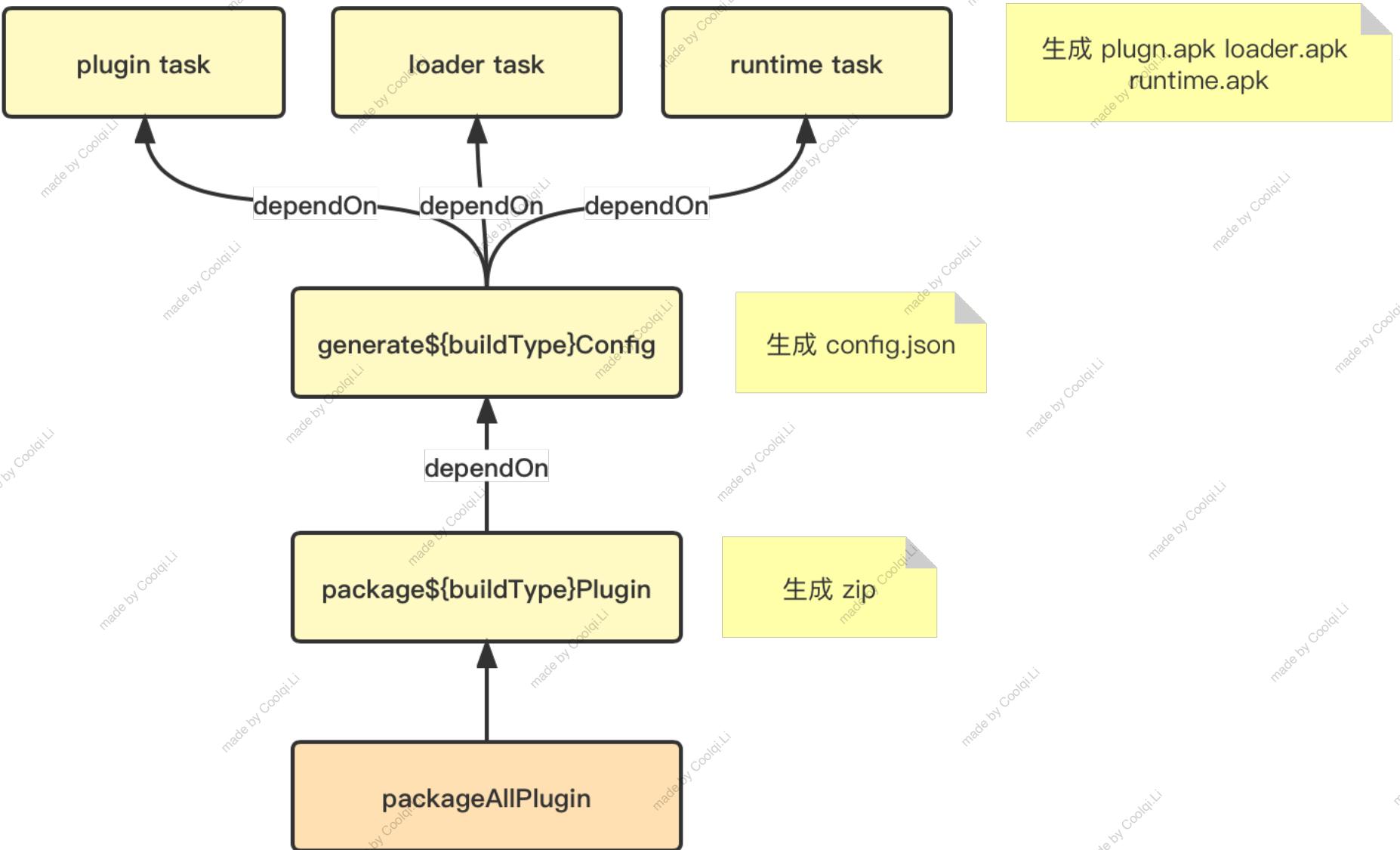
            // 4、创建生成 PluginManifest.java 的任务，用于解析 Manifest.xml
            createGeneratePluginManifestTasks(project,
                appExtension, pluginVariant)
        }
    }
}
```

snappy.io

Gradle Plugin 简析-Task in Shadow

```
private fun createPackagePluginTasks(project: Project) {
    val packagePlugin = project.extensions.findByName("packagePlugin")
    val extension = packagePlugin as PackagePluginExtension
    val buildTypes = extension.buildTypes // debug/release

    val tasks = mutableListOf<Task>()
    for (i in buildTypes) {
        project.logger.info("buildTypes = " + i.name)
        // 创建 package${buildType}Plugin 任务
        val task = createPackagePluginTask(project, i)
        tasks.add(task)
    }
    if (tasks.isNotEmpty()) {
        // 创建 packageAllPlugin 任务, 依赖上面的所有插件
        project.tasks.create("packageAllPlugin") {
            it.group = "plugin"
            it.description = "打包所有插件"
            }.dependsOn(tasks)
    }
}
```



```
/** made by Coolqi.Li
 * 创建生成 PluginManifest.java 的任务
 */
private fun createGeneratePluginManifestTasks(
    project: Project,
    appExtension: AppExtension,
    pluginVariant: ApplicationVariant
) {
    val output = pluginVariant.outputs.first()
    // 1、返回 ManifestProcessorTask, 作用是合并生成 Manifest.xml;
    val processManifestTask = agpCompat.getProcessManifestTask(output)
    val manifestFile = agpCompat.getManifestFile(processManifestTask)
    val variantName = pluginVariant.name
    val outputDir = File(project.buildDir, "generated/source/pluginManifest/$variantName")

    // 1、添加生成 PluginManifest.java 任务;
    val task = project.tasks.register("generate${variantName.capitalize()}") {
        // 依赖 ManifestProcessorTask 任务;
        it.dependsOn(processManifestTask)
        it.inputs.file(manifestFile)
        it.outputs.dir(outputDir).withPropertyName("outputDir")

        val packageForR = agpCompat.getPackageForR(project, variantName)

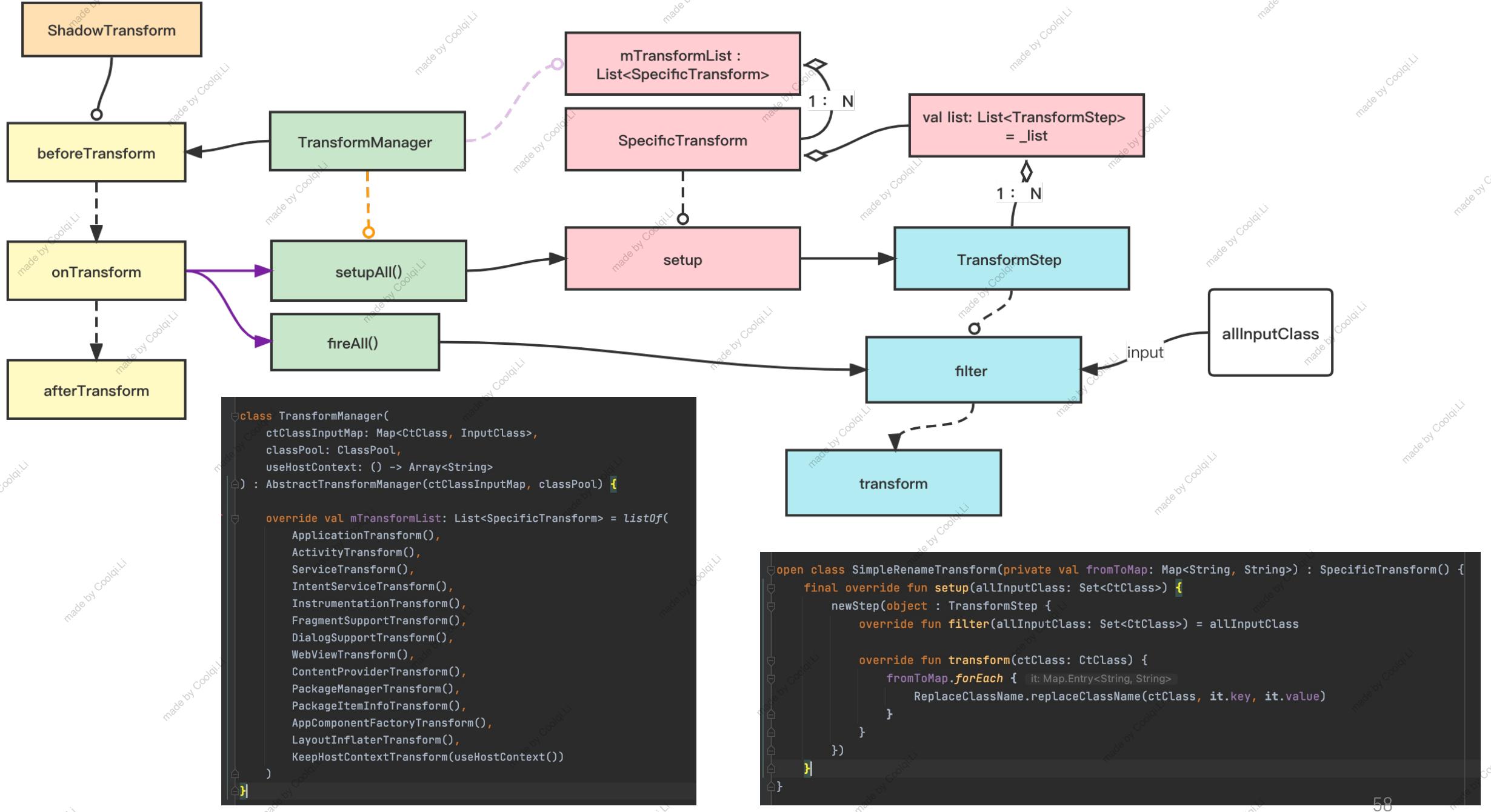
        it.doLast {
            generatePluginManifest(
                manifestFile,
                outputDir,
                "com.tencent.shadow.core.manifest_parser",
                packageForR
            )
        }
    }
    // compile${variantName}JavaWithJavac 依赖这个任务;;
    project.tasks.getByName("compile${variantName.capitalize()}JavaWithJavac")
        .dependsOn(task)

    // 把 PluginManifest.java 所在路径添加为源码路径;;
    appExtension.sourceSets.getByName(variantName).java.srcDir(outputDir)
}
```

```
fun generatePluginManifest(
    xmlFile: File,
    outputDir: File,
    packageName: String,
    packageForR: String
) {
    val androidManifest = AndroidManifestReader().read(xmlFile)
    val generator = PluginManifestGenerator(packageForR)
    generator.generate(androidManifest, outputDir, packageName)
}
```

snappyf.io

Gradle Plugin 简析- Transformer in Shadow



简单的 SpecificTransform in Shadow

```
class AppComponentFactoryTransform : SimpleRenameTransform(  
    mapOf(  
        "android.app.AppComponentFactory"  
            to "com.tencent.shadow.core.runtime.ShadowAppComponentFactory"  
    )  
)
```

```
class ServiceTransform : SimpleRenameTransform(  
    mapOf(  
        "android.app.Service"  
            to "com.tencent.shadow.core.runtime.ShadowService"  
    )  
)
```

```
class IntentServiceTransform : SimpleRenameTransform(  
    mapOf("android.app.IntentService" to "com.tencent.shadow.core.runtime.ShadowIntentService")  
)
```

```
class ActivityTransform : SimpleRenameTransform(  
    mapOf(  
        "android.app.Activity"  
            to "com.tencent.shadow.core.runtime.ShadowActivity",  
        "android.app.NativeActivity"  
            to "com.tencent.shadow.core.runtime.ShadowNativeActivity"  
    )  
)
```

```
class ApplicationTransform : SimpleRenameTransform(  
    mapOf(  
        "android.app.Application"  
            to "com.tencent.shadow.core.runtime.ShadowApplication",  
        "android.app.Application\$ActivityLifecycleCallbacks"  
            to "com.tencent.shadow.core.runtime.ShadowActivityLifecycleCallbacks"  
    )  
)
```

复杂的 SpecificTransform in Shadow

ContentProviderTransform

```
// call 和 parseCall
public static Bundle call(ContentResolver resolver, Uri uri,
    String method, String arg, Bundle extras) {
    if (extras == null) {
        extras = new Bundle();
    }
    Uri containerUri = UriConverter.parseCall(uri.toString(), extras);
    return resolver.call(containerUri, method, arg, extras);
}

public static Uri parseCall(String uriString, Bundle bundle) {
    if (sUriParseDelegate != null) {
        return sUriParseDelegate.parseCall(uriString, bundle);
    } else {
        return Uri.parse(uriString);
    }
}

// parse
public static Uri parse(String uriString) {
    if (sUriParseDelegate != null) {
        return sUriParseDelegate.parse(uriString);
    } else {
        return Uri.parse(uriString);
    }
}
```

目标

替换

android.net.Uri
parse

android.net.Uri\$Builder
build

android.content.ContentResolver
call

com.tencent.shadow.core.runtime.UriConverter
notifyChange

```
abstract class ShadowPluginLoader(hostAppContext: Context)
    : DelegateProvider, DI, ContentProviderDelegateProvider {

    private val mPluginContentProviderManager: PluginContentProviderManager =
        PluginContentProviderManager()

    init {
        UriConverter.setUriParseDelegate(mPluginContentProviderManager)
    }
}
```

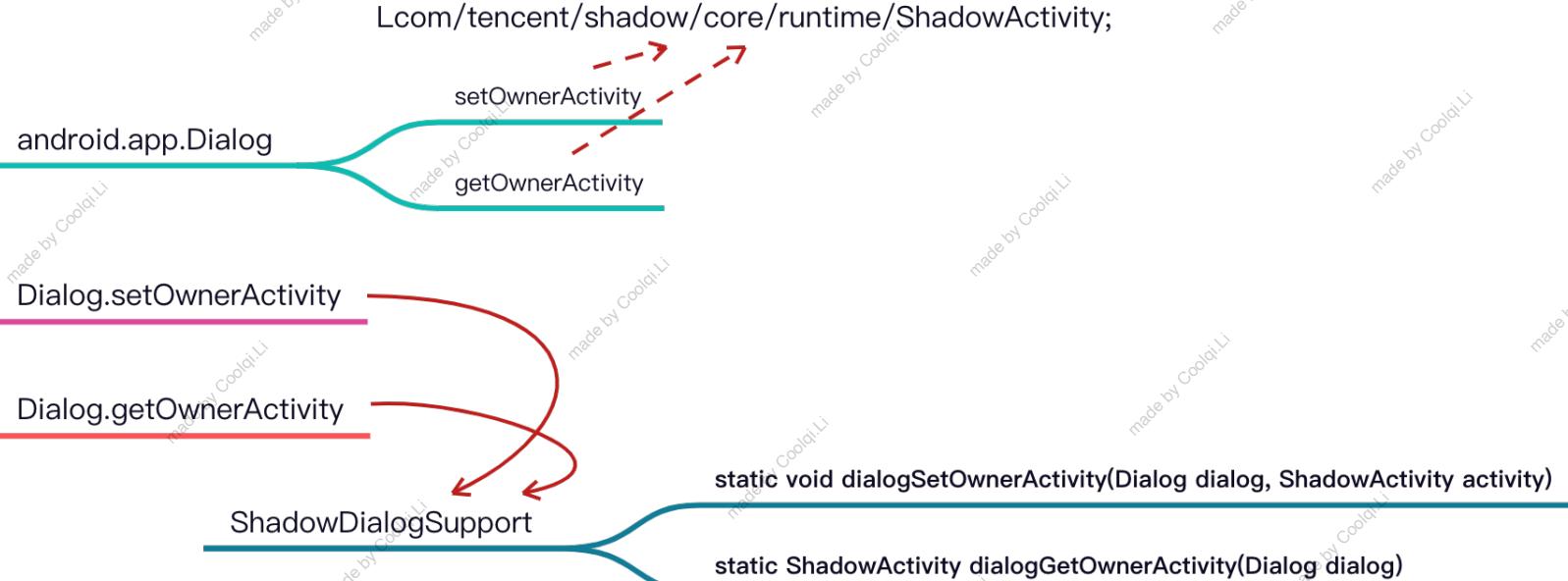
snappyf.io

```
// PluginContentProviderManager
override fun parse(uriString: String): Uri {
    if (uriString.startsWith(CONTENT_PREFIX)) {
        val uriContent = uriString.substring(CONTENT_PREFIX.length)
        val index = uriContent.indexOf("/")
        val originalAuthority = if (index != -1) uriContent.substring(0, index) else uriContent
        val containerAuthority = getContainerProviderAuthority(originalAuthority)
        if (containerAuthority != null) {
            // content://
            return Uri.parse("$CONTENT_PREFIX$containerAuthority/$uriContent")
        }
    }
    return Uri.parse(uriString)
}

override fun parseCall(uriString: String, extra: Bundle): Uri {
    val pluginUri = parse(uriString)
    extra.putString(SHADOW_BUNDLE_KEY, pluginUri.toString())
    return pluginUri
}
```

DialogSupportTransform

FragmentSupportTransform



```
public class ShadowDialogSupport {

    public static void dialogSetOwnerActivity(Dialog dialog, ShadowActivity activity) {
        Activity hostActivity = (Activity) activity
            .hostActivityDelegator.getHostActivity();
        dialog.setOwnerActivity(hostActivity);
    }

    public static ShadowActivity dialogGetOwnerActivity(Dialog dialog) {
        PluginContainerActivity ownerActivity =
            (PluginContainerActivity) dialog.getOwnerActivity();
        if (ownerActivity != null) {
            return (ShadowActivity) PluginActivity.get(ownerActivity);
        } else {
            return null;
        }
    }
}
```

LayoutInflaterTransform

替换调用

android.view.LayoutInflater

LayoutInflater.getFactory

LayoutInflater.getFactory2

ShadowLayoutInflater

static Factory getOriginalFactory(LayoutInflater inflater)

static Factory2 getOriginalFactory2(LayoutInflater inflater)

```
@Override  
public Object getSystemService(String name) {  
    if (LAYOUT_INFLATER_SERVICE.equals(name)) {  
        if (mLayoutInflater == null) {  
            LayoutInflater inflater = (LayoutInflater) super.getSystemService(name);  
            mLayoutInflater = ShadowLayoutInflater.build(inflater, newContext: this, mPartKey);  
        }  
        return mLayoutInflater;  
    }  
    return super.getSystemService(name);  
}
```

```
public static Factory getOriginalFactory(LayoutInflater inflater) {  
    if (inflater instanceof ShadowLayoutInflater) {  
        return ((ShadowLayoutInflater) inflater).mOriginalFactory;  
    } else {  
        return inflater.getFactory();  
    }  
}  
  
public static Factory2 getOriginalFactory2(LayoutInflater inflater) {  
    if (inflater instanceof ShadowLayoutInflater) {  
        return ((ShadowLayoutInflater) inflater).mOriginalFactory2;  
    } else {  
        return inflater.getFactory2();  
    }  
}
```

PendingIntentTransform

PackageManagerTransform

替换方法

static ApplicationInfo **getApplicationInfo**(ClassLoader classLoaderOfInvokeCode,
String packageName, int flags)

static PluginPackageManager **getPluginPackageManager**(ClassLoader classLoaderOfInvokeCode)

```
public static PluginPackageManager getPluginPackageManager(ClassLoader classLoaderOfInvokeCode) {  
    return PluginPartInfoManager.getPluginInfo(classLoaderOfInvokeCode).packageManager;  
}  
  
public static ApplicationInfo getApplicationInfo(ClassLoader classLoaderOfInvokeCode, String packageName, int flags) {  
    return getPluginPackageManager(classLoaderOfInvokeCode).getApplicationInfo(packageName, flags);  
}
```

```
// javassist 生成的代码  
public static ApplicationInfo getApplicationInfo_shadow(  
    PackageManager pm,  
    String packageName,  
    int flags) {  
    Classloader classloader = this.getClass().getClassLoader();  
    return PackageManagerInvokeRedirect  
        .getApplicationInfo(classloader, packageName, flags);  
}
```

snappify.io

```
@SuppressLint("WrongConstant")  
internal class PluginPackageManagerImpl(  
    private val pluginApplicationInfoFromPluginManifest: ApplicationInfo,  
    private val pluginArchiveFilePath: String,  
    private val componentManager: ComponentManager,  
    private val hostPackageManager: PackageManager  
) : PluginPackageManager {  
    override fun getApplicationInfo(packageName: String, flags: Int): ApplicationInfo =  
        if (packageName.isPlugin()) {  
            getPluginApplicationInfo(flags)  
        } else {  
            hostPackageManager.getApplicationInfo(packageName, flags)  
        }  
}
```

参考文献 about Shadow

Shadow 官网
<https://github.com/Tencent/Shadow>

Shadow 作者本人的博客：
<https://juejin.cn/user/536217405890903>