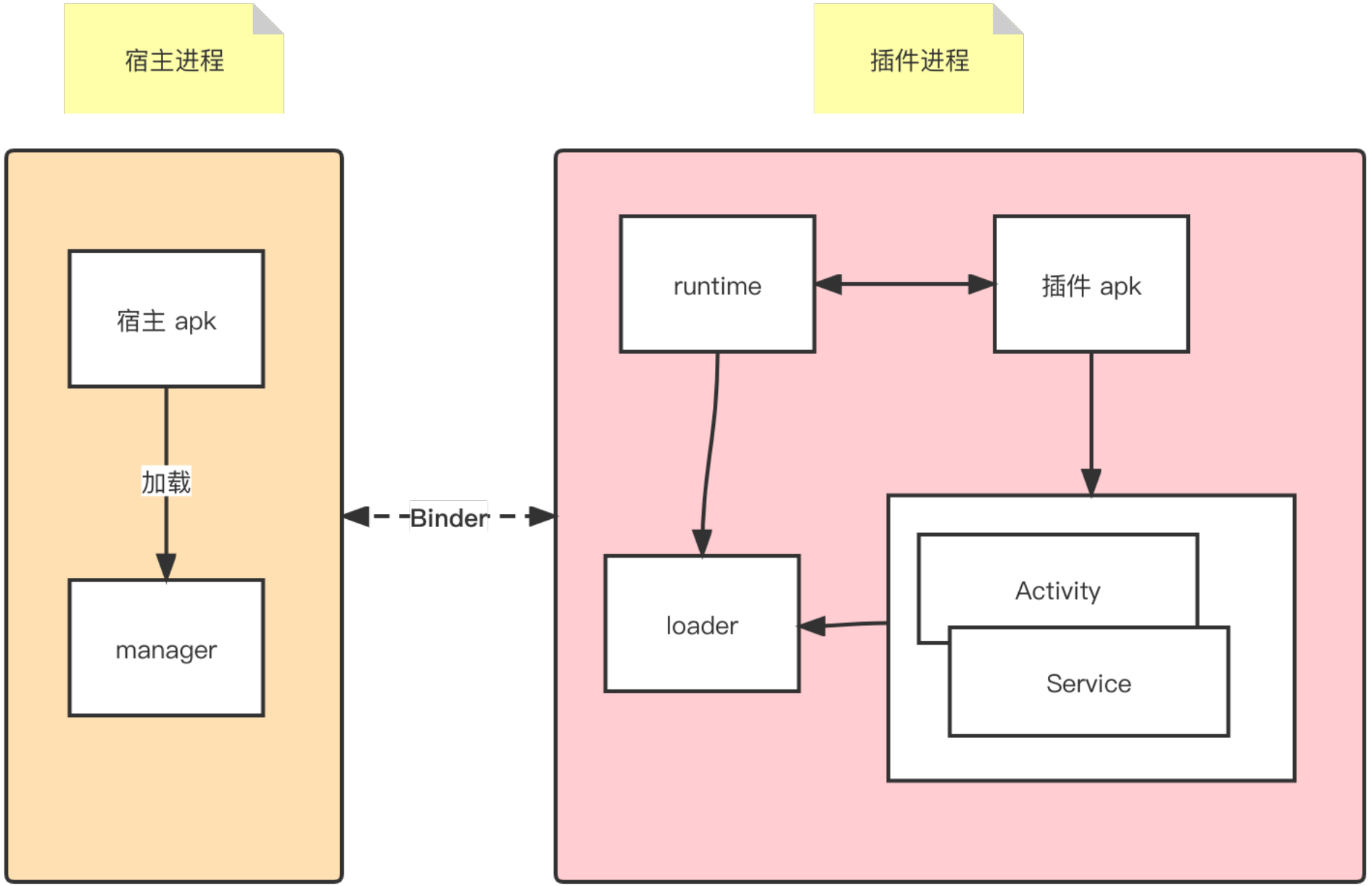
插件化 shadow 原理分享

李帅奇

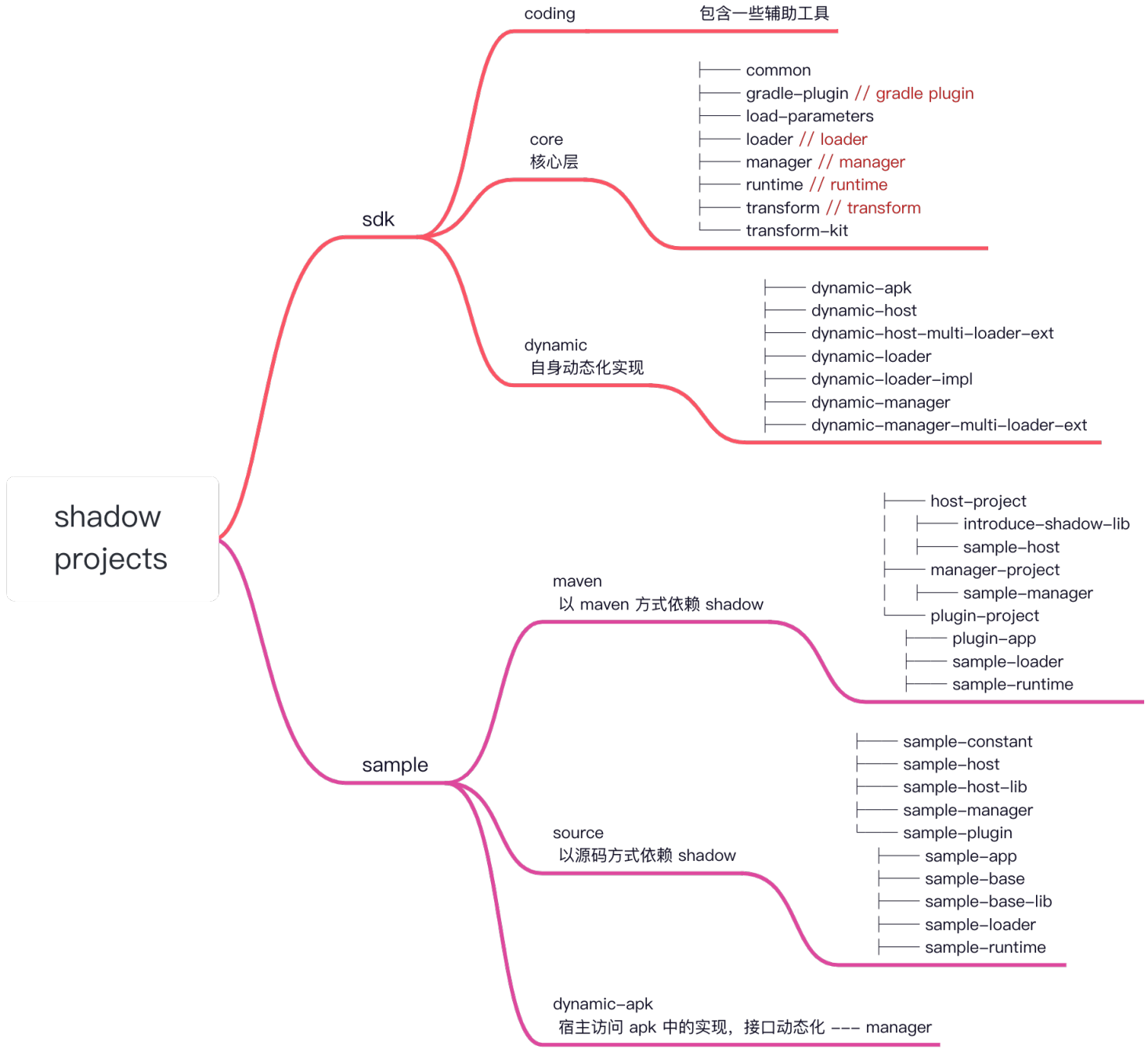
原理总览

in Shadow


宿主进程
插件进程
宿主 apk
加载
manager
Binder
runtime
插件 apk
loader
Activity
Service

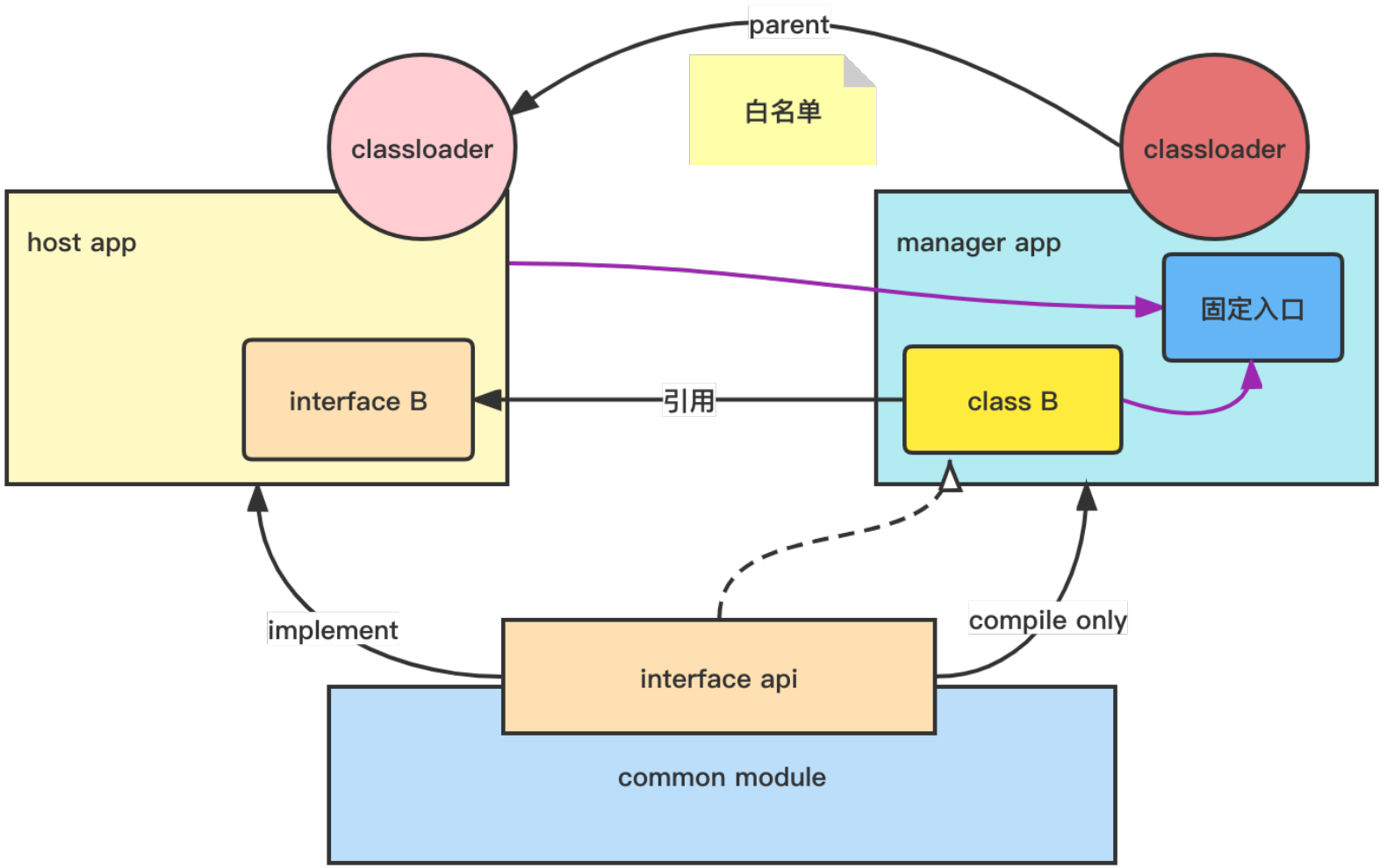
工程结构

in Shadow


shadow projects
sdk
coding
包含一些辅助工具
core
核心层
common
gradle-plugin // gradle plugin
load-parameters
loader // loader
manager // manager
runtime // runtime
transform // transform
transform-kit
dynamic
自身动态化实现
dynamic-apk
dynamic-host
dynamic-host-multi-loader-ext
dynamic-loader
dynamic-loader-impl
dynamic-manager
dynamic-manager-multi-loader-ext
sample
maven
以 maven 方式依赖 shadow
host-project
introduce-shadow-lib
sample-host
manager-project
sample-manager
plugin-project
plugin-app
sample-loader
sample-runtime
source
以源码方式依赖 shadow
sample-constant
sample-host
sample-host-lib
sample-manager
sample-plugin
sample-app
sample-base
sample-base-lib
sample-loader
sample-runtime
dynamic-apk
宿主访问 apk 中的实现，接口动态化 --- manager

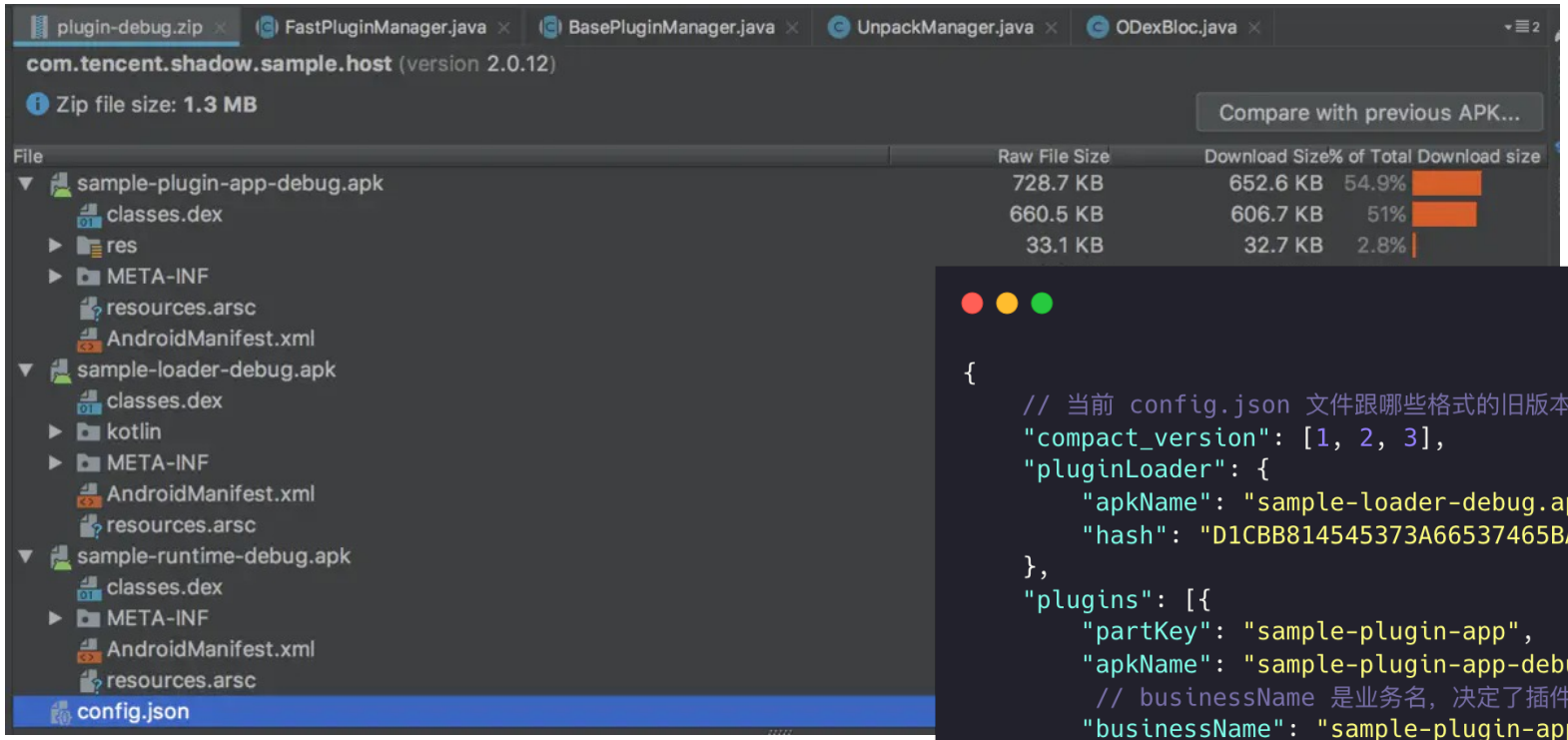
接口动态化原理

in shadow


parent
白名单
classloader
classloader
host app
manager app
固定入口
interface B
引用
class B
implement
compile only
interface api
common module

插件包的组成

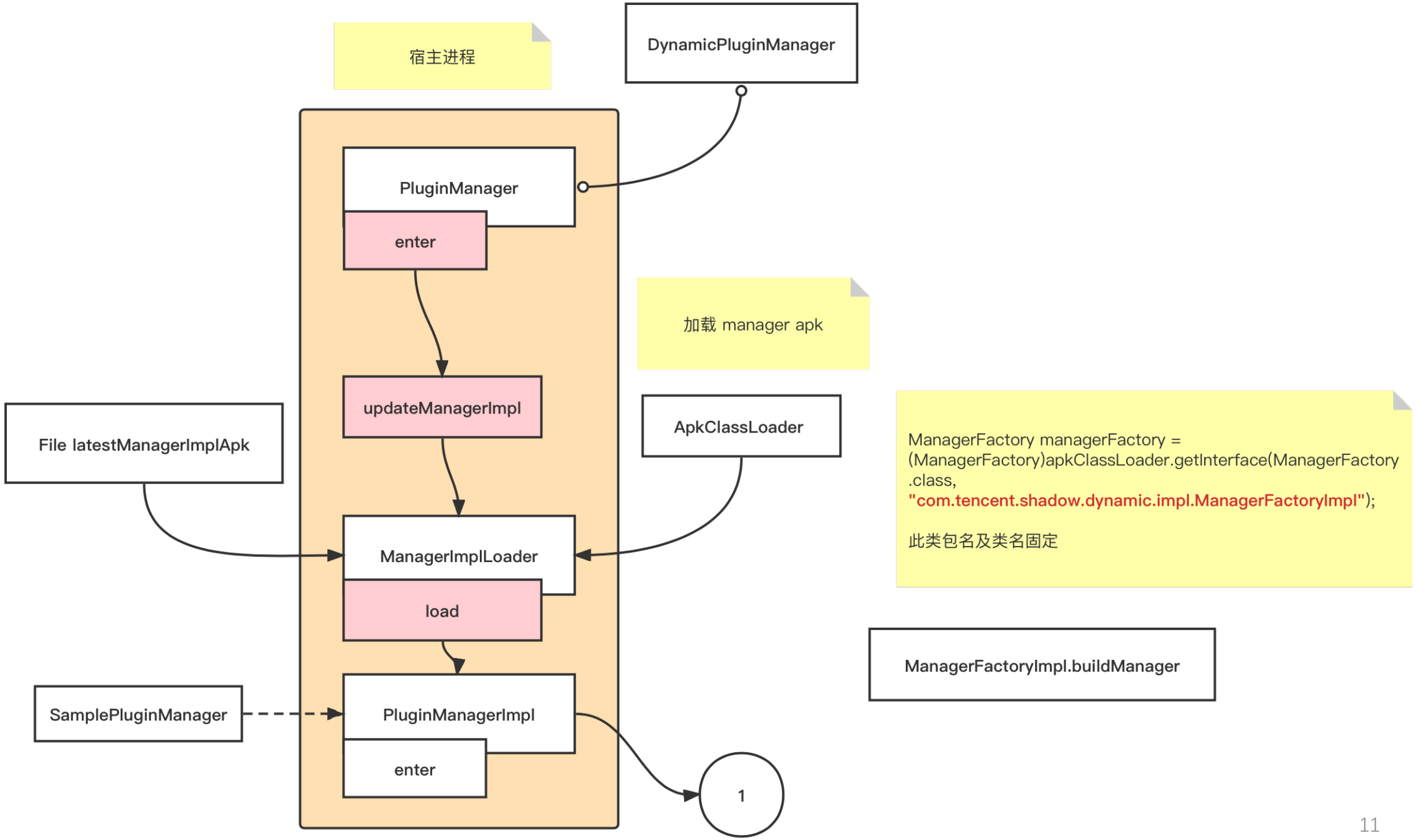
in Shadow

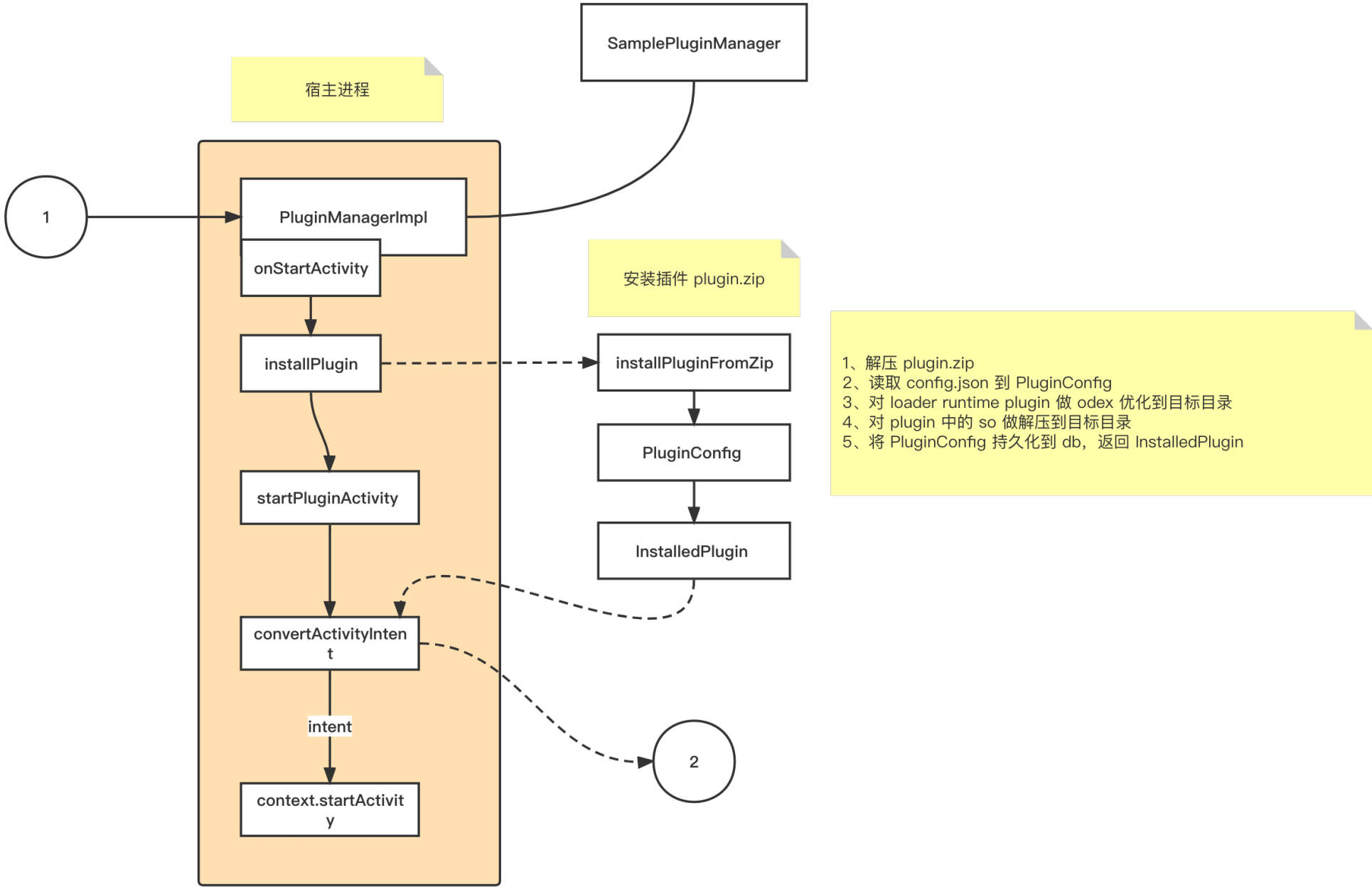


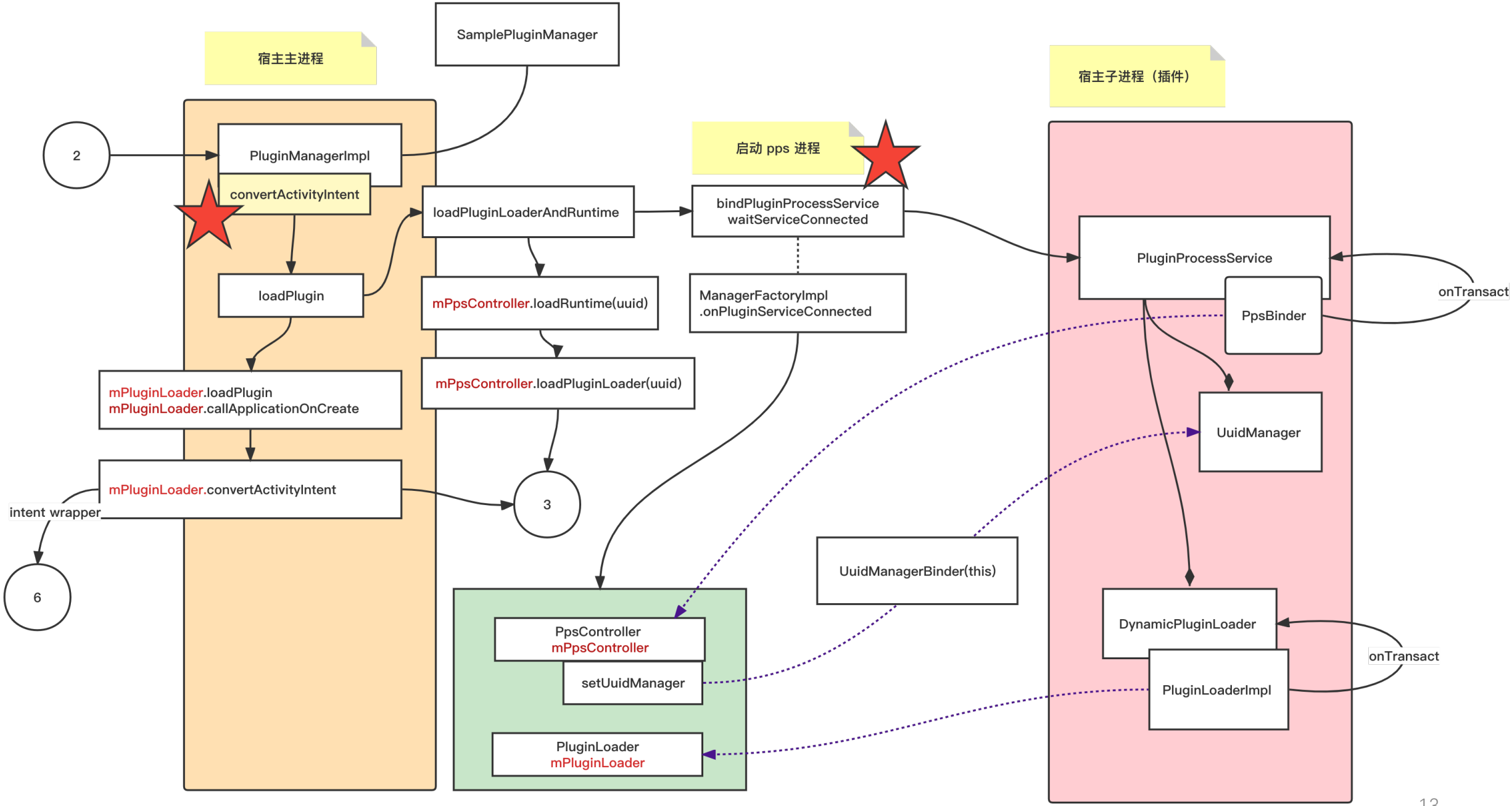

```
{
    // 当前 config.json 文件跟哪些格式的旧版本是兼容的，可以被支持旧版本格式的 Manager 使用
    "compact_version": [1, 2, 3],
    "pluginLoader": {
        "apkName": "sample-loader-debug.apk",
        "hash": "D1CBB814545373A66537465BAD8F37B5"
    },
    "plugins": [{
        "partKey": "sample-plugin-app",
        "apkName": "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"
}
```

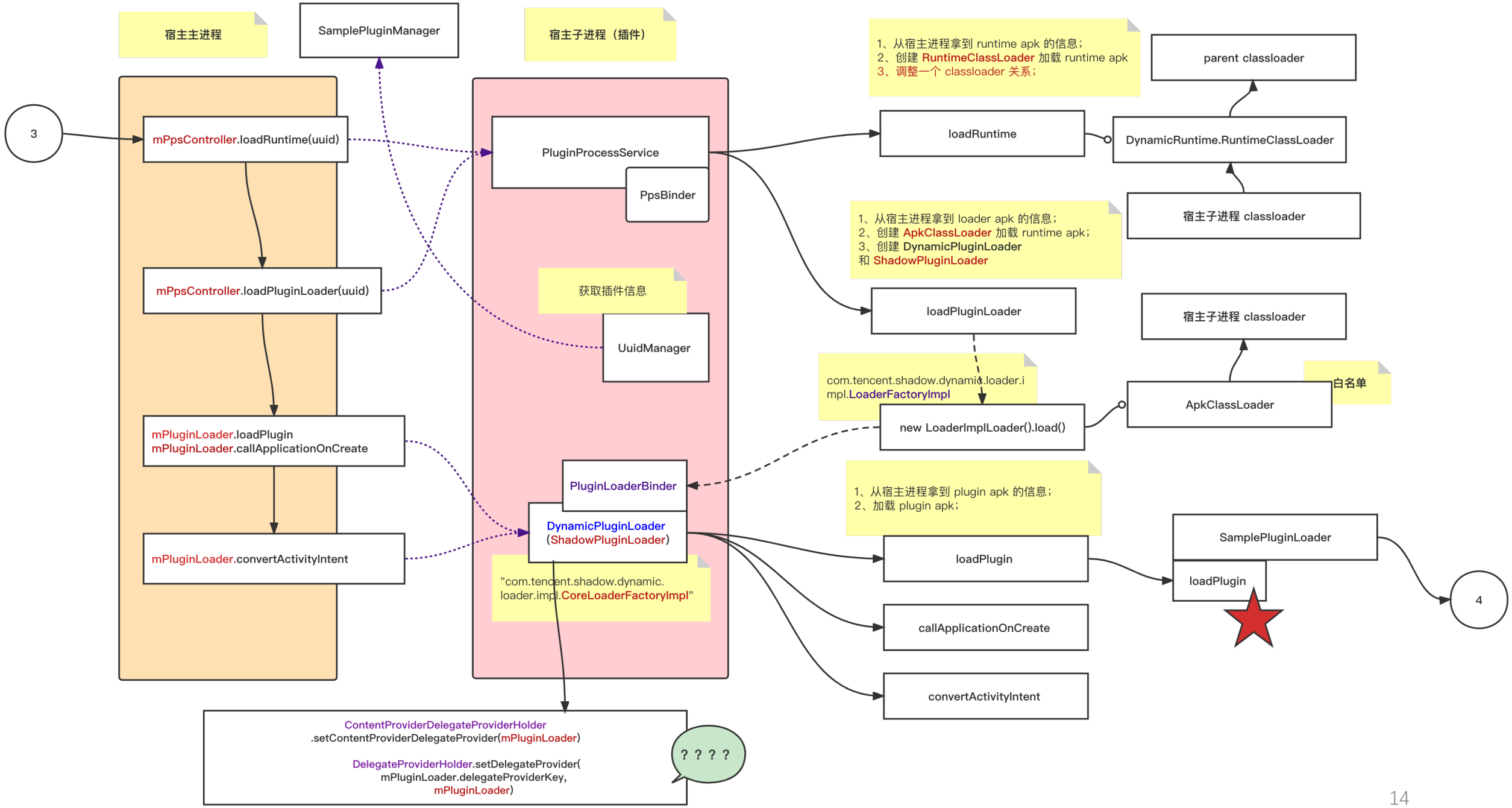
宿主如何启动插件 Activity

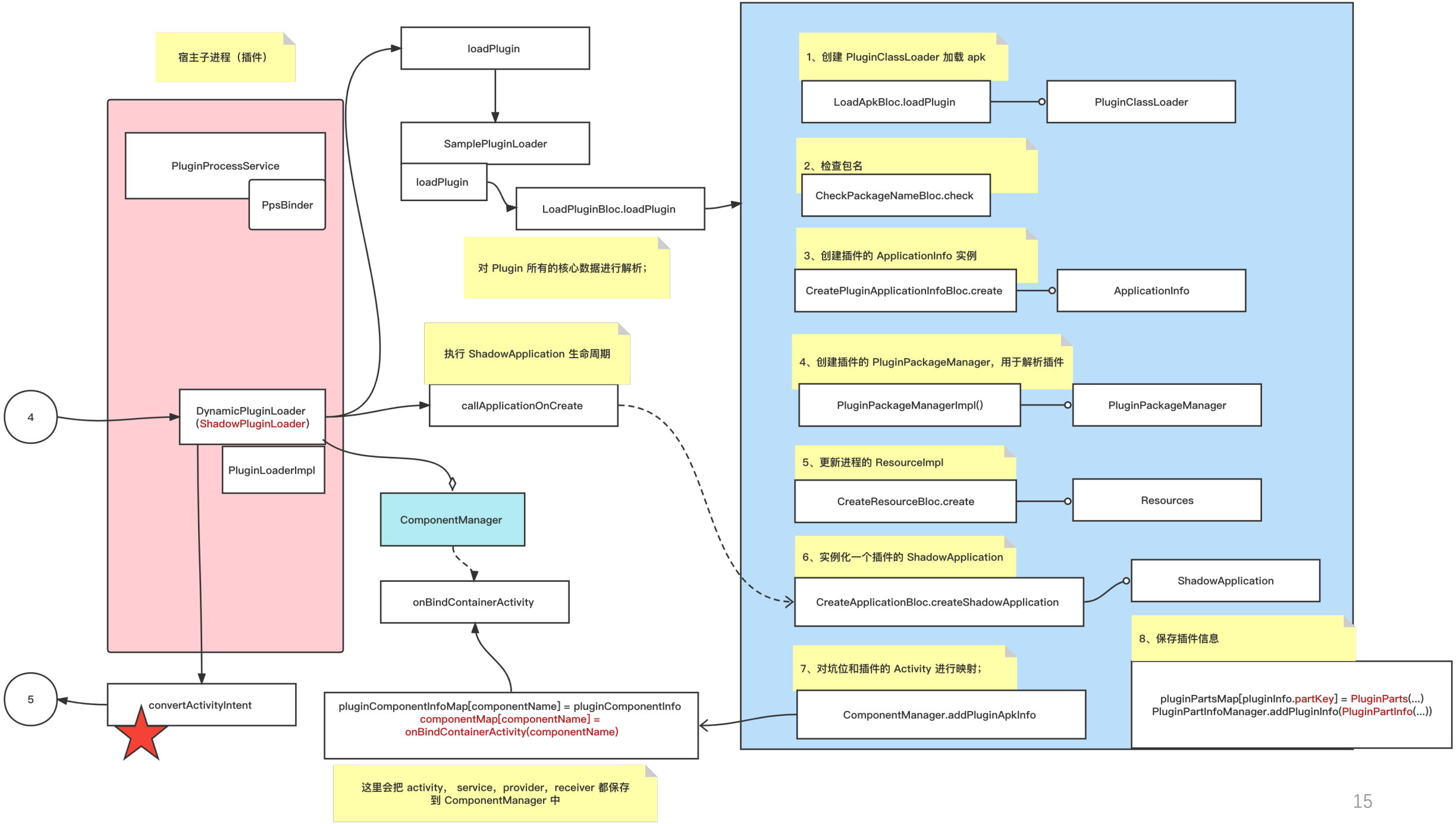
in Shadow

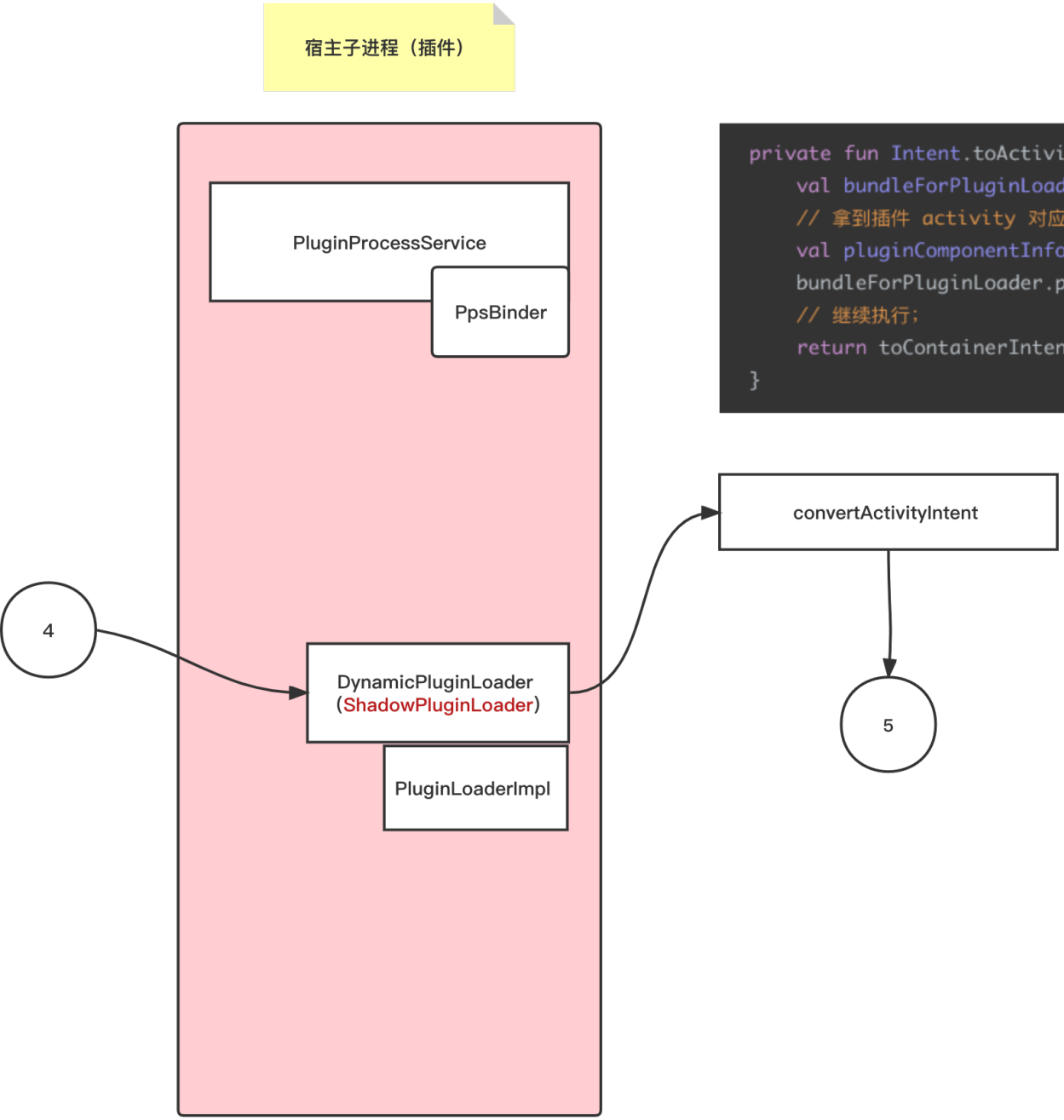

宿主进程
DynamicPluginManager
PluginManager
enter
加载 manager apk
updateManagerImpl
File latestManagerImplApk
ApkClassLoader
ManagerImplLoader
load
ManagerFactory managerFactory =
(ManagerFactory)apkClassLoader.getInterface(ManagerFactory
.class,
"com.tencent.shadow.dynamic.impl.ManagerFactoryImpl");
此类包名及类名固定
ManagerFactoryImpl.buildManager
SamplePluginManager
PluginManagerImpl
enter
1


SamplePluginManager
宿主进程
1
PluginManagerImpl
onStartActivity
installPlugin
startPluginActivity
convertActivityIntent
intent
context.startActivity
2
安装插件 plugin.zip
installPluginFromZip
PluginConfig
InstalledPlugin
1、解压 plugin.zip
2、读取 config.json 到 PluginConfig
3、对 loader runtime plugin 做 odex 优化到目标目录
4、对 plugin 中的 so 做解压到目标目录
5、将 PluginConfig 持久化到 db，返回 InstalledPlugin


宿主主进程
SamplePluginManager
2
PluginManagerImpl
convertActivityIntent
loadPlugin
mPluginLoader.loadPlugin
mPluginLoader.callApplicationOnCreate
mPluginLoader.convertActivityIntent
intent wrapper
6
3
loadPluginLoaderAndRuntime
mPpsController.loadRuntime(uuid)
mPpsController.loadPluginLoader(uuid)
启动 pps 进程
bindPluginProcessService
waitServiceConnected
ManagerFactoryImpl
.onPluginServiceConnected
PpsController
mPpsController
setUuidManager
PluginLoader
mPluginLoader
UuidManagerBinder(this)
宿主子进程（插件）
PluginProcessService
PpsBinder
onTransact
UuidManager
DynamicPluginLoader
PluginLoaderImpl
onTransact

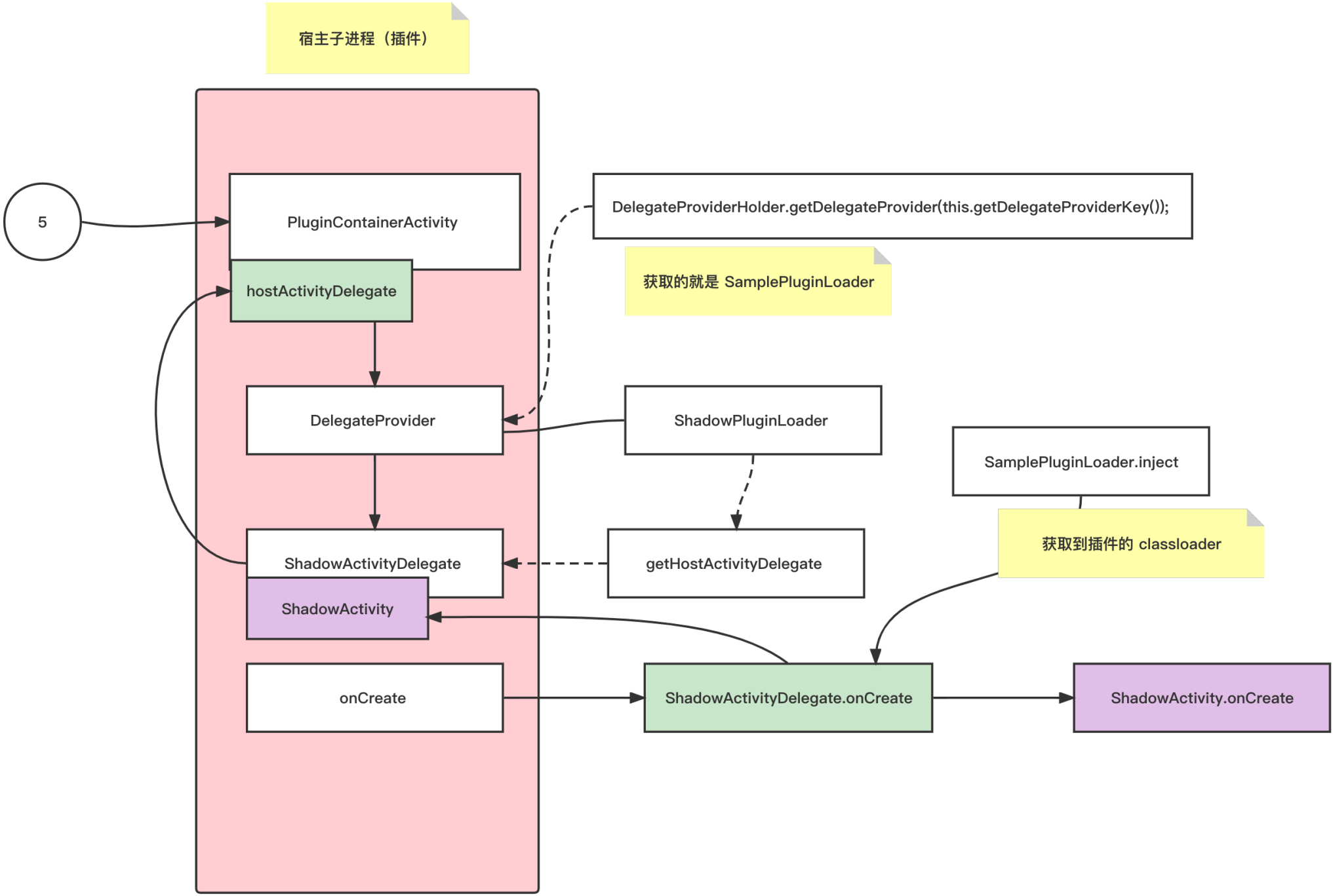

宿主主进程
SamplePluginManager
宿主子进程（插件）
3
mPpsController.loadRuntime(uuid)
PluginProcessService
PpsBinder
1、从宿主进程拿到 runtime apk 的信息；
2、创建 RuntimeClassLoader 加载 runtime apk
3、调整一个 classloader 关系；
parent classloader
loadRuntime
DynamicRuntime.RuntimeClassLoader
宿主子进程 classloader
mPpsController.loadPluginLoader(uuid)
获取插件信息
UuidManager
1、从宿主进程拿到 loader apk 的信息；
2、创建 ApkClassLoader 加载 runtime apk；
3、创建 DynamicPluginLoader
和 ShadowPluginLoader
loadPluginLoader
宿主子进程 classloader
mPluginLoader.loadPlugin
mPluginLoader.callApplicationOnCreate
com.tencent.shadow.dynamic.loader.impl.LoaderFactoryImpl
白名单
ApkClassLoader
new LoaderImplLoader().load()
PluginLoaderBinder
DynamicPluginLoader
(ShadowPluginLoader)
1、从宿主进程拿到 plugin apk 的信息；
2、加载 plugin apk；
mPluginLoader.convertActivityIntent
"com.tencent.shadow.dynamic.loader.impl.CoreLoaderFactoryImpl"
loadPlugin
SamplePluginLoader
loadPlugin
4
callApplicationOnCreate
convertActivityIntent
ContentProviderDelegateProviderHolder
.setContentProviderDelegateProvider(mPluginLoader)
DelegateProviderHolder.setDelegateProvider(
mPluginLoader.delegateProviderKey,
mPluginLoader)
? ? ? ?


宿主子进程（插件）
loadPlugin
SamplePluginLoader
loadPlugin
LoadPluginBloc.loadPlugin
PluginProcessService
PpsBinder
对 Plugin 所有的核心数据进行解析；
执行 ShadowApplication 生命周期
callApplicationOnCreate
4
DynamicPluginLoader
(ShadowPluginLoader)
PluginLoaderImpl
ComponentManager
onBindContainerActivity
5
convertActivityIntent
pluginComponentInfoMap[componentName] = pluginComponentInfo
componentMap[componentName] =
onBindContainerActivity(componentName)
这里会把 activity， service，provider，receiver 都保存
到 ComponentManager 中
1、创建 PluginClassLoader 加载 apk
LoadApkBloc.loadPlugin
PluginClassLoader
2、检查包名
CheckPackageNameBloc.check
3、创建插件的 ApplicationInfo 实例
CreatePluginApplicationInfoBloc.create
ApplicationInfo
4、创建插件的 PluginPackageManager，用于解析插件
PluginPackageManagerImpl()
PluginPackageManager
5、更新进程的 ResourceImpl
CreateResourceBloc.create
Resources
6、实例化一个插件的 ShadowApplication
CreateApplicationBloc.createShadowApplication
ShadowApplication
7、对坑位和插件的 Activity 进行映射；
ComponentManager.addPluginApkInfo
8、保存插件信息
pluginPartsMap[pluginInfo.partKey] = PluginParts(...)
PluginPartInfoManager.addPluginInfo(PluginPartInfo(...))



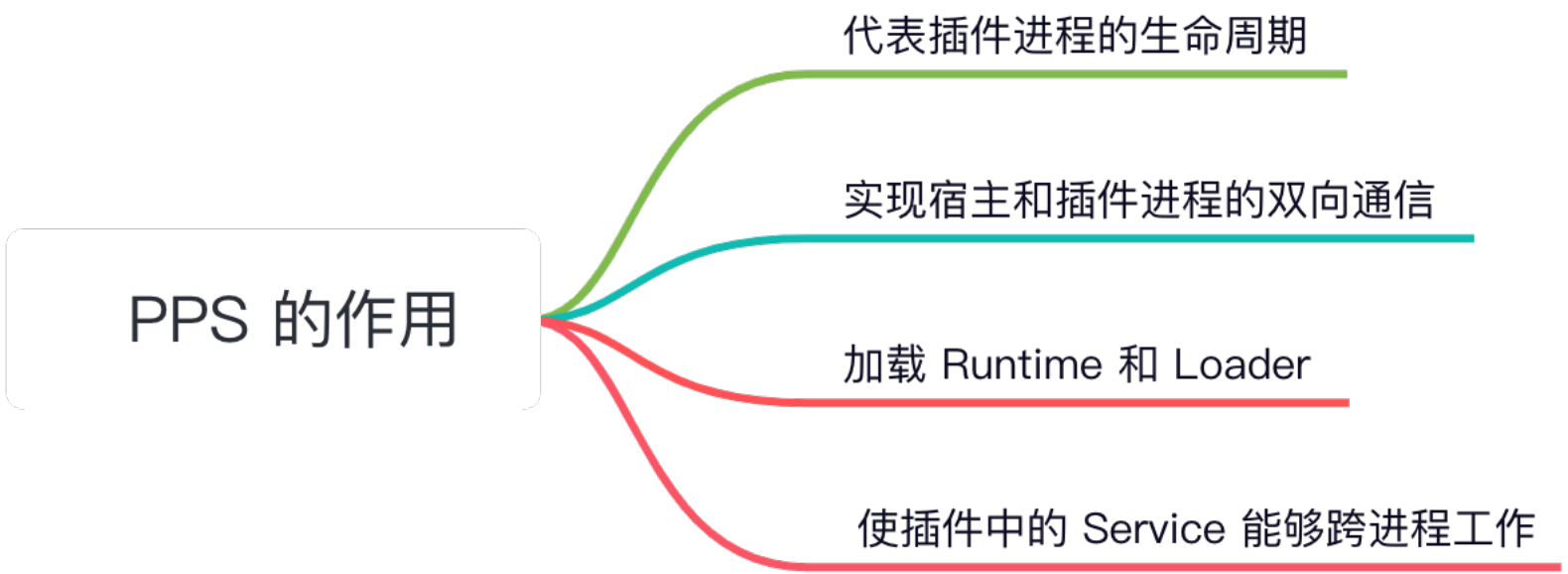

```kotlin
private fun Intent.toActivityContainerIntent(): Intent {
    val bundleForPluginLoader = Bundle()
    // 拿到插件 activity 对应的 pluginComponentInfo 对象;
    val pluginComponentInfo = pluginComponentInfoMap[component]!!
    bundleForPluginLoader.putParcelable(CM_ACTIVITY_INFO_KEY, pluginComponentInfo)
    // 继续执行;
    return toContainerIntent(bundleForPluginLoader)
}
```

```kotlin
    val packageName = packageNameMap[className]!!
    this.component = ComponentName(packageName, className)
    val containerComponent = componentMap[component]!! // 拿到了坑 activity;
    val businessName = pluginInfoMap[component]!!.businessName
    val partKey = pluginInfoMap[component]!!.partKey

    val pluginExtras: Bundle? = extras
    replaceExtras(null as Bundle?)

    // 新建了一个 intent
    val containerIntent = Intent(this)
    containerIntent.component = containerComponent // 设置坑 activity 为要启动的目标 activity;

    bundleForPluginLoader.putString(CM_CLASS_NAME_KEY, className)
    bundleForPluginLoader.putString(CM_PACKAGE_NAME_KEY, packageName)

    containerIntent.putExtra(CM_EXTRAS_BUNDLE_KEY, pluginExtras)
    containerIntent.putExtra(CM_BUSINESS_NAME_KEY, businessName)
    containerIntent.putExtra(CM_PART_KEY, partKey)
    containerIntent.putExtra(CM_LOADER_BUNDLE_KEY, bundleForPluginLoader) // 目标 activity;
    containerIntent.putExtra(LOADER_VERSION_KEY, BuildConfig.VERSION_NAME)
    containerIntent.putExtra(PROCESS_ID_KEY, DelegateProviderHolder.sCustomPid)
    return containerIntent
}
```


宿主子进程（插件）
5
PluginContainerActivity
hostActivityDelegate
DelegateProviderHolder.getDelegateProvider(this.getDelegateProviderKey());
获取的就是 SamplePluginLoader
DelegateProvider
ShadowPluginLoader
SamplePluginLoader.inject
ShadowActivityDelegate
getHostActivityDelegate
获取到插件的 classloader
ShadowActivity
onCreate
ShadowActivityDelegate.onCreate
ShadowActivity.onCreate

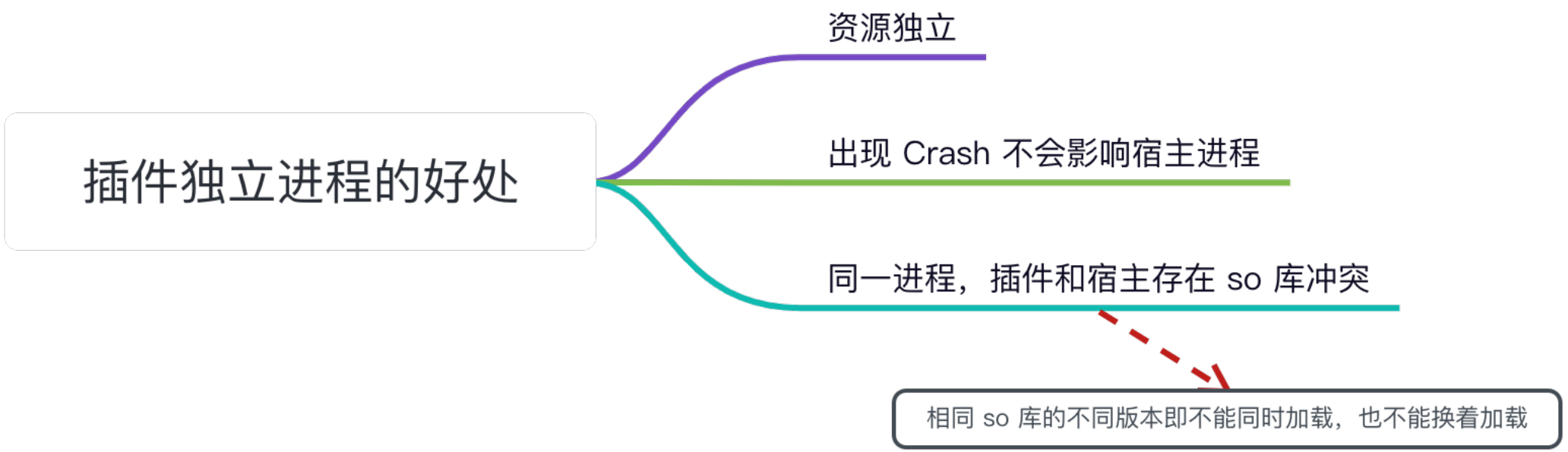
PPS 的作用

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

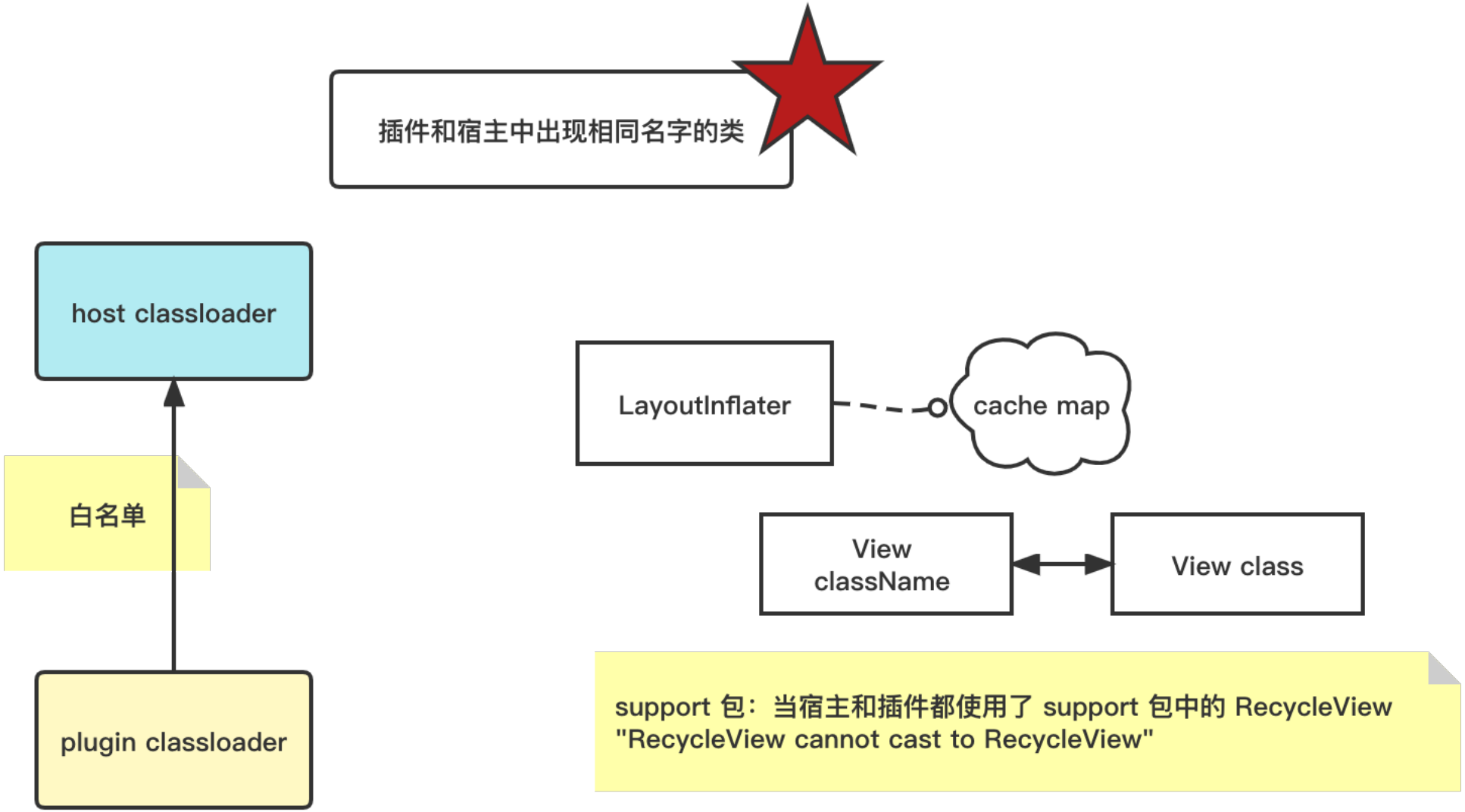
插件独立进程的好处

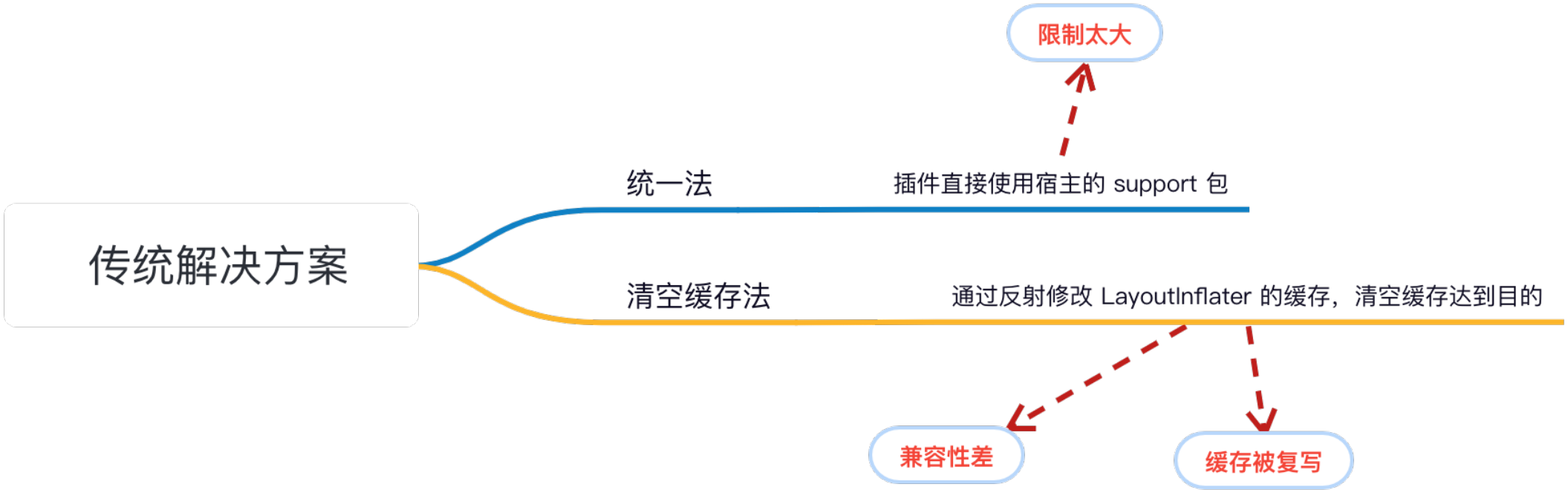
in Shadow

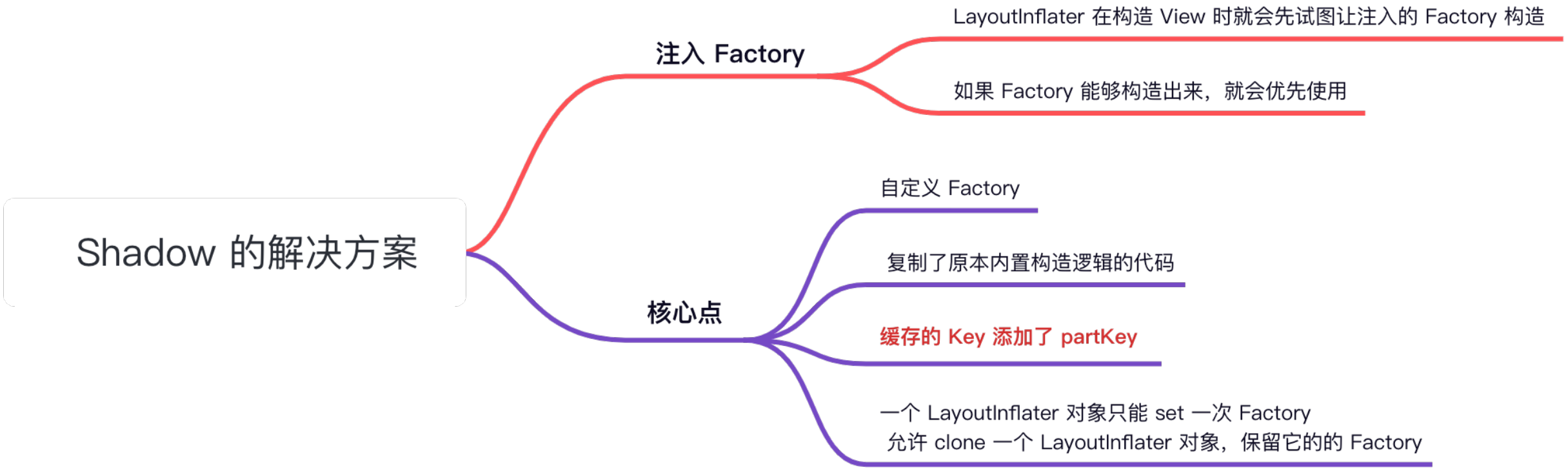

插件独立进程的好处
资源独立
出现 Crash 不会影响宿主进程
同一进程，插件和宿主存在 so 库冲突
相同 so 库的不同版本即不能同时加载，也不能换着加载

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
host classloader
白名单
plugin classloader
LayoutInflater
cache map
View
className
View class
support 包：当宿主和插件都使用了 support 包中的 RecycleView
"RecycleView cannot cast to RecycleView"


传统解决方案
统一法
插件直接使用宿主的 support 包
限制太大
清空缓存法
通过反射修改 LayoutInflater 的缓存，清空缓存达到目的
兼容性差
缓存被复写



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

```
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);
```

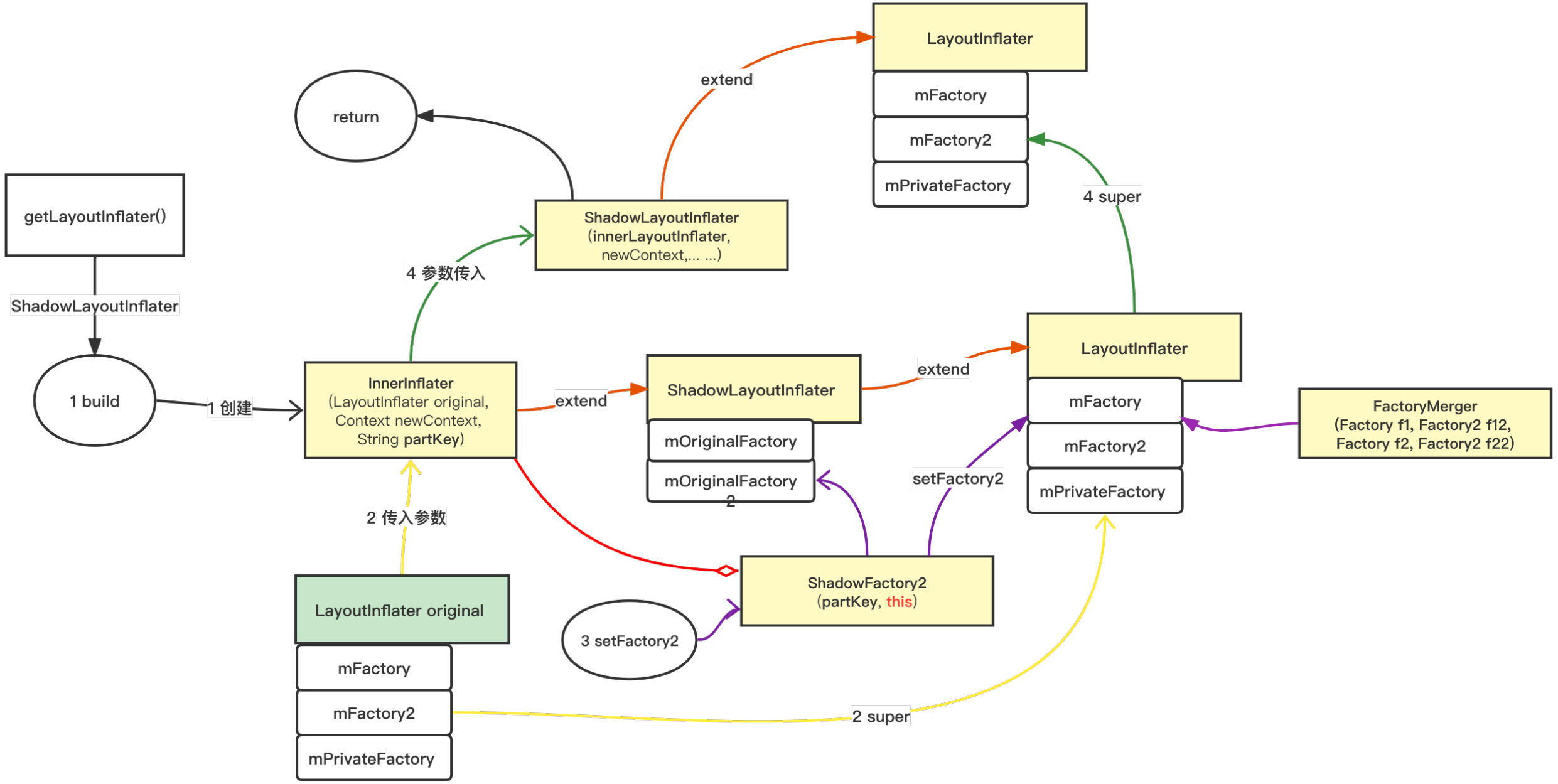
ShadowFactory2

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

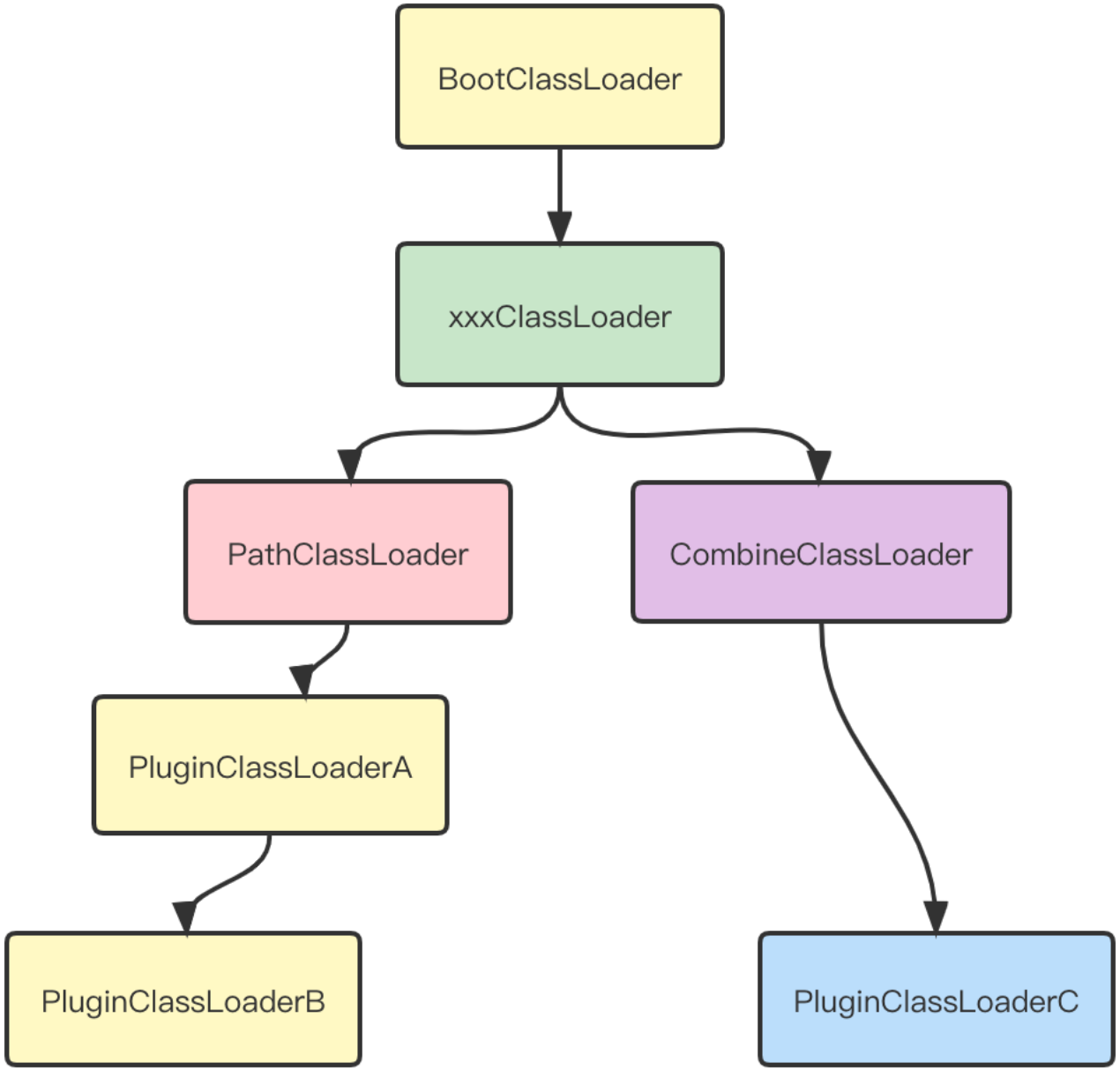
```java
public final View tryCreateView(@Nullable View parent, @NonNull String name,
    @NonNull Context context,
    @NonNull AttributeSet attrs) {
    if (name.equals(TAG_1995)) {
        // Let's party like it's 1995!
        return new BlinkLayout(context, attrs);
    }

    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;
}
```

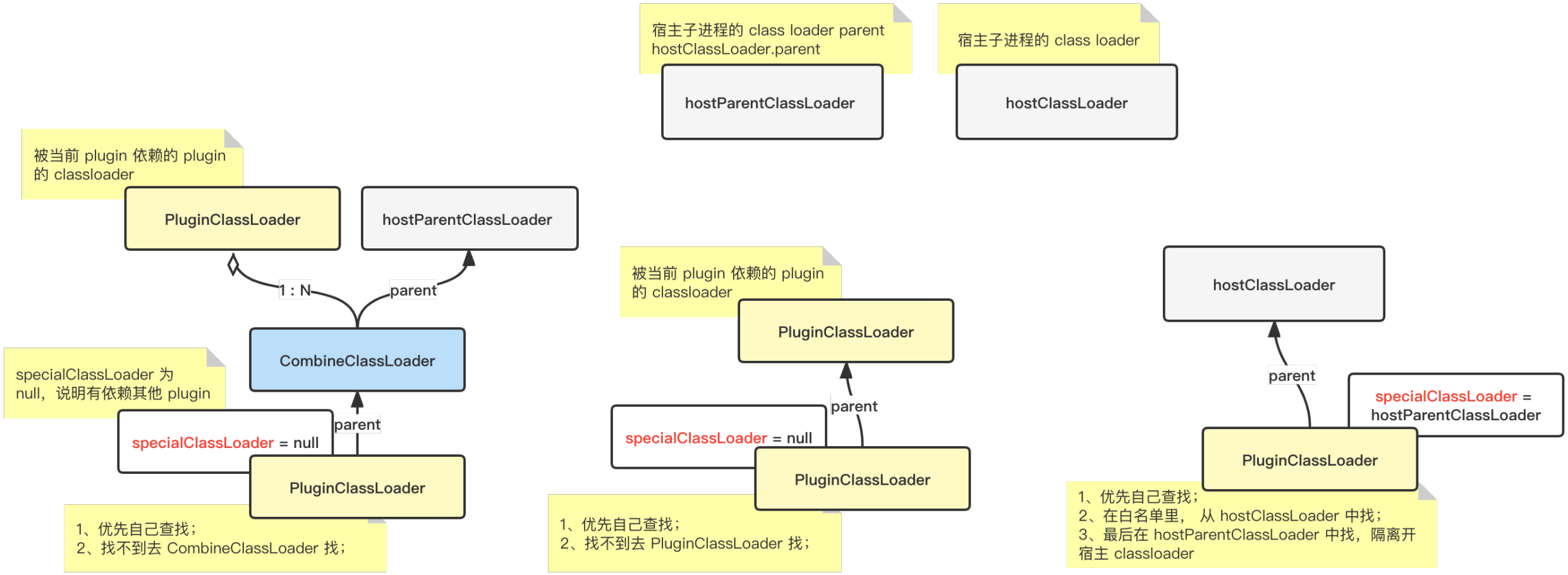
LayoutInflater


LayoutInflater
mFactory
mFactory2
mPrivateFactory
extend
return
getLayoutInflater()
ShadowLayoutInflater
(innerLayoutInflater,
newContext,... ...)
4 super
ShadowLayoutInflater
4 参数传入
1 build
1 创建
InnerInflater
(LayoutInflater original,
Context newContext,
String partKey)
extend
ShadowLayoutInflater
mOriginalFactory
mOriginalFactory
2
extend
LayoutInflater
mFactory
mFactory2
mPrivateFactory
FactoryMerger
(Factory f1, Factory2 f12,
Factory f2, Factory2 f22)
setFactory2
2 传入参数
ShadowFactory2
(partKey, this)
3 setFactory2
LayoutInflater original
mFactory
mFactory2
mPrivateFactory
2 super

插件 PluginClassLoader 浅析

in Shadow


BootClassLoader
xxxClassLoader
PathClassLoader
CombineClassLoader
PluginClassLoaderA
PluginClassLoaderB
PluginClassLoaderC


宿主子进程的 class loader parent
hostClassLoader.parent
hostParentClassLoader
宿主子进程的 class loader
hostClassLoader
被当前 plugin 依赖的 plugin 的 classloader
PluginClassLoader
hostParentClassLoader
1 : N
parent
CombineClassLoader
specialClassLoader 为 null，说明有依赖其他 plugin
specialClassLoader = null
parent
PluginClassLoader
1、优先自己查找；
2、找不到去 CombineClassLoader 找；
被当前 plugin 依赖的 plugin 的 classloader
PluginClassLoader
parent
specialClassLoader = null
PluginClassLoader
1、优先自己查找；
2、找不到去 PluginClassLoader 找；
hostClassLoader
parent
specialClassLoader = hostParentClassLoader
PluginClassLoader
1、优先自己查找；
2、在白名单里， 从 hostClassLoader 中找；
3、最后在 hostParentClassLoader 中找，隔离开宿主 classloader

插件包名和宿主包名

in Shadow

```kotlin
object CheckPackageNameBloc {
    @Throws(ParsePluginApkException::class)
    fun check(
        pluginManifest: PluginManifest,
        hostAppContext: Context
    ) {
        if (pluginManifest.applicationPackageName != hostAppContext.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:${hostAppContext.packageName} 插件:${
        }
    }
}
```

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) {
```

ShadowContext 代理

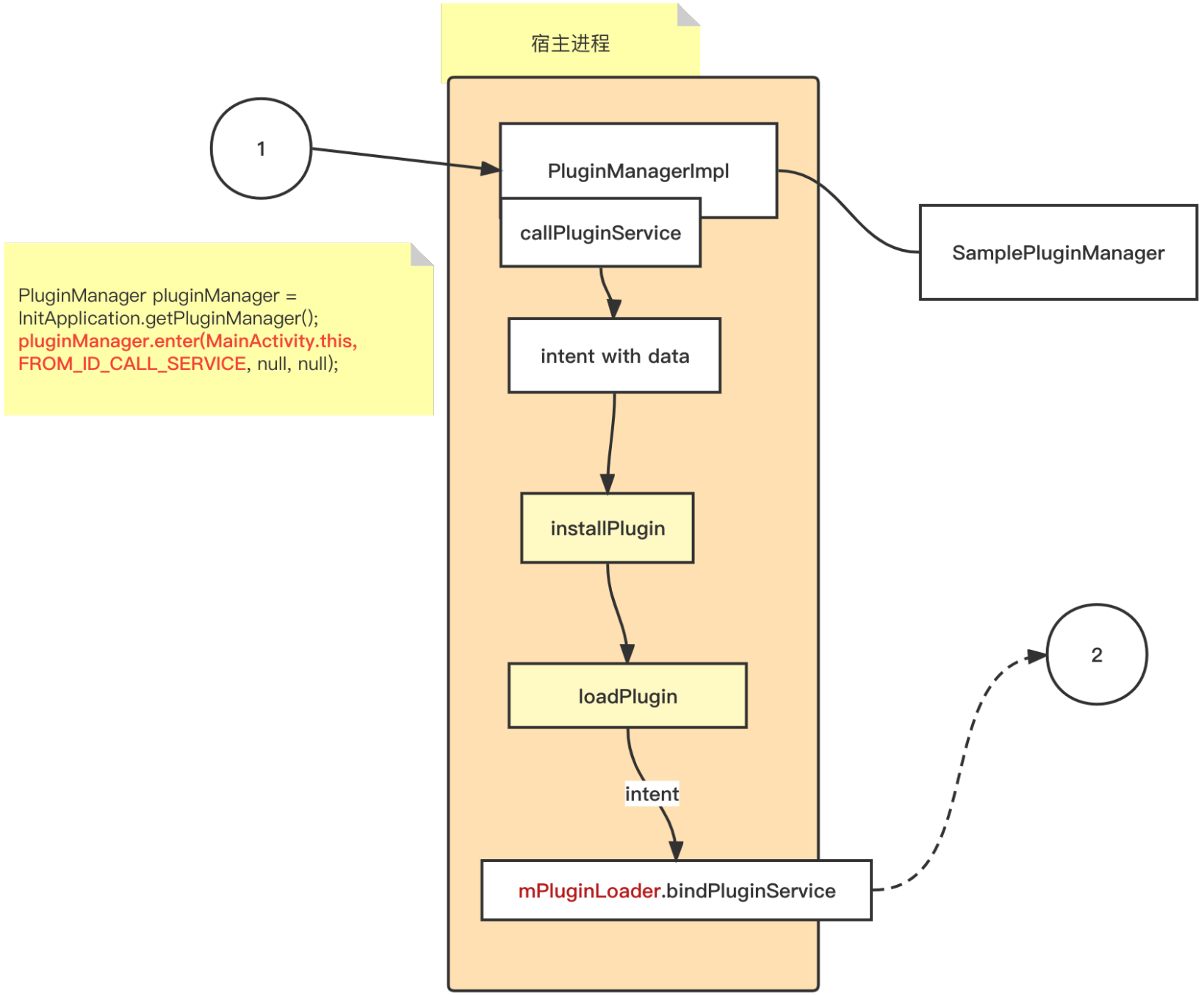
in Shadow

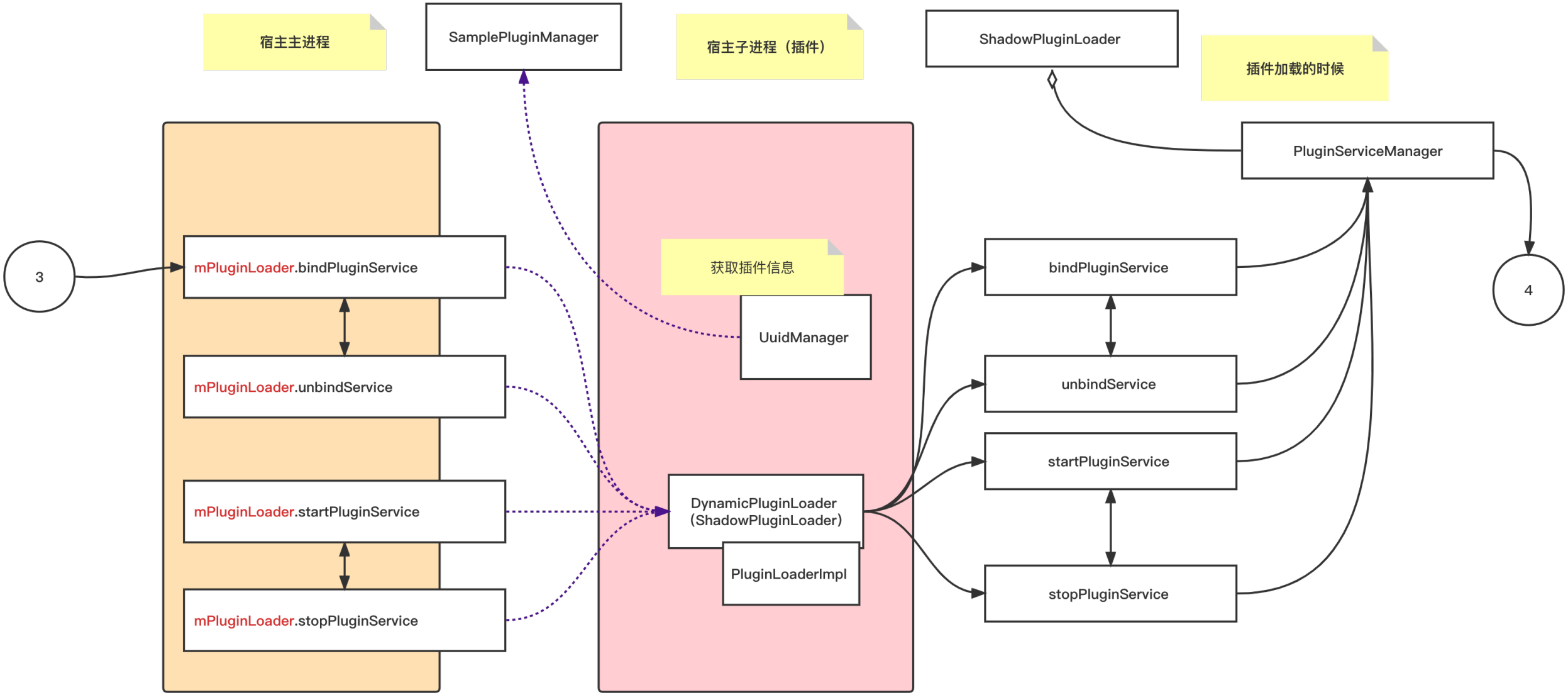
```java
public class ShadowContext extends SubDirContextThemeWrapper {
    PluginComponentLauncher mPluginComponentLauncher;
    ClassLoader mPluginClassLoader;
    ShadowApplication mShadowApplication;
    Resources mPluginResources;
    Resources mMixResources;
    LayoutInflater mLayoutInflater;
    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);
    }
```

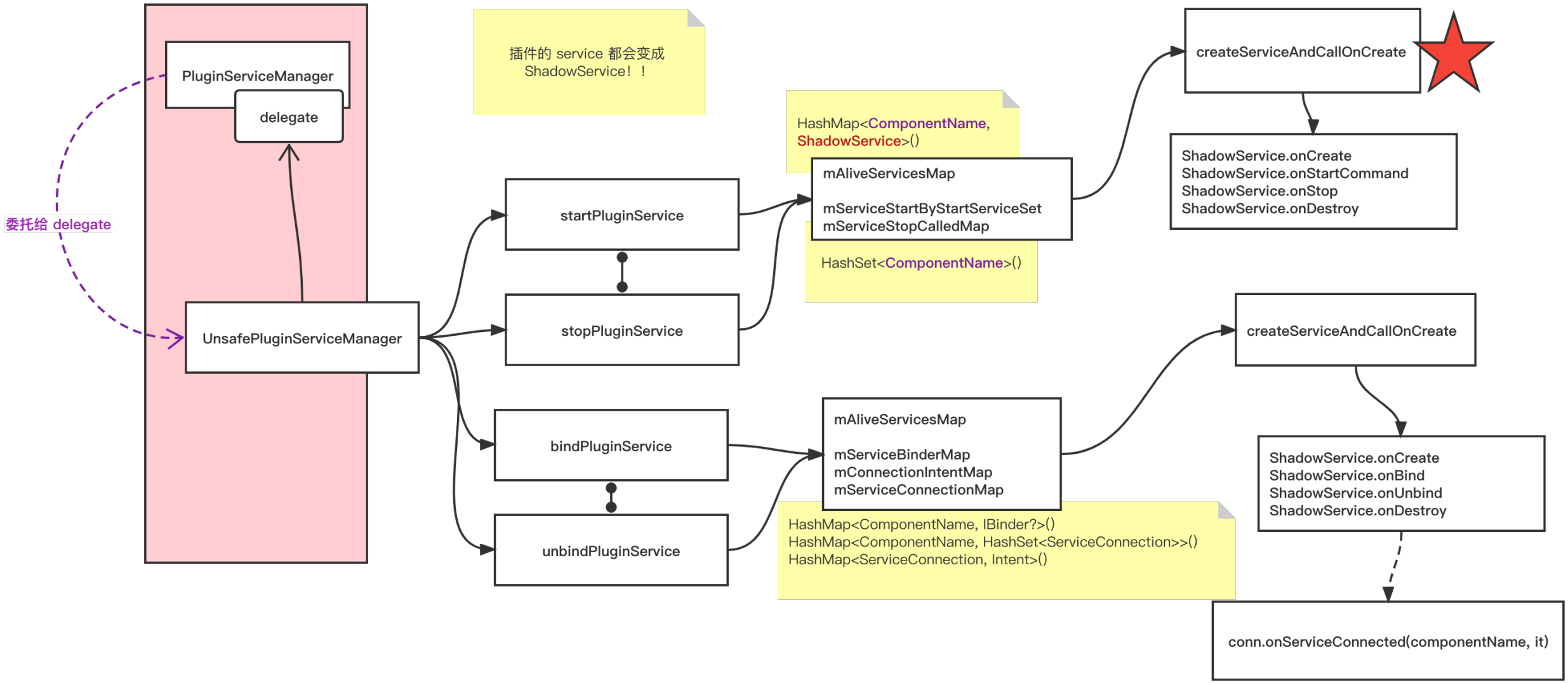
ShadowActivity、ShadowApplication 的父类都是 ShadowContext

宿主如何启动插件 Service

in Shadow

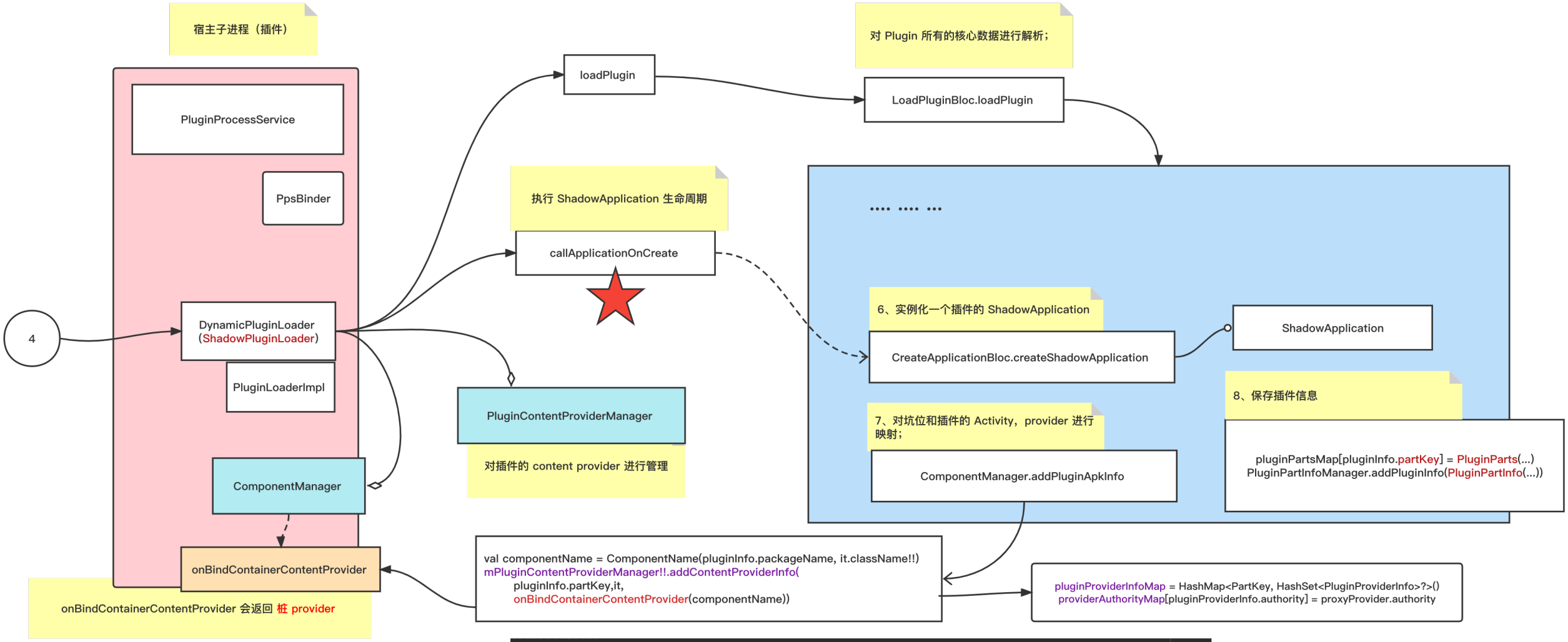

宿主进程
1
PluginManagerImpl
callPluginService
SamplePluginManager
PluginManager pluginManager =
InitApplication.getPluginManager();
pluginManager.enter(MainActivity.this,
FROM_ID_CALL_SERVICE, null, null);
intent with data
installPlugin
loadPlugin
intent
mPluginLoader.bindPluginService
2

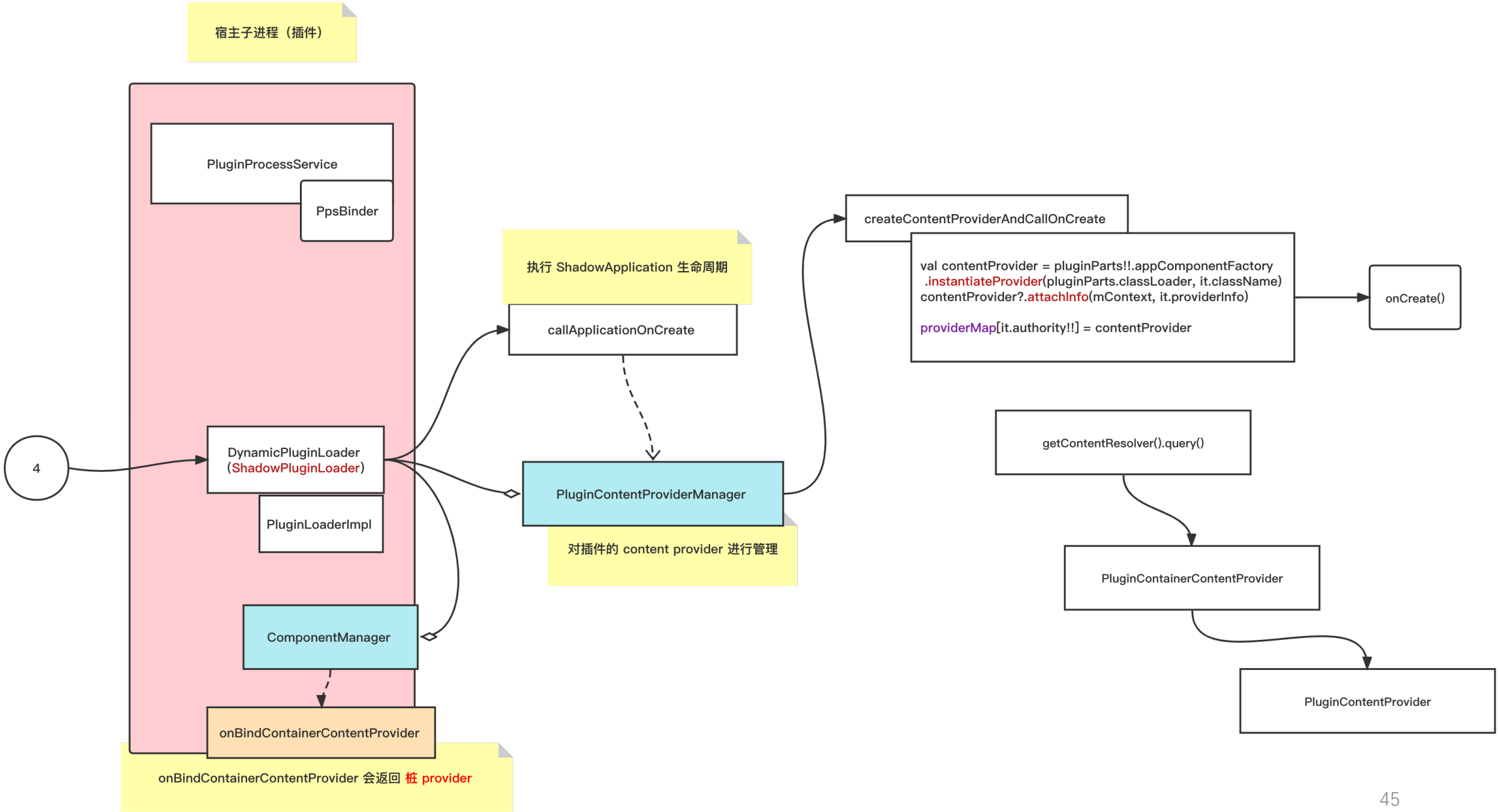

宿主主进程
SamplePluginManager
宿主子进程（插件）
ShadowPluginLoader
插件加载的时候
PluginServiceManager
3
mPluginLoader.bindPluginService
mPluginLoader.unbindService
mPluginLoader.startPluginService
mPluginLoader.stopPluginService
获取插件信息
UuidManager
DynamicPluginLoader
(ShadowPluginLoader)
PluginLoaderImpl
bindPluginService
unbindService
startPluginService
stopPluginService
4

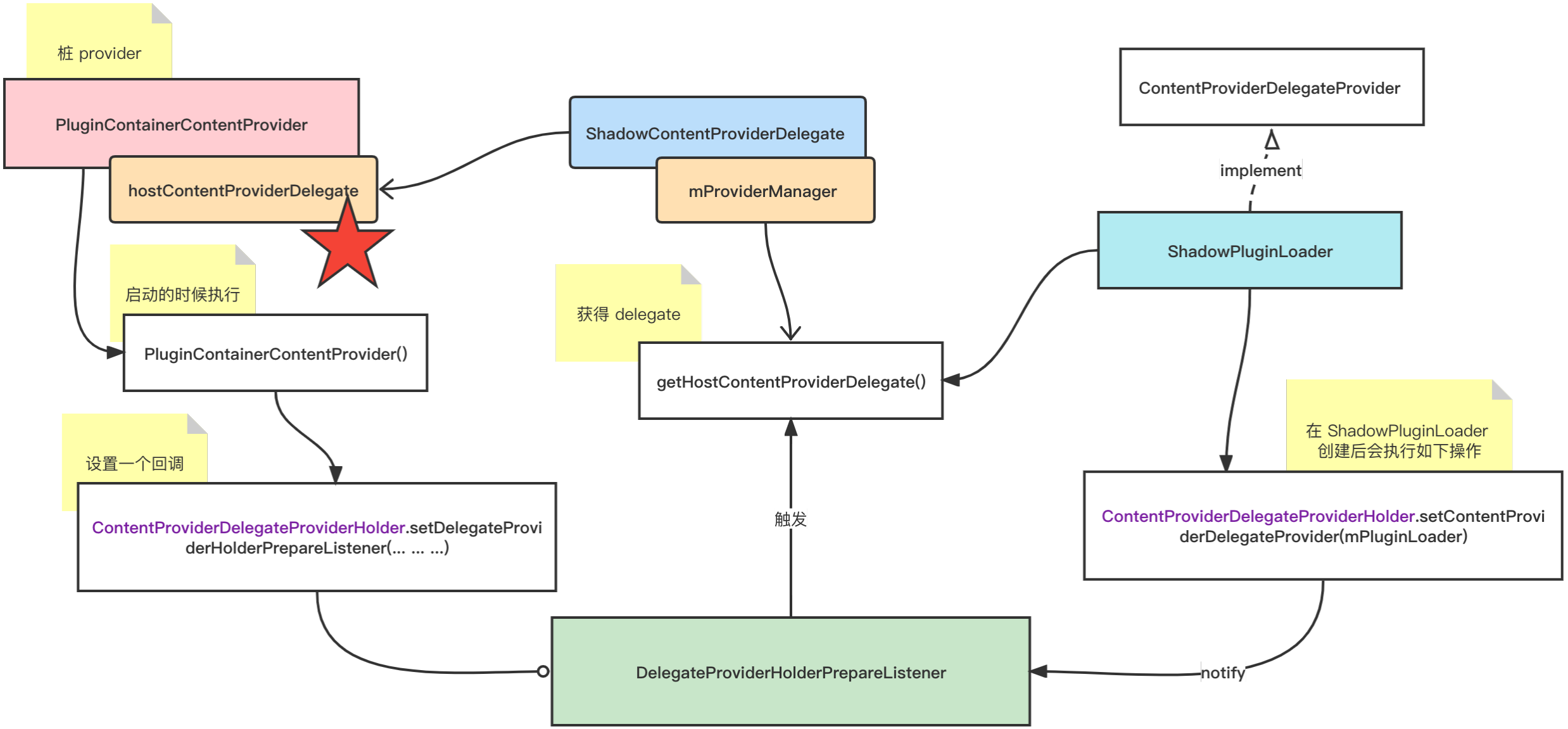

PluginServiceManager
delegate
委托给 delegate
UnsafePluginServiceManager
插件的 service 都会变成
ShadowService！！
startPluginService
stopPluginService
bindPluginService
unbindPluginService
HashMap<ComponentName,
ShadowService>()
mAliveServicesMap
mServiceStartByStartServiceSet
mServiceStopCalledMap
HashSet<ComponentName>()
mAliveServicesMap
mServiceBinderMap
mConnectionIntentMap
mServiceConnectionMap
HashMap<ComponentName, IBinder?>()
HashMap<ComponentName, HashSet<ServiceConnection>>()
HashMap<ServiceConnection, Intent>()
createServiceAndCallOnCreate
ShadowService.onCreate
ShadowService.onStartCommand
ShadowService.onStop
ShadowService.onDestroy
createServiceAndCallOnCreate
ShadowService.onCreate
ShadowService.onBind
ShadowService.onUnbind
ShadowService.onDestroy
conn.onServiceConnected(componentName, it)

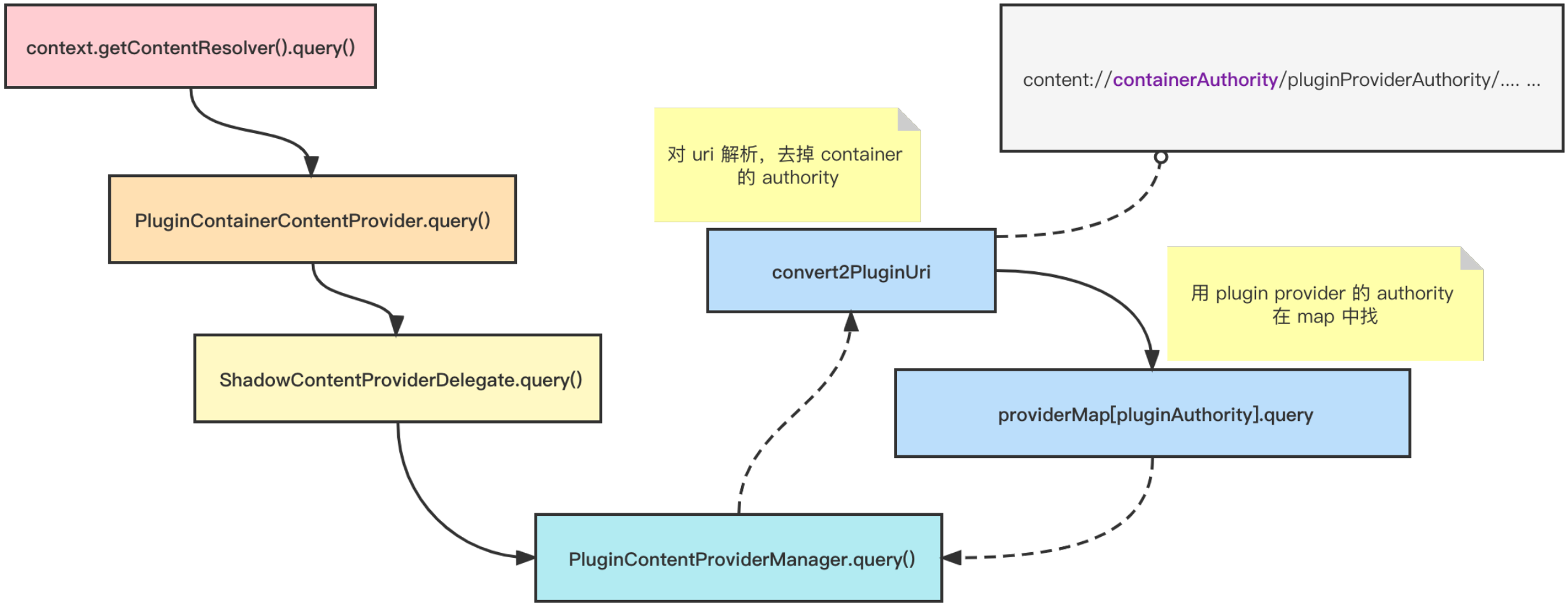
```
private fun createServiceAndCallOnCreate(intent: Intent): ShadowService {
    val service = newServiceInstance(intent)
    service.onCreate()
    return service
}


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

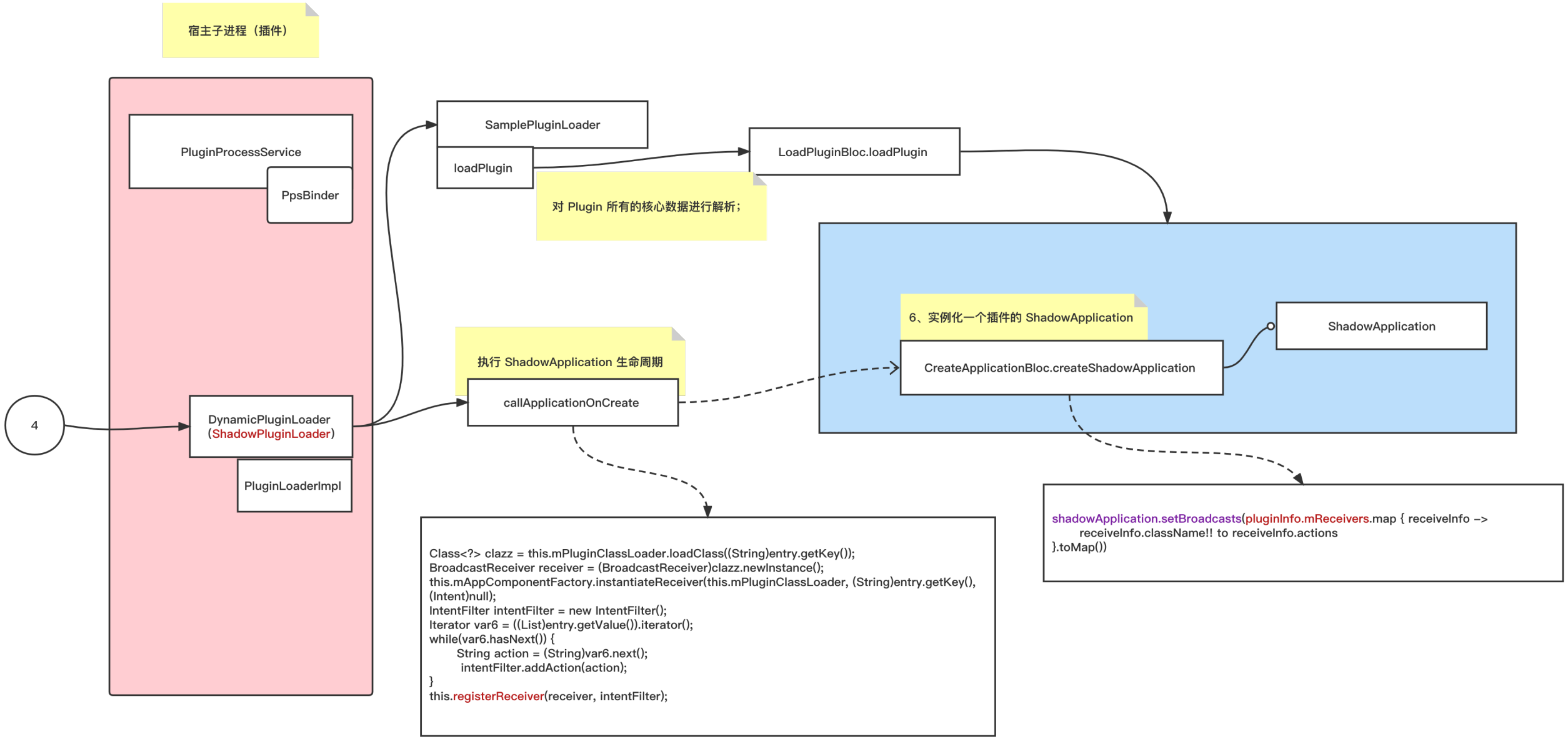

宿主子进程（插件）
PluginProcessService
PpsBinder
4
DynamicPluginLoader
(ShadowPluginLoader)
PluginLoaderImpl
ComponentManager
onBindContainerContentProvider
onBindContainerContentProvider 会返回 桩 provider
loadPlugin
对 Plugin 所有的核心数据进行解析；
LoadPluginBloc.loadPlugin
执行 ShadowApplication 生命周期
callApplicationOnCreate
PluginContentProviderManager
对插件的 content provider 进行管理
....
6、实例化一个插件的 ShadowApplication
CreateApplicationBloc.createShadowApplication
ShadowApplication
7、对坑位和插件的 Activity，provider 进行映射；
ComponentManager.addPluginApkInfo
8、保存插件信息
pluginPartsMap[pluginInfo.partKey] = PluginParts(...)
PluginPartInfoManager.addPluginInfo(PluginPartInfo(...))
val componentName = ComponentName(pluginInfo.packageName, it.className!!)
mPluginContentProviderManager!!.addContentProviderInfo(
pluginInfo.partKey,it,
onBindContainerContentProvider(componentName))
pluginProviderInfoMap = HashMap<PartKey, HashSet<PluginProviderInfo>?>()
providerAuthorityMap[pluginProviderInfo.authority] = proxyProvider.authority
<provider
android:authorities="com.tencent.shadow.contentprovider.authority.dynamic"
android:name="com.tencent.shadow.core.runtime.container.PluginContainerContentProvider" />
<!--container 注册 end -->


宿主子进程（插件）
PluginProcessService
PpsBinder
4
DynamicPluginLoader
(ShadowPluginLoader)
PluginLoaderImpl
ComponentManager
onBindContainerContentProvider
onBindContainerContentProvider 会返回 桩 provider
执行 ShadowApplication 生命周期
callApplicationOnCreate
PluginContentProviderManager
对插件的 content provider 进行管理
createContentProviderAndCallOnCreate
val contentProvider = pluginParts!!.appComponentFactory
.instantiateProvider(pluginParts.classLoader, it.className)
contentProvider?.attachInfo(mContext, it.providerInfo)
providerMap[it.authority!!] = contentProvider
onCreate()
getContentResolver().query()
PluginContainerContentProvider
PluginContentProvider


桩 provider
PluginContainerContentProvider
hostContentProviderDelegate
ShadowContentProviderDelegate
mProviderManager
ContentProviderDelegateProvider
implement
ShadowPluginLoader
启动的时候执行
PluginContainerContentProvider()
获得 delegate
getHostContentProviderDelegate()
在 ShadowPluginLoader
创建后会执行如下操作
设置一个回调
ContentProviderDelegateProviderHolder.setDelegateProviderHolderPrepareListener(... ... ...)
触发
ContentProviderDelegateProviderHolder.setContentProviderDelegateProvider(mPluginLoader)
DelegateProviderHolderPrepareListener
notify


context.getContentResolver().query()
PluginContainerContentProvider.query()
ShadowContentProviderDelegate.query()
PluginContentProviderManager.query()
对 uri 解析，去掉 container 的 authority
convert2PluginUri
content://containerAuthority/pluginProviderAuthority/.... ...
用 plugin provider 的 authority 在 map 中找
providerMap[pluginAuthority].query

宿主如何启动和访问插件 receiver in Shadow


宿主子进程（插件）
PluginProcessService
PpsBinder
SamplePluginLoader
loadPlugin
LoadPluginBloc.loadPlugin
对 Plugin 所有的核心数据进行解析；
4
DynamicPluginLoader
(ShadowPluginLoader)
PluginLoaderImpl
执行 ShadowApplication 生命周期
callApplicationOnCreate
6、实例化一个插件的 ShadowApplication
CreateApplicationBloc.createShadowApplication
ShadowApplication
Class<?> clazz = this.mPluginClassLoader.loadClass((String)entry.getKey());
BroadcastReceiver receiver = (BroadcastReceiver)clazz.newInstance();
this.mAppComponentFactory.instantiateReceiver(this.mPluginClassLoader, (String)entry.getKey(),
(Intent)null);
IntentFilter intentFilter = new IntentFilter();
Iterator var6 = ((List)entry.getValue()).iterator();
while(var6.hasNext()) {
String action = (String)var6.next();
intentFilter.addAction(action);
}
this.registerReceiver(receiver, intentFilter);
shadowApplication.setBroadcasts(pluginInfo.mReceivers.map { receiveInfo ->
receiveInfo.className!! to receiveInfo.actions
}.toMap())

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"
    }
}
```

snappify.io

```
// 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)

}
```

snappify.io

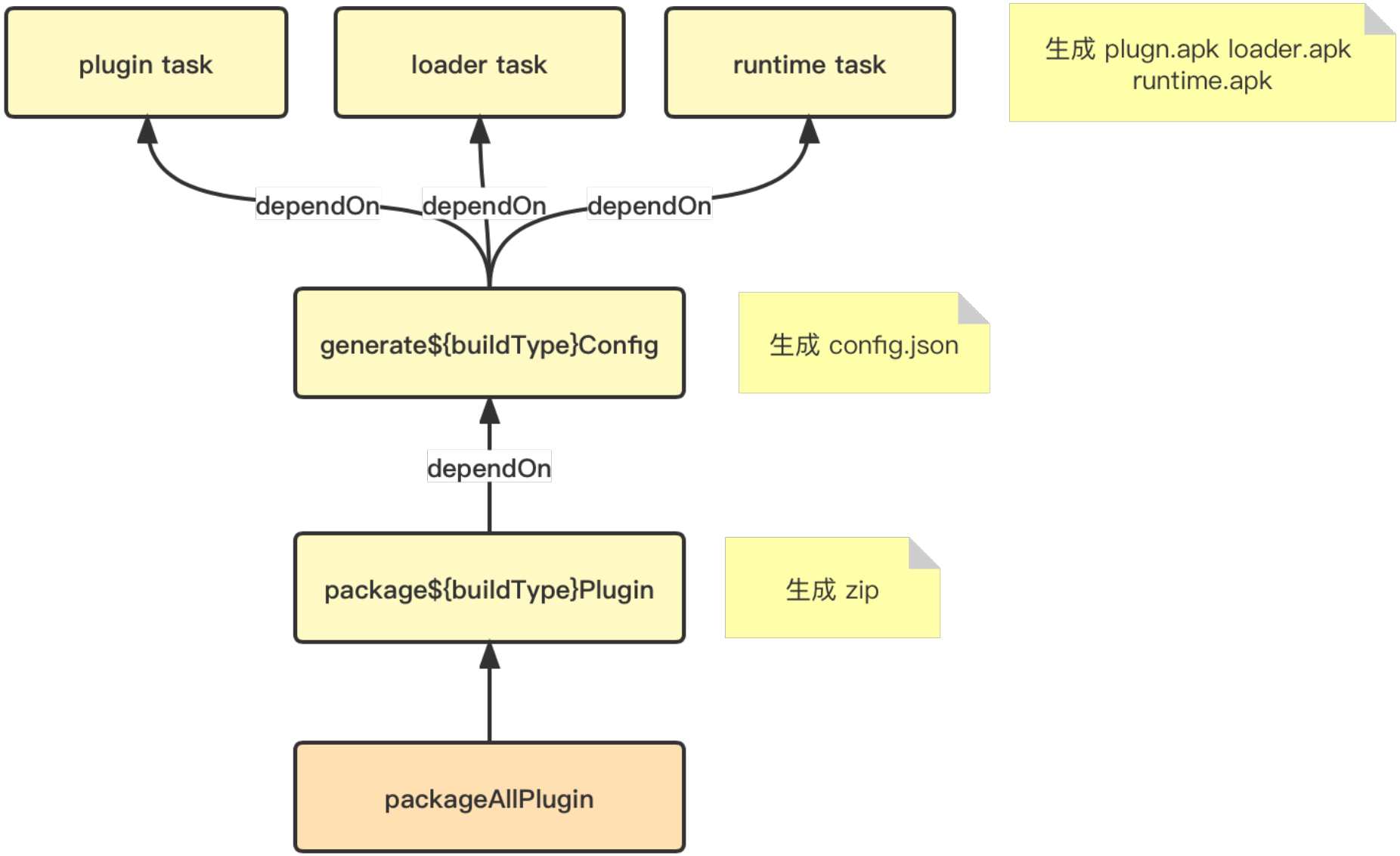
```
// ShadowPlugin
override fun apply(project: Project) {
    project.afterEvaluate {
        initAndroidClassPoolBuilder(baseExtension, project)
        // 1、创建 plugin 的编译 task;;
        createPackagePluginTasks(project)
        // 2、遍历每个变体;;
        onEachPluginVariant(project) { pluginVariant →
            // 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
            checkAaptPackageIdConfig(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)
        }
    }

}
```

snappify.io

Gradle Plugin 简析-Task

in Shadow

```kotlin
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)
    }
}
```

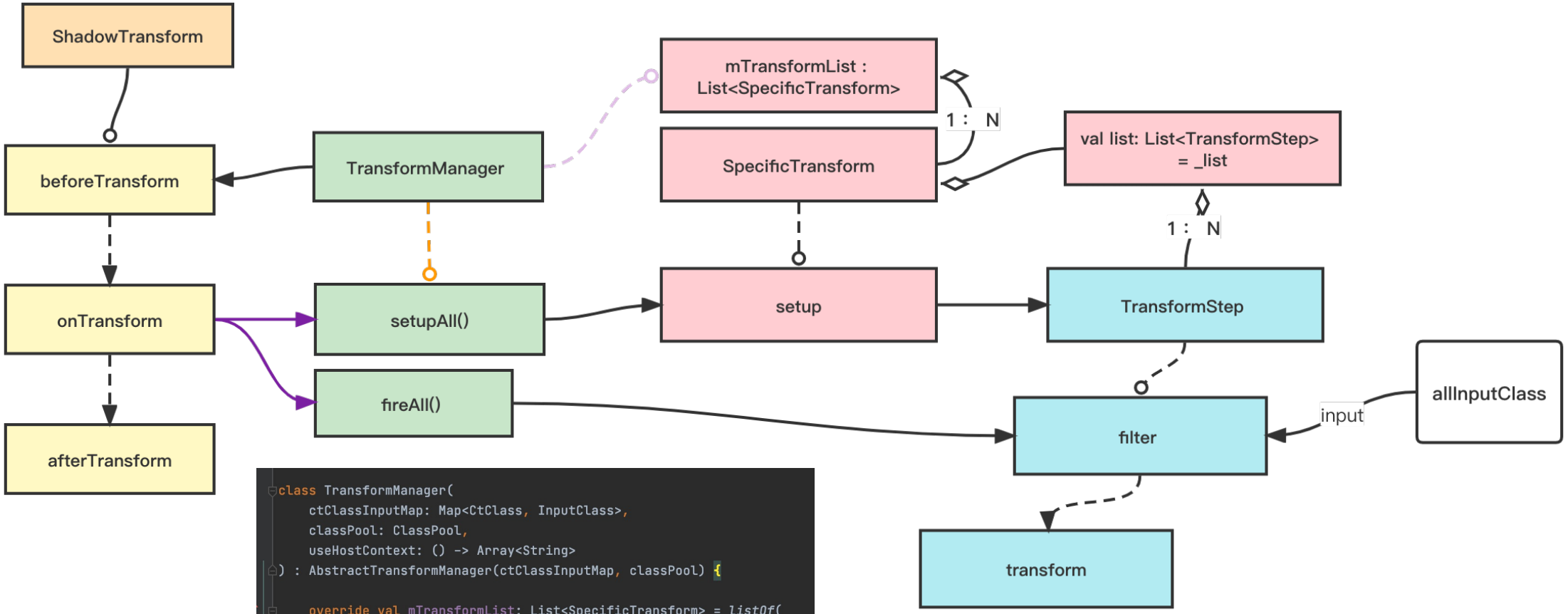

plugin task
loader task
runtime task
生成 plugn.apk loader.apk runtime.apk
dependOn
dependOn
dependOn
generate${buildType}Config
生成 config.json
dependOn
package${buildType}Plugin
生成 zip
packageAllPlugin

```kotlin
/**
 * 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)
}
```



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

Gradle Plugin 简析- Transformer in Shadow




```
class TransformManager(
    ctClassInputMap: Map<CtClass, InputClass>,
    classPool: ClassPool,
    useHostContext: () -> Array<String>
) : AbstractTransformManager(ctClassInputMap, classPool) {

    override val mTransformList: List<SpecificTransform> = listOf(
        ApplicationTransform(),
        ActivityTransform(),
        ServiceTransform(),
        IntentServiceTransform(),
        InstrumentationTransform(),
        FragmentSupportTransform(),
        DialogSupportTransform(),
        WebViewTransform(),
        ContentProviderTransform(),
        PackageManagerTransform(),
        PackageItemInfoTransform(),
        AppComponentFactoryTransform(),
        LayoutInflaterTransform(),
        KeepHostContextTransform(useHostContext())
    )
}
```

```
open class SimpleRenameTransform(private val fromToMap: Map<String, String>) : SpecificTransform() {
    final override fun setup(allInputClass: Set<CtClass>) {
        newStep(object : TransformStep {
            override fun filter(allInputClass: Set<CtClass>) = allInputClass

            override fun transform(ctClass: CtClass) {
                fromToMap.forEach { it: Map.Entry<String, String>
                    ReplaceClassName.replaceClassName(ctClass, it.key, it.value)
                }
            }
        })
    }
}
```

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")
)
```

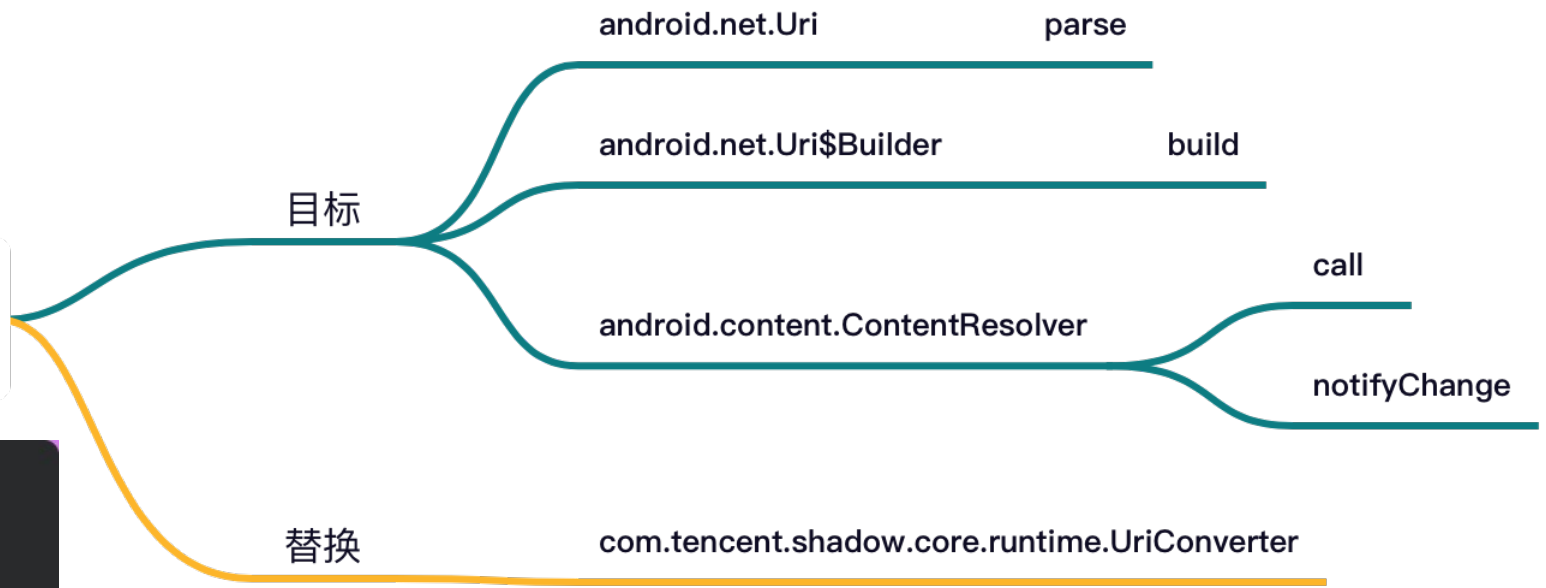
```
class ActivityTransform : SimpleRenameTransform(
    mapOf(
        "android.app.Activity"
                to "com.tencent.shadow.core.runtime.ShadowActivity",
        "android.app.NativeActivity"
                to "com.tencent.shadow.core.runtime.ShadowNativeActivity"
    )
)
```

```
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



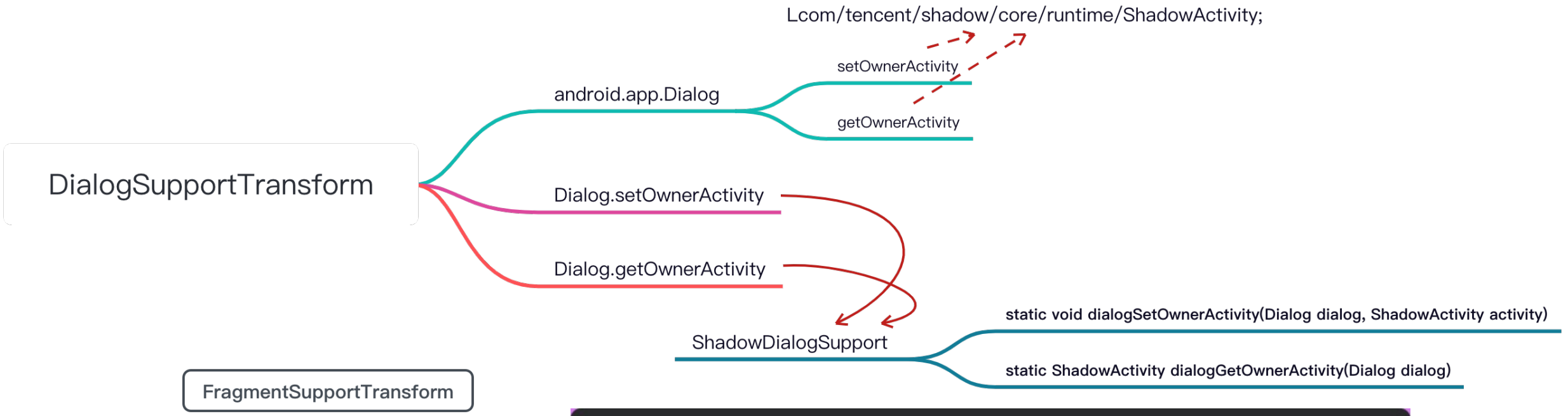

```java
// call 和 parseCall
public static Bundle call(ContentResolver resolver, Uri uri,
                  String method, String arg, Bundle extras) {
    if (extras == null) {
        extras = new Bundle();
    }
    Uri containerUri = UriConverter.parseCall(uri.toString(), extras);
    return resolver.call(containerUri,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

snappify.io

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

    private val mPluginContentProviderManager: PluginContentProviderManager =
        PluginContentProviderManager()

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

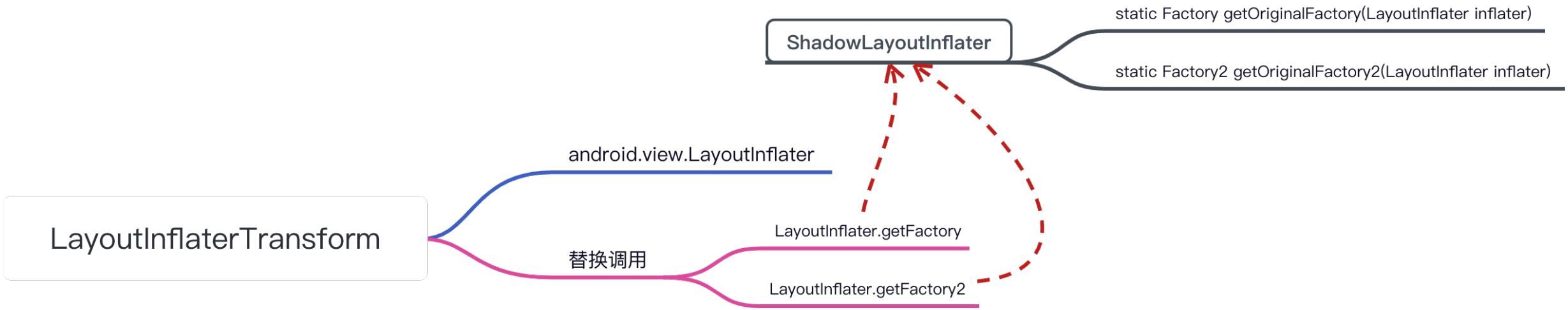
snappify.io

```kotlin
// 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
}
```




```
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;
        }
    }
}
```

snappify.io




```java
@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);
}
```

```java
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

com.tencent.shadow.core.
runtime.PackageManagerInvokeRedirect

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

static PluginPackageManager getPluginPackageManager(ClassLoader classLoaderOfInvokeCode)

android.content.pm.PackageManager

替换方法

"getApplicationInfo",
"getActivityInfo",
"getServiceInfo",
"getPackageInfo",
"resolveContentProvider",
"queryContentProviders",
"resolveActivity",
"resolveService",

```
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);
}
```

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

snappify.io

```
@SuppressLint( ...value: "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