TLC 2019 大前端信息流

腾讯LIVE开发者大会

TENCENT LIVE DEVELOPER CONFERENCE

主题：Tencent Shadow

如何在成熟的Android插件技术领域继续创新?

郭琨　前腾讯高级工程师

个人介绍

郭琨

- 前腾讯高级工程师
- 平安科技安卓技术专家
- Shadow插件框架作者之一

内容大纲

01 插件框架是做什么的?

02 为什么说插件框架是个成熟的领域了?

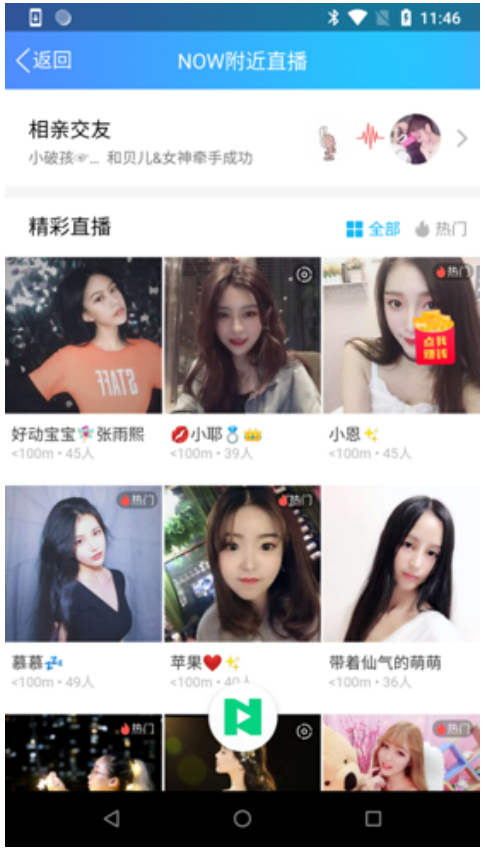
03 Shadow在这个领域做出了哪些创新?

04 Shadow核心技术原理

插件框架是什么？

没用插件框架时

插件框架是做什么的?



用了插件框架时

插件框架是做什么的?

- 减少安装包体积
- 动态发布新功能
- 动态修复线上bug
- 自解耦业务

……

插件框架的其他用途

为什么说插件框架是个成熟的领域?

框架名称	开发者	推出时间
AndroidDynamicLoader	大众点评	2012年6月
Atlas	阿里	2013年中
DL(Dynamic_Load_apk)	任玉刚	2014年1月
OpenAltas(ACCD)	bunnyblue	2015年8月
DroidPlugin	360	2015年8月
DynamicAPK	携程	2015年11月
Small	林广亮	2016年6月
VirtualAPK	滴滴	2017年6月
RePlugin	360	2017年7月
Phantom	满帮集团	2018年10月
Tencent Shadow	腾讯	2019年6月

插件框架发展时间线

为什么说插件框架是个成熟的领域?

360发布RePlugin

Shadow做出了哪些创新?

- 零反射、无Hack实现插件框架
- 全动态插件框架

Shadow做出了哪些创新?

专访DroidPlugin作者张勇：安卓黑科技是怎样炼成的 - InfoQ

https://www.infoq.cn/article/.../droidplugin-zhangyong-interview ▼ Translate this page

Sep 29, 2015 - 一时间，它被誉为安卓黑科技，引起行业内的关注。 ... 一个Android 开源项目 DroidPlugin，这是一个实现动态加载的Android 插件框架，可以免安装、 ...

详解Android插件框架(一)--初见| 蒲文辉

www.puwenhui.com/2018/.../详解Android插件框架%20(一)--初见... ▼ Translate this page

Jun 25, 2018 - 用古人的话说这就叫“ 黑科技”。 本系列的文章我们主要来讲插件化框架，也就是 VirtualApk 和Replugin这类框架主要实现的思想和技术，先打个底。

插件框架——公认的黑科技

Shadow做出了哪些创新?

https://github.com/Qihoo360/RePlugin/issues/516

Android 9.0将禁止非SDK接口的使用，是否会影响到Replugin #516

原 Android P阻止调用非sdk api后，Atlas该何去何从

置顶 2018年05月27日 12:59:56 smallnewQaQ 阅读数：1841

Android 9.0 限制私有API调用

Shadow做出了哪些创新?

Phantom — 唯一零 Hook 稳定占坑类 Android 热更新插件化方案

Phantom 是满帮集团开源的一套稳定、灵活、兼容性好的 Android 插件化方案。

Phantom 的优势

- 兼容性好：零 Hook，兼容 Android 4.0 ~ Android Q beta 4(final APIs)

Github上第一个号称零Hook的插件框架

Shadow做出了哪些创新?

W/.phantom.sampl: Accessing hidden method Landroid/view/View;->computeFitSystemWindows(Landroid/graphics/Rect;Landroid/graphics/Rect;)Z (light greylist, reflection)

W/System.err: StrictMode VmPolicy violation with POLICY_DEATH; shutting down.

I/Process: Sending signal. PID: 15302 SIG: 9

Application terminated.

严格模式下运行Phantom的Sample

Shadow做出了哪些创新?


```
ON_CREATE = ReflectUtils.getMethod(activityClass, fieldName: "onCreate", bundleClass);
ON_POST_CREATE = ReflectUtils.getMethod(activityClass, fieldName: "onPostCreate", bundleC
ON_START = ReflectUtils.getMethod(activityClass, fieldName: "onStart");
ON_RESUME = ReflectUtils.getMethod(activityClass, fieldName: "onResume");
ON_POST_RESUME = ReflectUtils.getMethod(activityClass, fieldName: "onPostResume");
ON_PAUSE = ReflectUtils.getMethod(activityClass, fieldName: "onPause");
ON_STOP = ReflectUtils.getMethod(activityClass, fieldName: "onStop");
ON_DESTROY = ReflectUtils.getMethod(activityClass, fieldName: "onDestroy");
ON_SAVE_INSTANCE_STATE = ReflectUtils.getMethod(activityClass, fieldName: "onSaveInstance
ON_RESTORE_INSTANCE_STATE = ReflectUtils.getMethod(activityClass, fieldName: "onRestoreIn
ON_ATTACHED_TO_WINDOW = ReflectUtils.getMethod(activityClass, fieldName: "onAttachedToWin
ON_DETACHED_FROM_WINDOW = ReflectUtils.getMethod(activityClass, fieldName: "onDetachedFro
ON_KEY_DOWN = ReflectUtils.getMethod(activityClass, fieldName: "onKeyDown", integerClass,
ON_KEY_UP = ReflectUtils.getMethod(activityClass, fieldName: "onKeyUp", integerClass, key
ON_BACK_PRESSED = ReflectUtils.getMethod(activityClass, fieldName: "onBackPressed");
ON_ACTIVITY_RESULT = ReflectUtils.getMethod(activityClass, fieldName: "onActivityResult",
        integerClass, integerClass, intentClass);
ON_NEW_INTENT = ReflectUtils.getMethod(activityClass, fieldName: "onNewIntent", intentCla

M_FRAGMENTS = ReflectUtils.getField(FragmentActivity.class, fieldName: "mFragments");
M_HOST = ReflectUtils.getField(FragmentController.class, fieldName: "mHost");
M_ACTIVITY = ReflectUtils.getField(fragmentHostCallbackClass, fieldName: "mActivity");
M_CONTEXT = ReflectUtils.getField(fragmentHostCallbackClass, fieldName: "mContext");
```

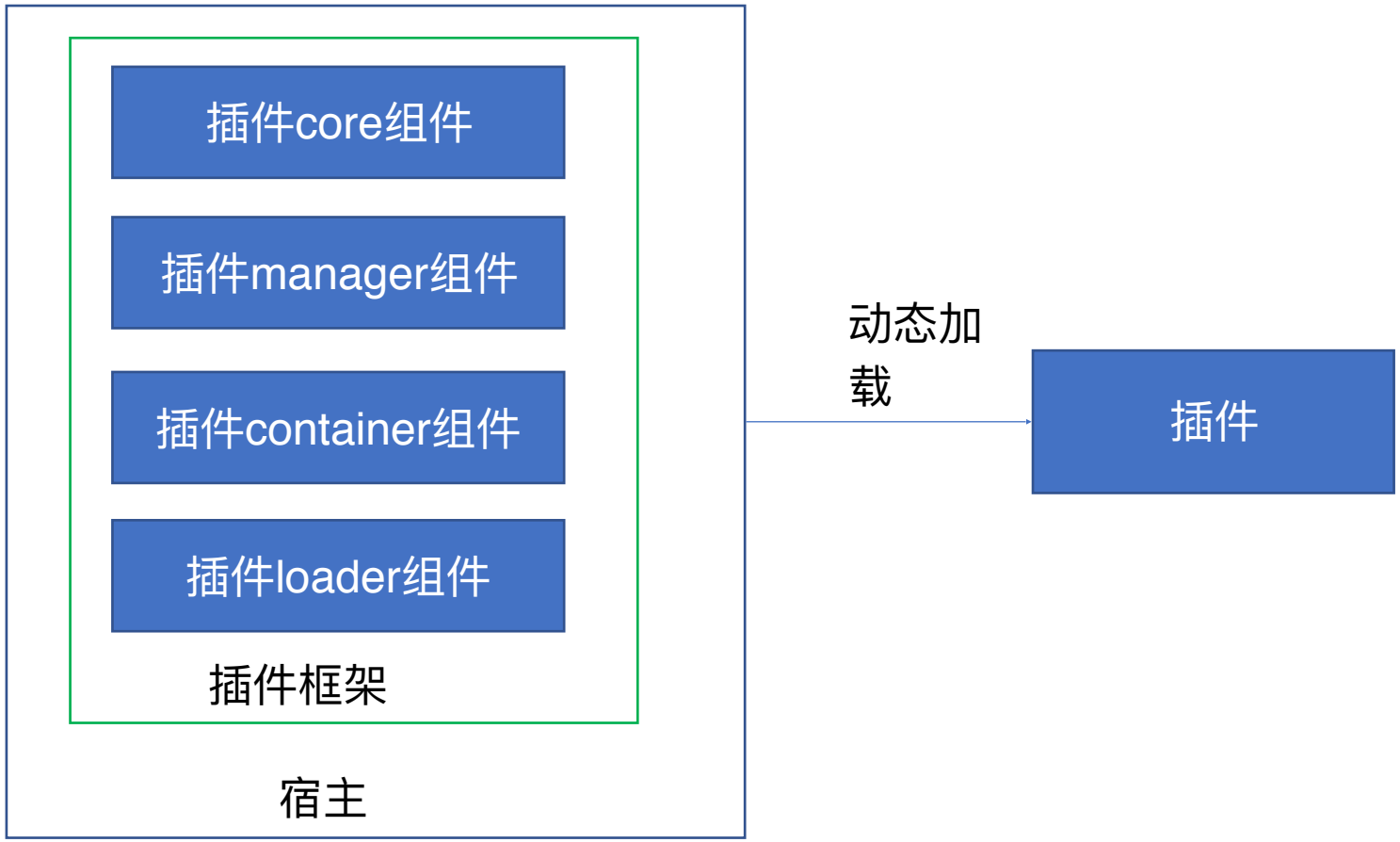
反射调用插件Activity的生命周期方法

```
/**
 * 将宿主Activity的一些变量值赋值给插件Activity，使插件Activity与宿主占位
 */
private void attachStatus(PluginInterceptActivity pluginActivity)
    for (Field field : ACTIVITY_FIELDS) {
        int modifiers = field.getModifiers();
        if (Modifier.isStatic(modifiers)
                || Modifier.isFinal(modifiers)
                || Modifier.isVolatile(modifiers)
                || Modifier.isTransient(modifiers)) {
            continue;
        }

        field.setAccessible(true);
        Object fieldsValue = field.get(mBaseContext);
        field.set(pluginActivity, fieldsValue);
    }
```

反射复制Activity的私有域

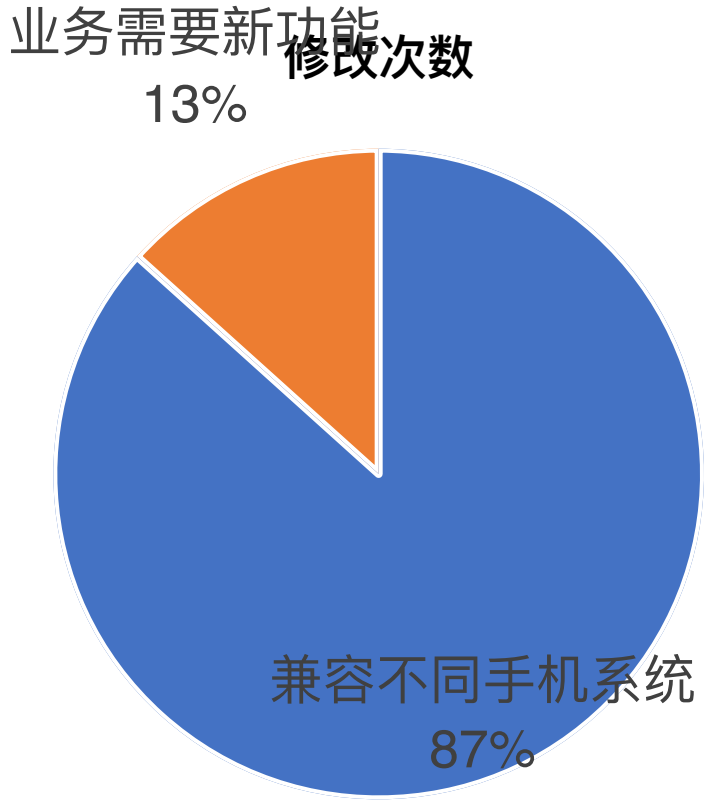
零Hook不等于零反射

Shadow做出了哪些创新?



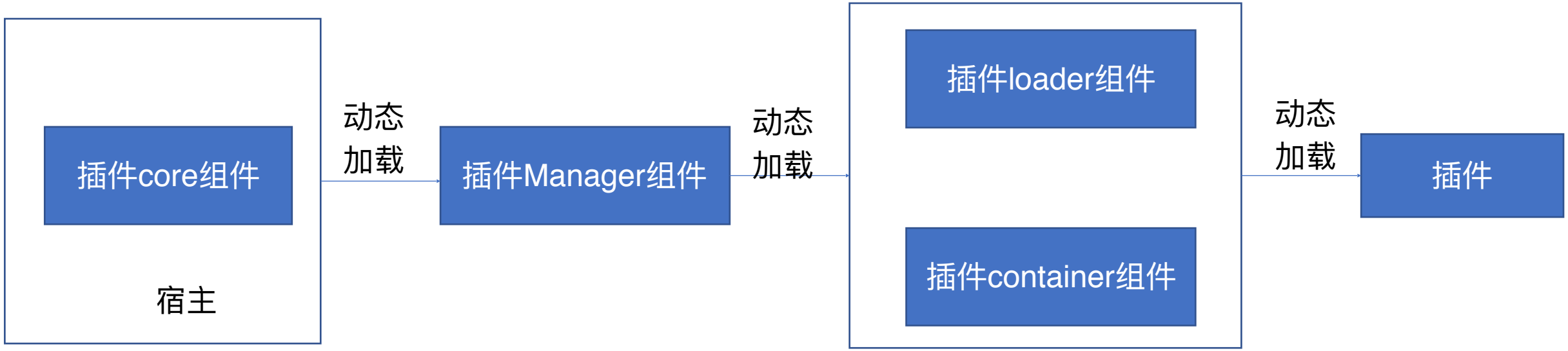
通用的插件框架架构

Shadow做出了哪些创新?



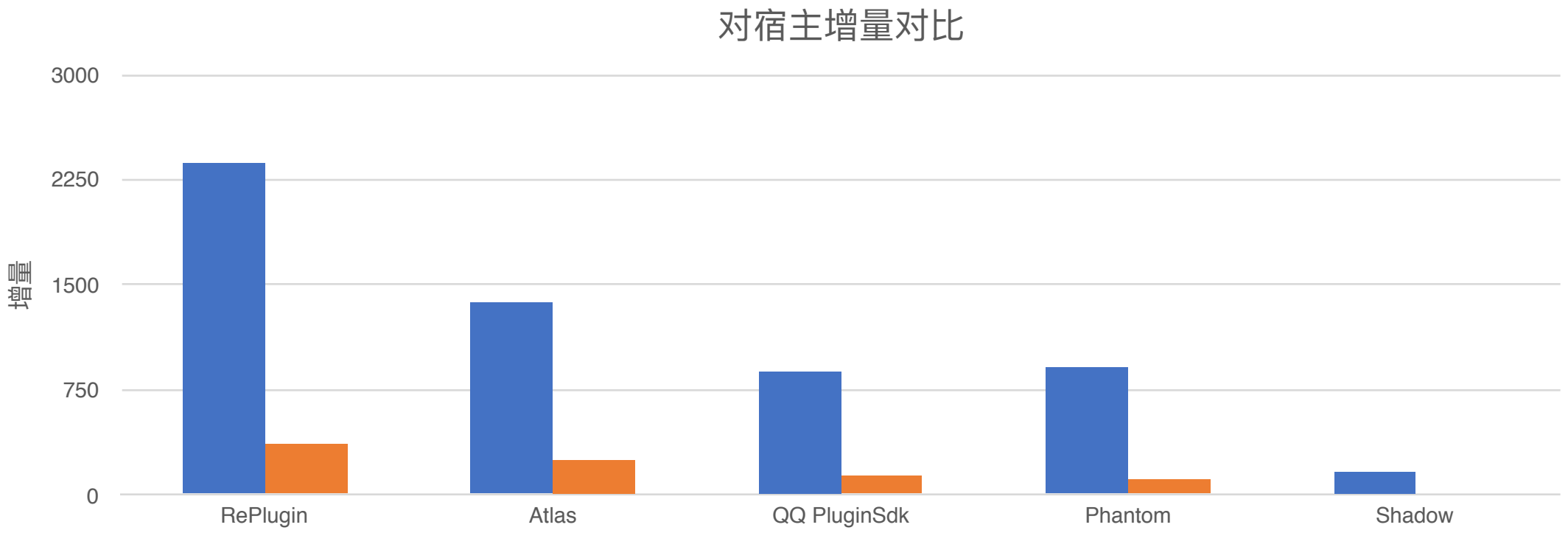
插件框架本身也有更新需求

Shadow做出了哪些创新?



全动态插件架构

Shadow做出了哪些创新?



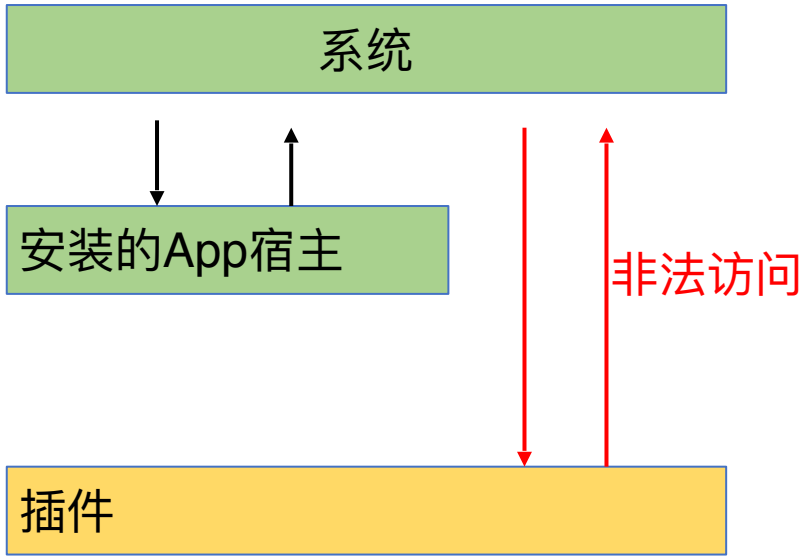
全动态架构对宿主增量极小

Shadow核心技术原理

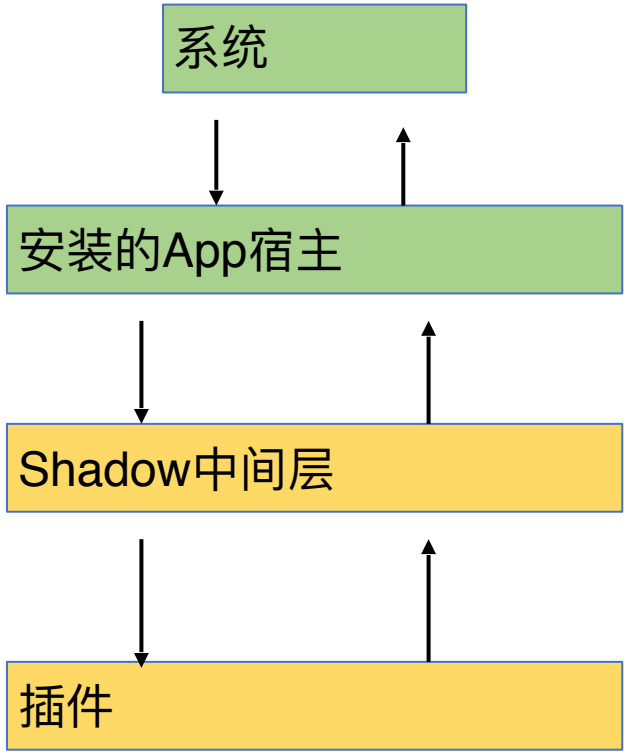
那么我们是如何实现这些创新呢?

Shadow核心技术原理

其他框架：



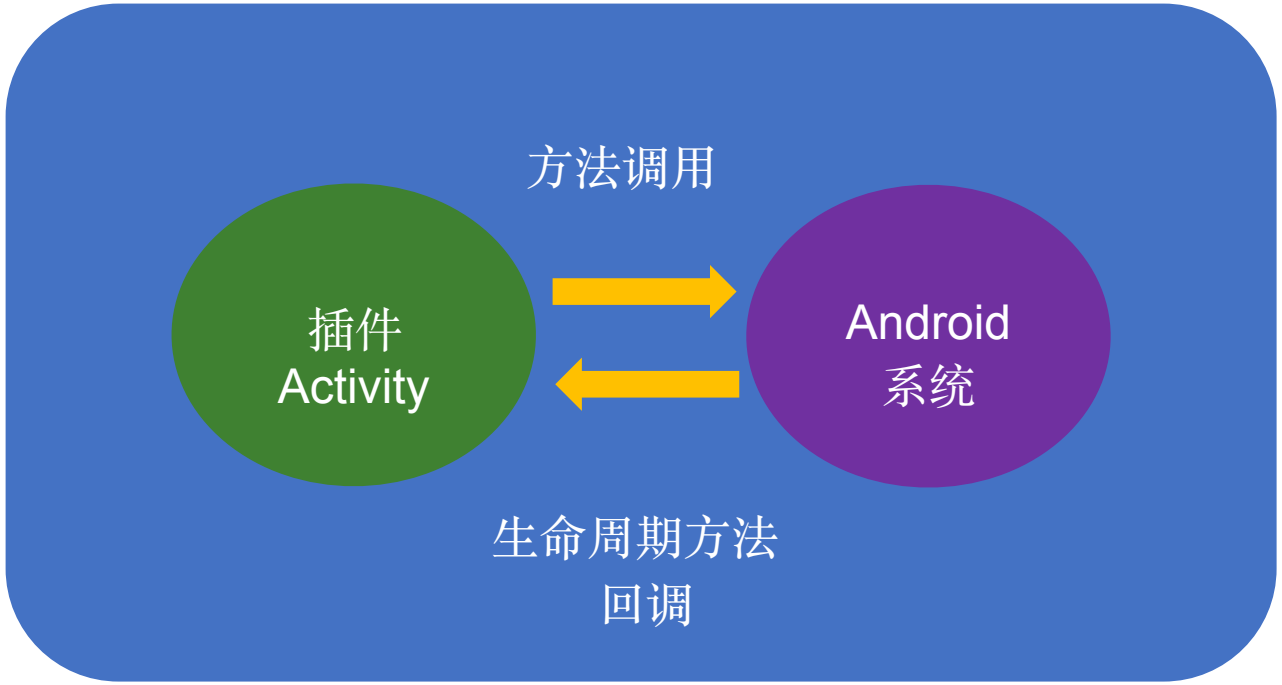
Shadow：



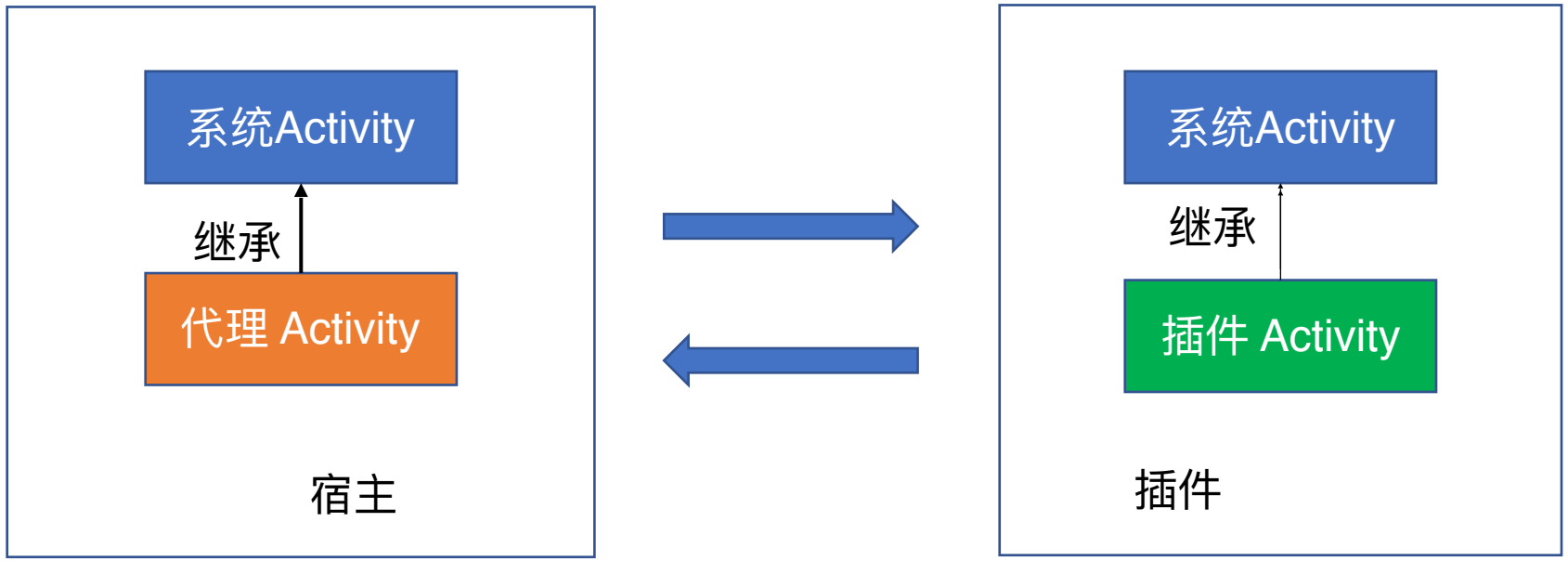
Shadow基本原理

Shadow核心技术原理

插件Activity正常运行2 个条件：系统方法调用成功
生命周期方法正确回调

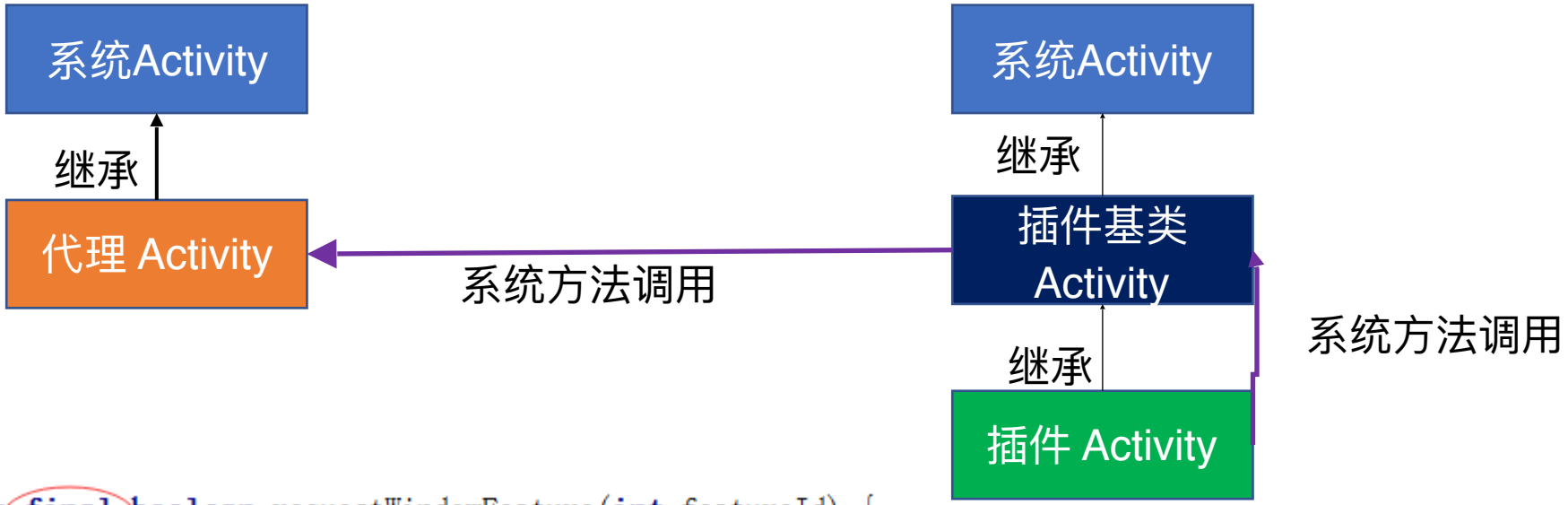


Shadow核心技术原理



代理模式开发插件Activity的基本原理

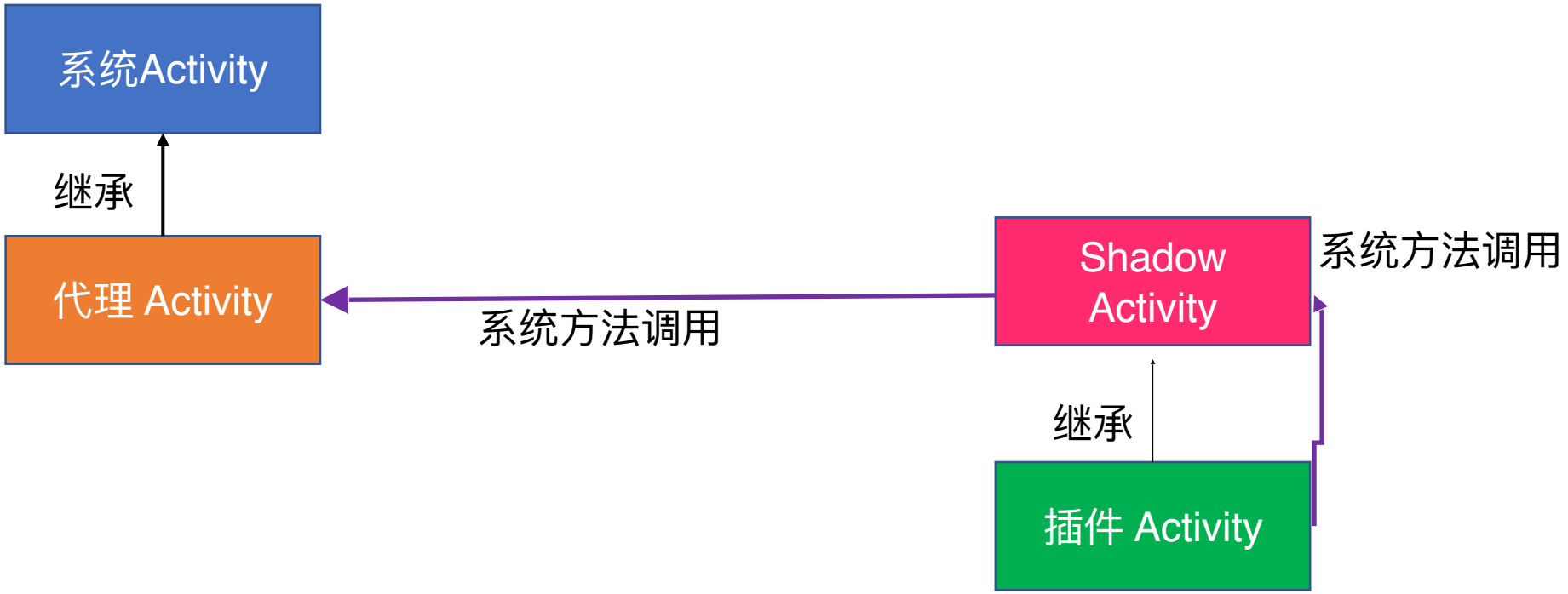
Shadow核心技术原理




```
public final boolean requestWindowFeature(int featureId) {
    return getWindow().requestFeature(featureId);
}

public final void setFeatureDrawableResource(int featureId, @DrawableRes int resId) {
    getWindow().setFeatureDrawableResource(featureId, resId);
}
```

常规代理模式系统方法调用

Shadow核心技术原理



Shadow中的插件Activity方法调用

Shadow核心技术原理


```
class ShadowActivity {
    public void onCreate(Bundle savedInstanceState) {
    }
}

class PluginActivity extends ShadowActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        System.out.println("Hello World!");
    }
}

class ContainerActivity extends Activity {
    ShadowActivity pluginActivity;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        pluginActivity.onCreate(savedInstanceState);
    }
}
```

```
class ContainerActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        System.out.println("Hello World!");
    }
}
```

常规代理模式生命周期方法回调

Shadow核心技术原理


```
class PluginActivity extends ShadowActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        savedInstanceState.clear();
        super.onCreate(savedInstanceState);
    }
}
```

```
class ContainerActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        savedInstanceState.clear();
    }
}
```

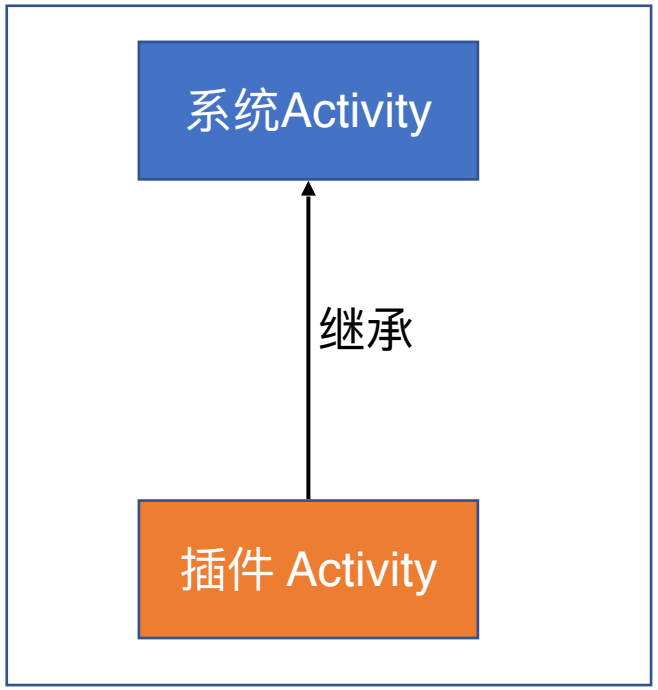
常规代理模式方法回调有执行顺序问题

Shadow核心技术原理

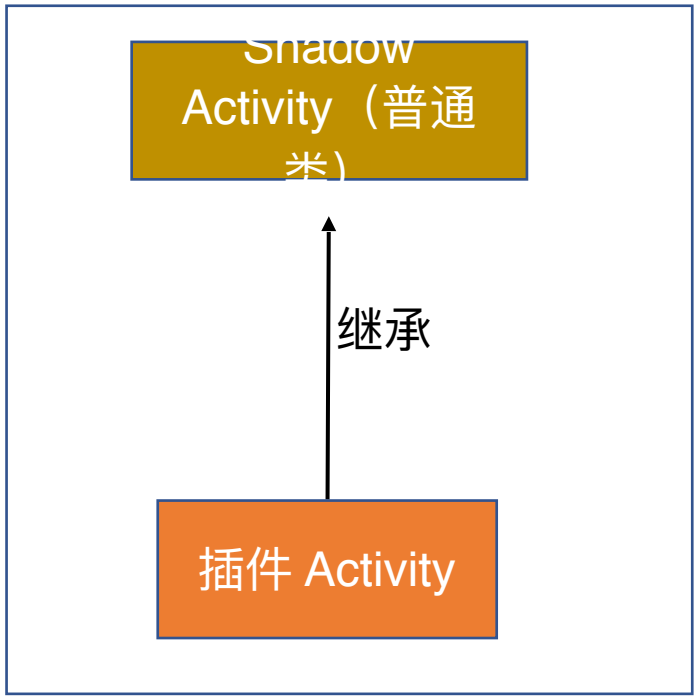

```
class ShadowActivity {
    ContainerActivity containerActivity;
    public void onCreate(Bundle savedInstanceState) {
        containerActivity.superOnCreate(savedInstanceState);
    }
}

class PluginActivity extends ShadowActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        savedInstanceState.clear();
        super.onCreate(savedInstanceState);
    }
}

class ContainerActivity extends Activity {
    ShadowActivity pluginActivity;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        pluginActivity.onCreate(savedInstanceState);
    }

    public void superOnCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

Shadow采用组合解决

Shadow核心技术原理



正常APP开发

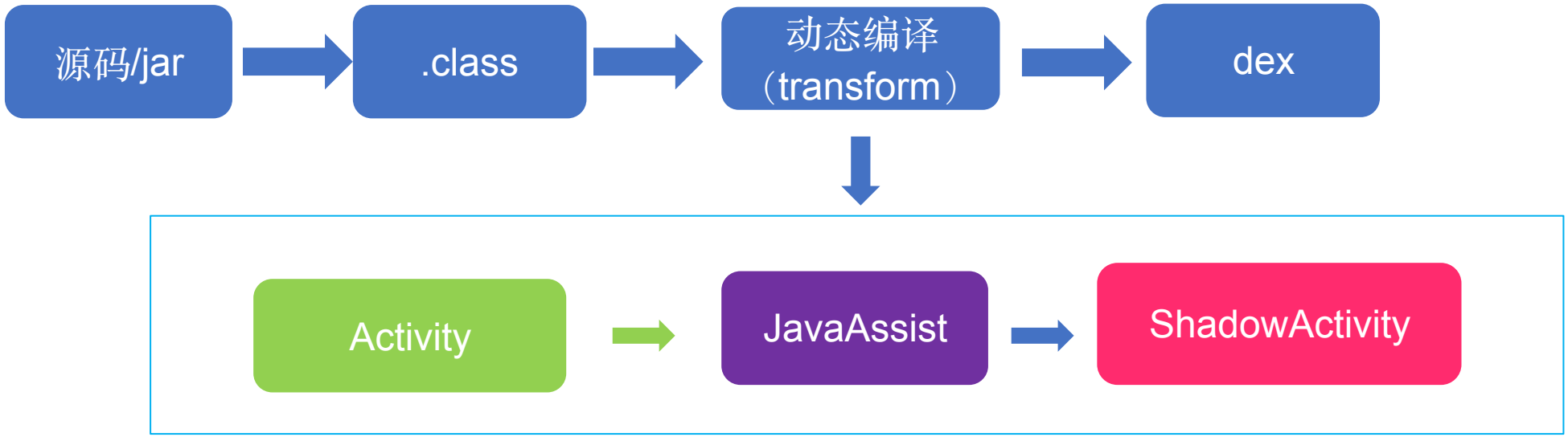


Shadow插件开发

代理模式有代码侵入性问题

Shadow核心技术原理

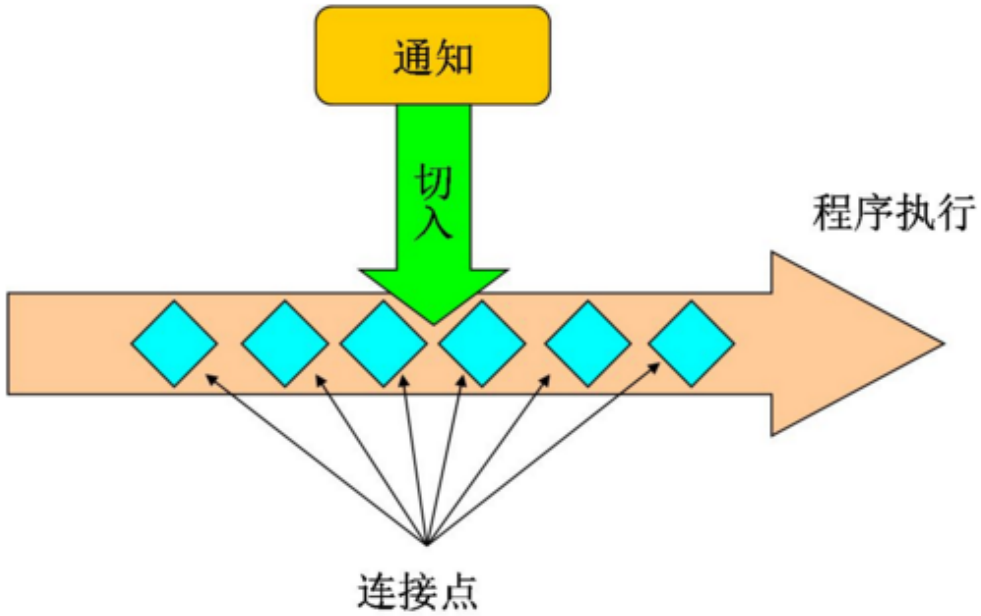
代理模式对开发插件带来了代码侵入性，同时无法处理jar包中activity



Shadow对代码侵入性的解决方案

Shadow核心技术原理

AOP——面向切面编程



插件开发就是不断寻找合适的切面

Shadow核心技术原理

插件Activity的intent，系统无法识别，需要转换为对应壳子Activity的intent

PendingIntent
.*getActivity*(context, requestCode, intent, flags, options);

ShadowPendingIntent
.getActivity(context, requestCode, intent, flags, options);

大部分切面都能通过Javassist轻松修改

Shadow核心技术原理

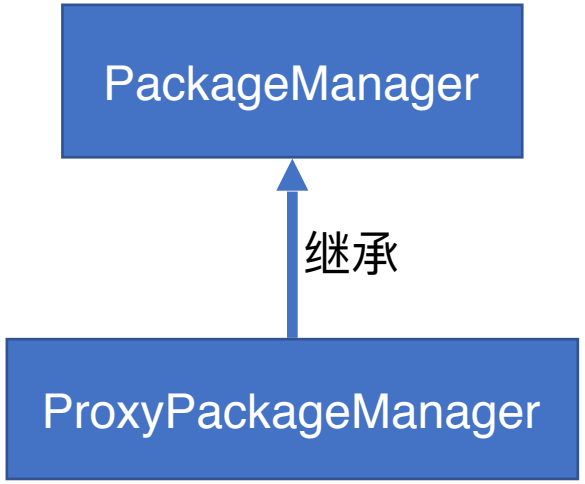

```
ApplicationInfo getApplicationInfo(String packageName, int flags)

ActivityInfo getActivityInfo(ComponentName component, int flags)

ServiceInfo getServiceInfo(ComponentName component,int flags)
```

getPackageManager() -> PackageManager

getPackageManager() -> ProxyPackageManager

常规插件框架对PackageManager的处理

Shadow核心技术原理



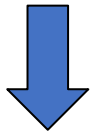
@Override
getApplicationInfo()
@hide abstract
hideMethod()

@Override
getApplicationInfo()

继承的方式需要兼容hide及厂商添加方法

Shadow核心技术原理

packageManager.getApplicationInfo(String packageName,int flag)



ShadowPackageManager.getApplicationInfo(packageManager,packageName,flag)

期望的方法转调处理

Shadow核心技术原理

Javassist

Java bytecode engineering toolkit since 1999

redirectMethodCall

```
public void redirectMethodCall(CtMethod origMethod,
CtMethod substMethod) throws CannotCompileException
```

Modify method invocations in a method body so that a different method will be invoked.

Note that the target object, the parameters, or the type of invocation (static method call, interface call, or private method call) are not modified. Only the method name is changed. The substituted method must have the same signature that the original one has. If the original method is a static method, the substituted method must be static.

Parameters:

`origMethod` - original method

`substMethod` - substituted method

Throws:

`CannotCompileException`

Javassist没有对应支持的高级API

Shadow核心技术原理


```
Class A {
public D add(B b, C c) { }
}
```

1. aload_0
2. bipush 12
3. bipush 13
4. invokevirtual #A.add(B,C)
5. ireturn

```
Class S {
public static D add(A a, B b, C c) { }
}
```

1. aload_0
2. bipush 12
3. bipush 13
4. invokestatic #S.add(A,B,C)
5. ireturn

分析2种方法的调用区别

Shadow核心技术原理

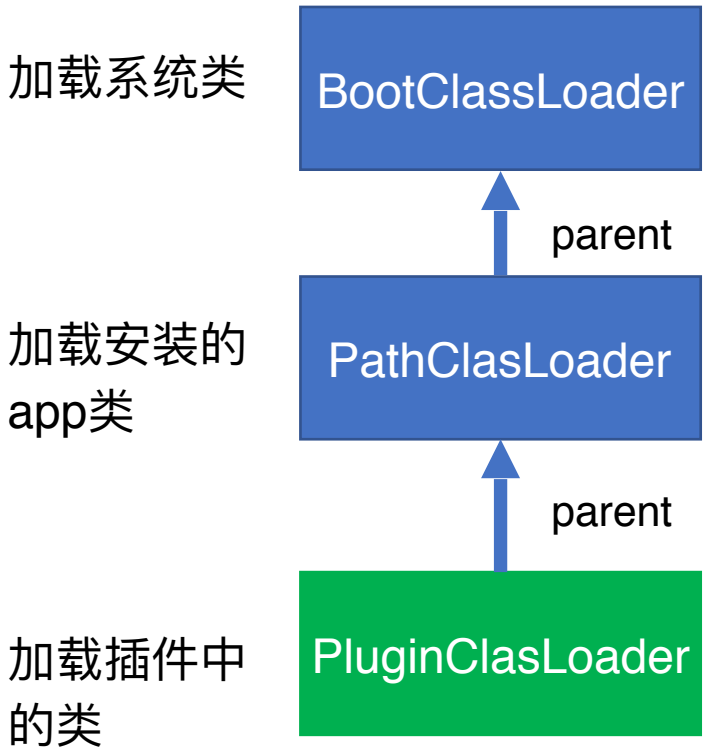

```
@Throws(BadBytecode::class)
override fun match(c: Int, pos: Int, iterator: CodeIterator,
                   typedesc: Int, cp: ConstPool): Int {
    if (newIndex == 0) {
        val desc : String! = Descriptor.insertParameter(classname, methodDescriptor)
        val nt : Int = cp.addNameAndTypeInfo(newMethodname, desc)
        val ci : Int = cp.addClassInfo(newClassname)
        newIndex = cp.addMethodrefInfo(ci, nt)
        constPool = cp
    }
    iterator.writeByte(Opcode.INVOKESTATIC, pos)
    iterator.write16bit(newIndex, index: pos + 1)
    return pos
}
```

非常小的改动达到了目的

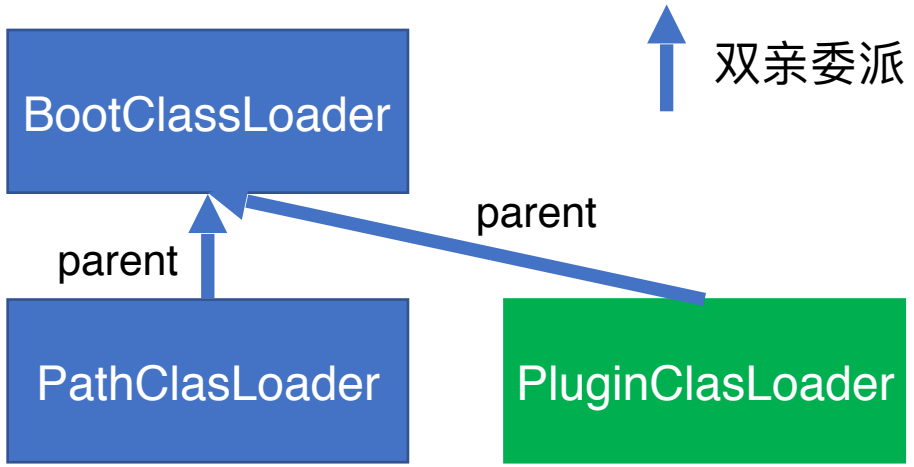
Shadow核心技术原理

全动态插件架构有什么创新点？

Shadow核心技术原理



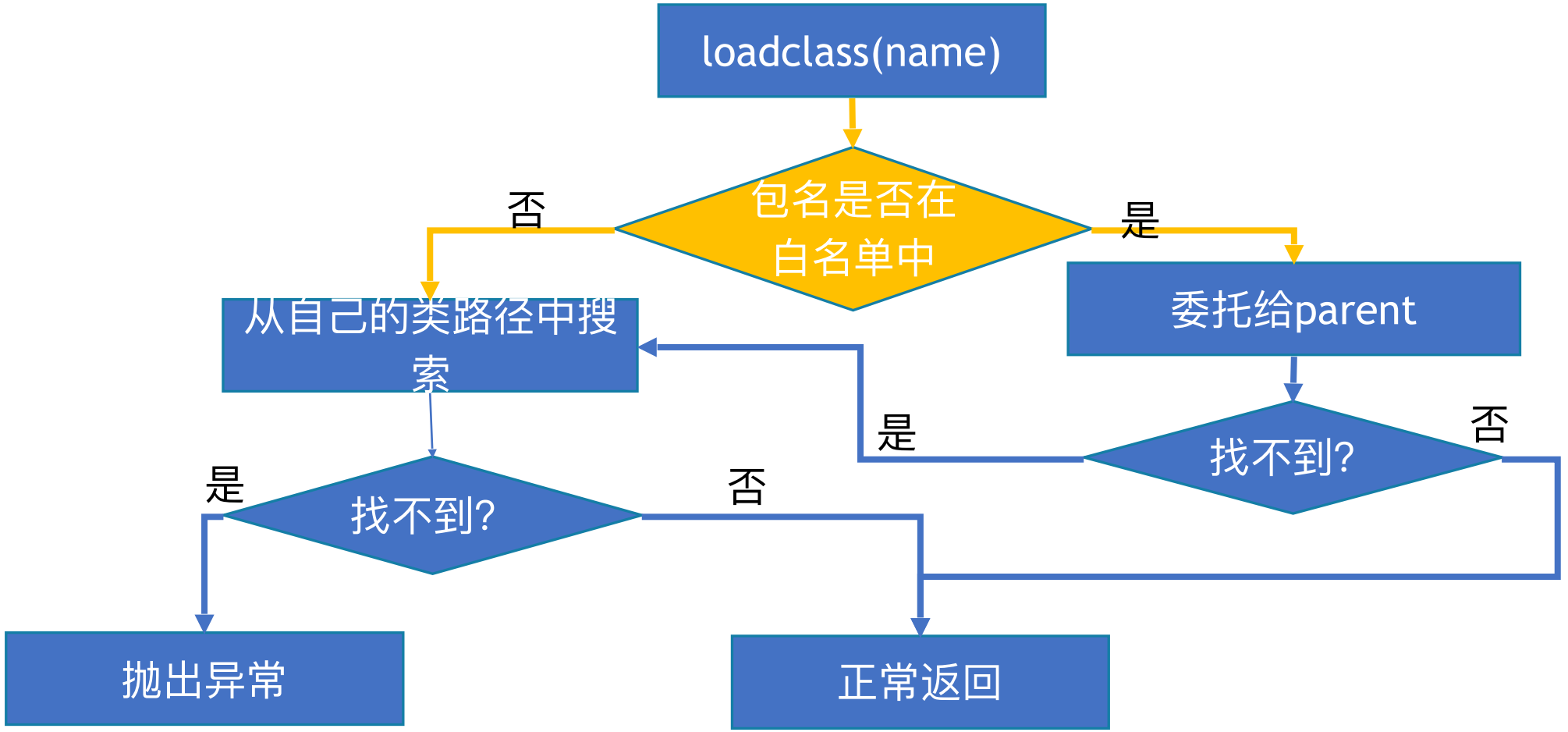
存在类冲突问题



无法复用宿主的类

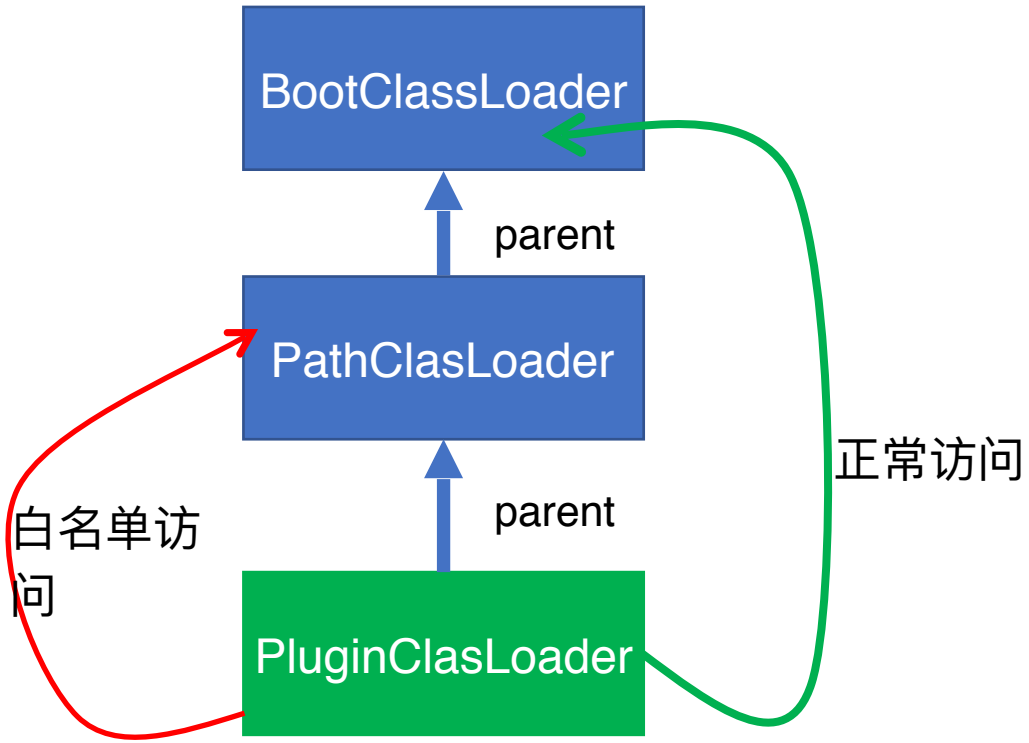
传统的ClassLoader类加载有局限性

Shadow核心技术原理



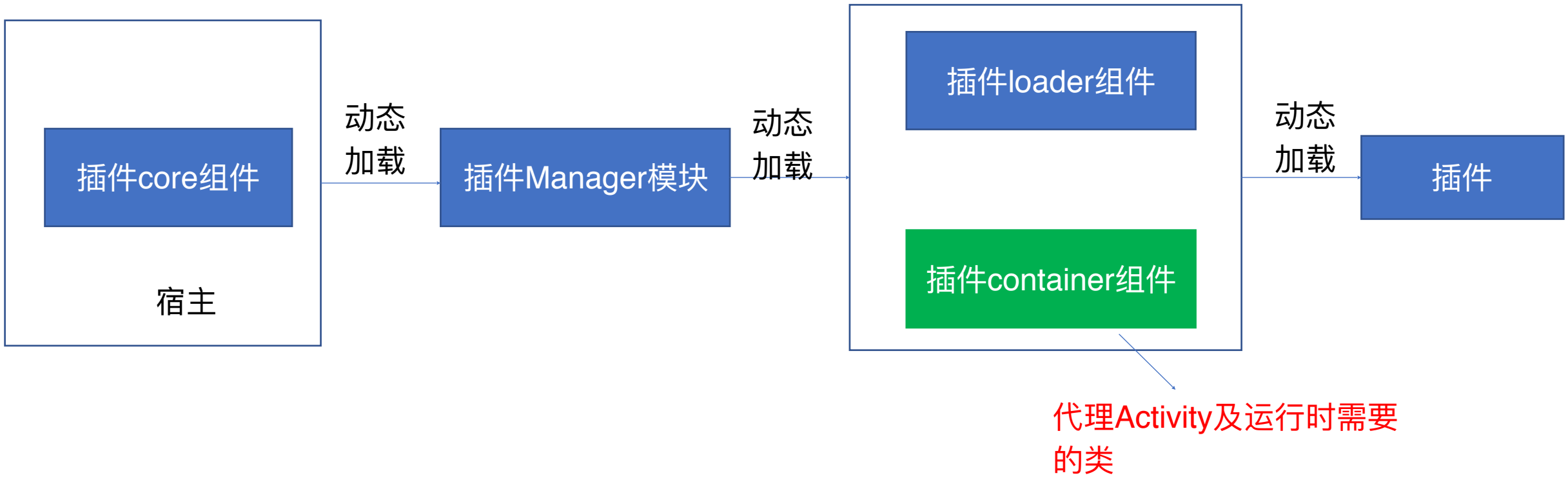
定制ClassLoader支持白名单访问

Shadow核心技术原理



定制插件ClassLoader类访问说明

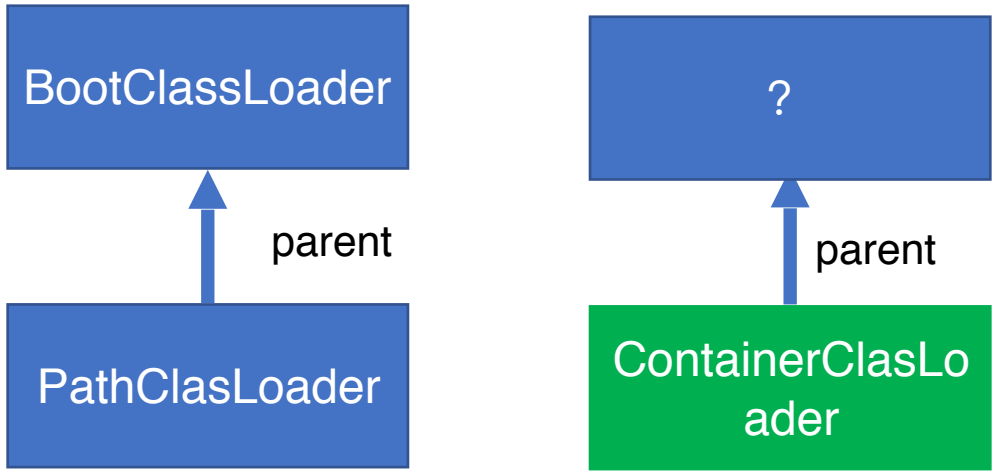
Shadow核心技术原理



回顾一下全动态插件架构

Shadow核心技术原理

系统启动Activity时，总是从PathClassLoader去查找



传统的ClassLoader双亲委派是否可以打破?

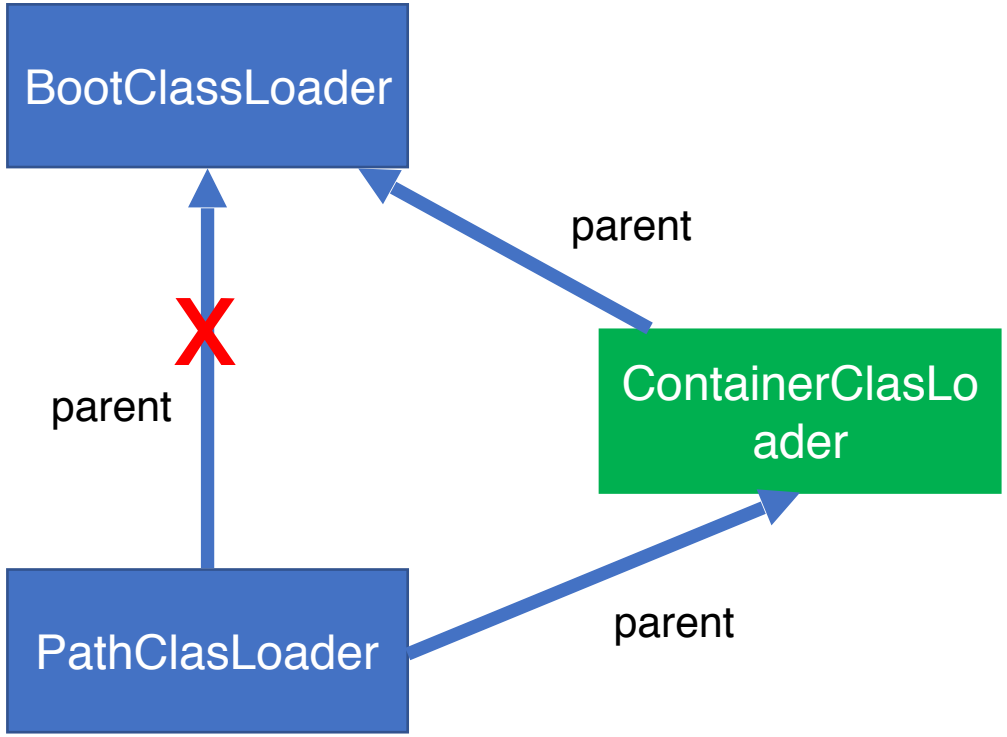
Shadow核心技术原理


```
public abstract class ClassLoader {

    private final ClassLoader parent;

    //.......

}
```

反射私有变量parent就能改变原有双亲委派结构

阅读源码寻找双亲委派的逻辑

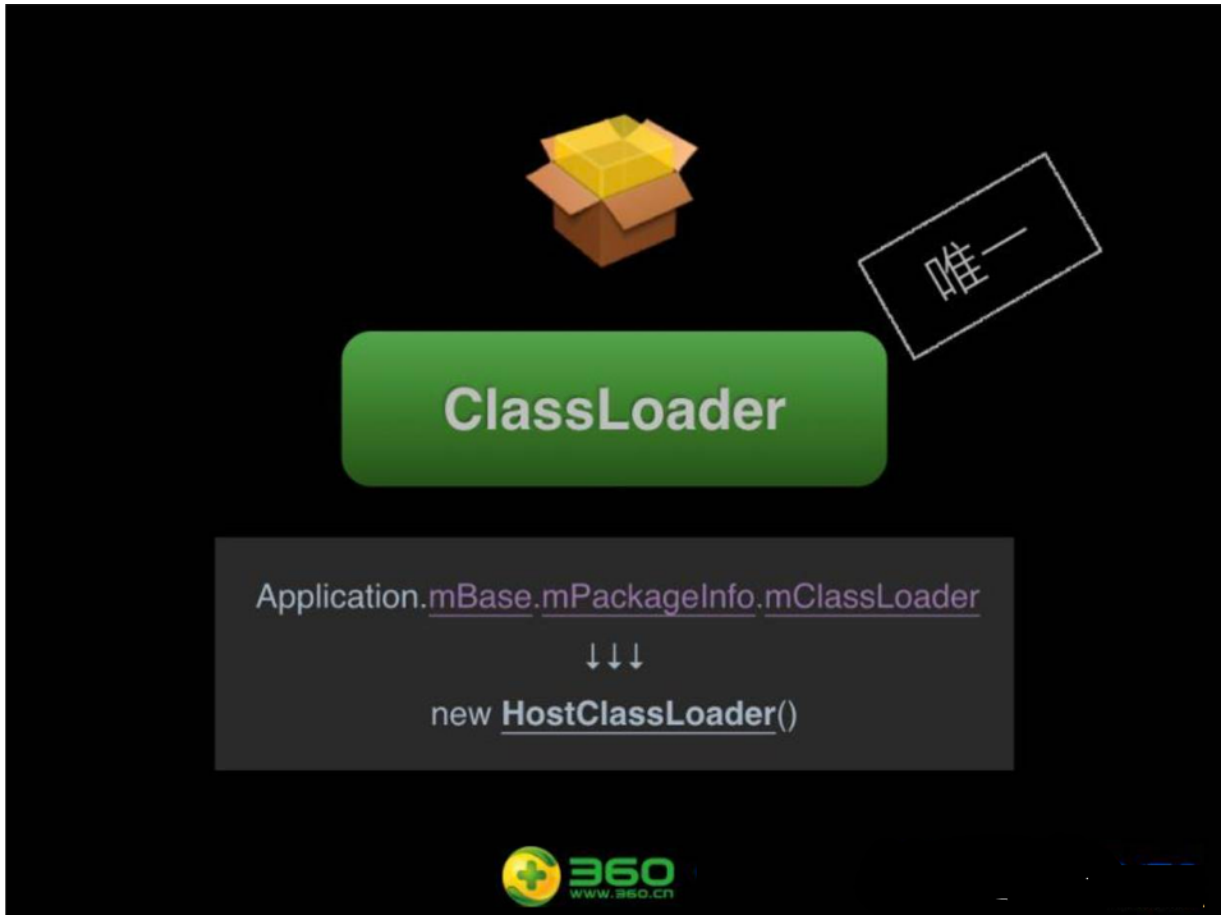
Shadow核心技术原理



- 修改安全
- 非插件框架必须点

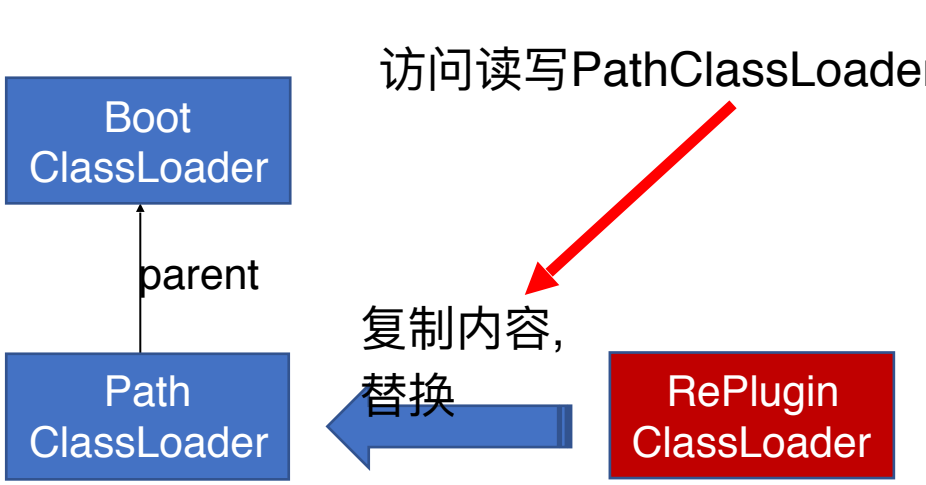
打破ClassLoader的双亲委派

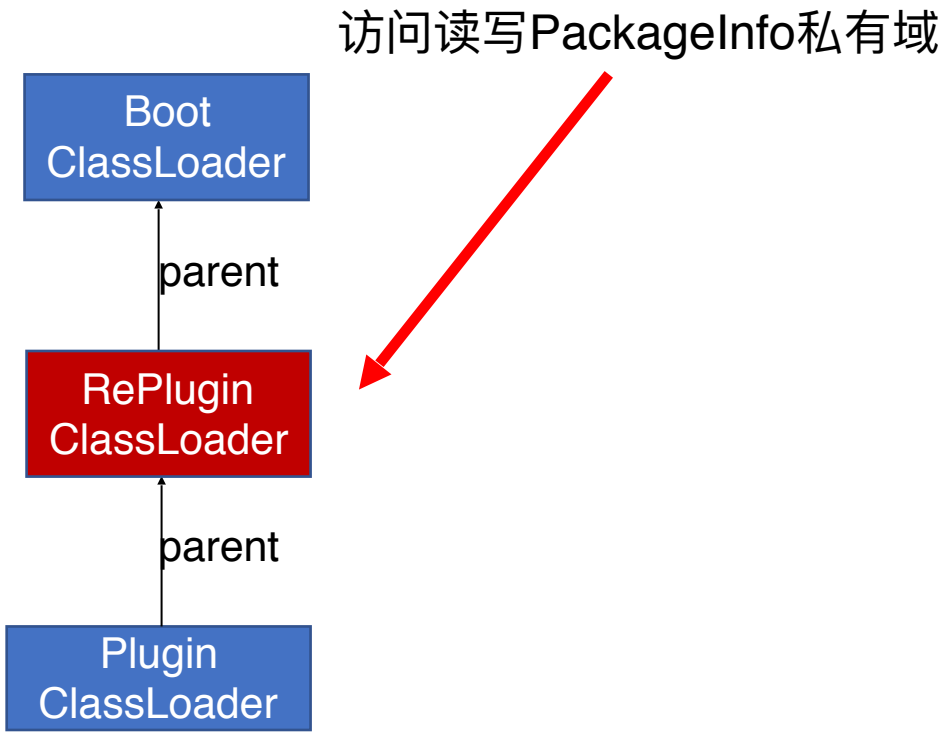
Shadow核心技术原理



优化RePlugin的核心技术

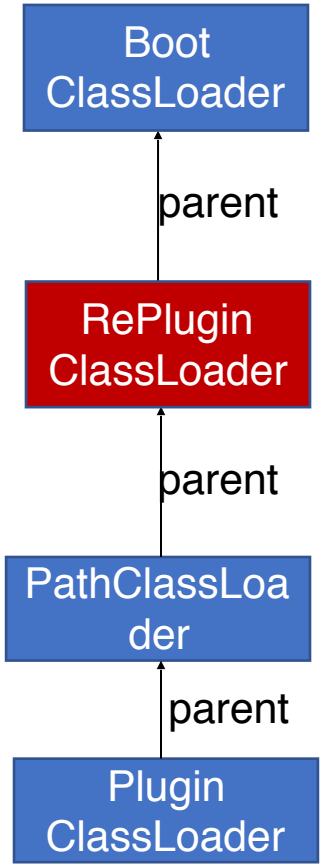
Shadow核心技术原理





RePlugin唯一的Hook点实现

Shadow核心技术原理



无限制API调用实现Replugin核心hook

总结

- 基础知识非常重要，总能在关键问题上起决定性作用
- 任何软件工程领域的问题，都可以通过增加一个中间层来解决
- 不建议使用反射，只有在面向未来编程时才是正确的解决方案

开源

Shadow GitHub开源地址:

https://github.com/Tencent/Shadow

欢迎star，issue，pr

Thank you for your attention!

郭琨