去中心化组件方案Coco

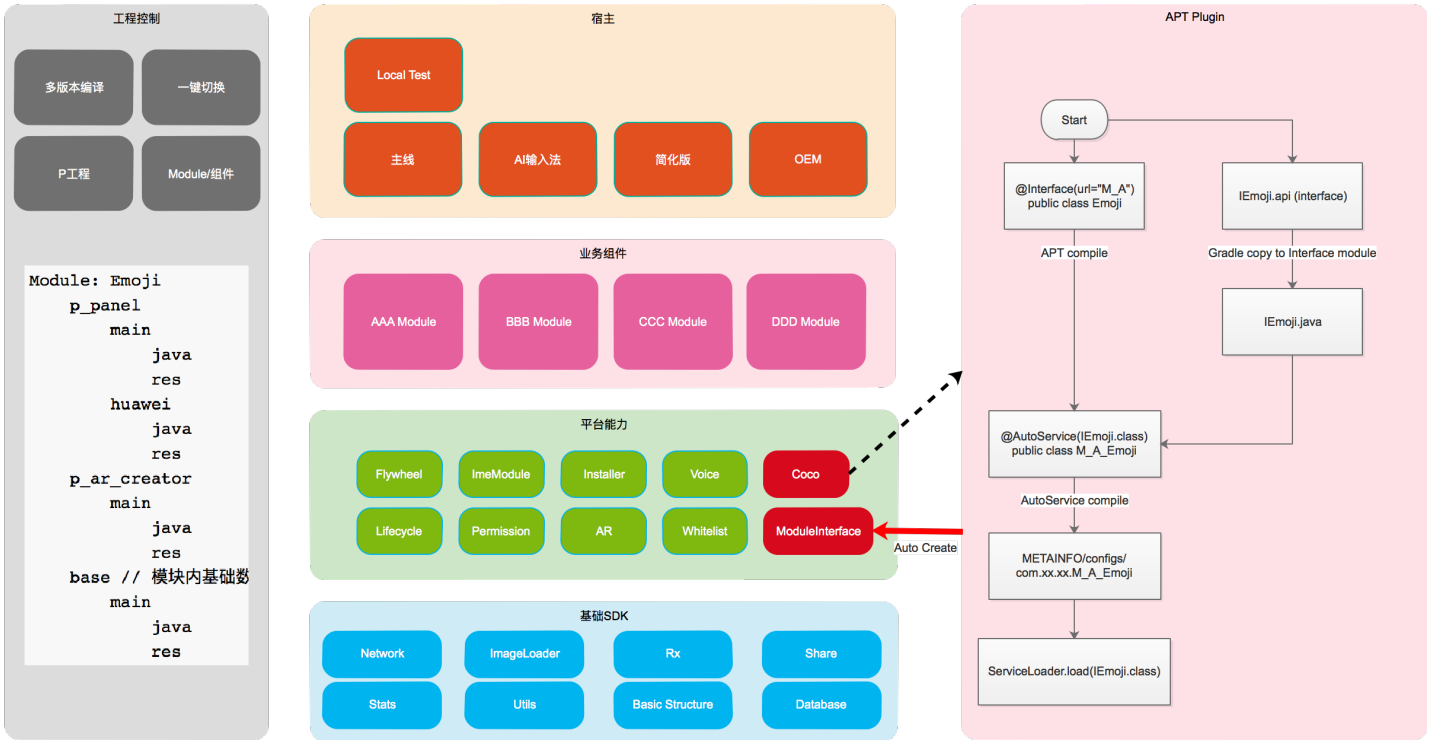
0. 概述

核心理念：

1. 调用基于接口，而非路由，放弃中心路由管理模块：保持正常开发逻辑，降低迁移成本。降低因路由带来的性能损耗与稳定性损耗
2. 通信基于组件间P/S，而非事件总线，放弃中心总线管理模块：保持整个调用路径清晰可控，影响范围缩小到通信组件双方。降低多组件间通信信道卡死的可能。

1. 总体架构

整体框架分为3个方向：



1. 项目工程：输入法项目本身，4层架构，包括基础SDK，平台能力，业务组件，宿主
2. Gradle编译控制：提供多版本编译控制能力，规范组件代码组织
3. Gradle组件化插件：提供编译期实现组件化能力

2. 组件化框架

2.1. 组件方案：

- ServiceLoader + Annotation实现组件调起、管理
- bpi接口机制实现组件间接口、数据模块共享
- 基于Interface实现组件间调用，如Bitmap
- 基于P/S架构实现组件间事件发布、通信
- 基于P工程拆分组件内功能点
- 基于flavor拆分版本定制化

实现 & 注册

```
@CocoInterface(Emoji.class)
public class EmojiModule extends CocoBaseModule implements Emoji {
}

public interface Emoji {
    View getEmojiView();
    Bitmap getAnimojiIcon();
}
```

使用

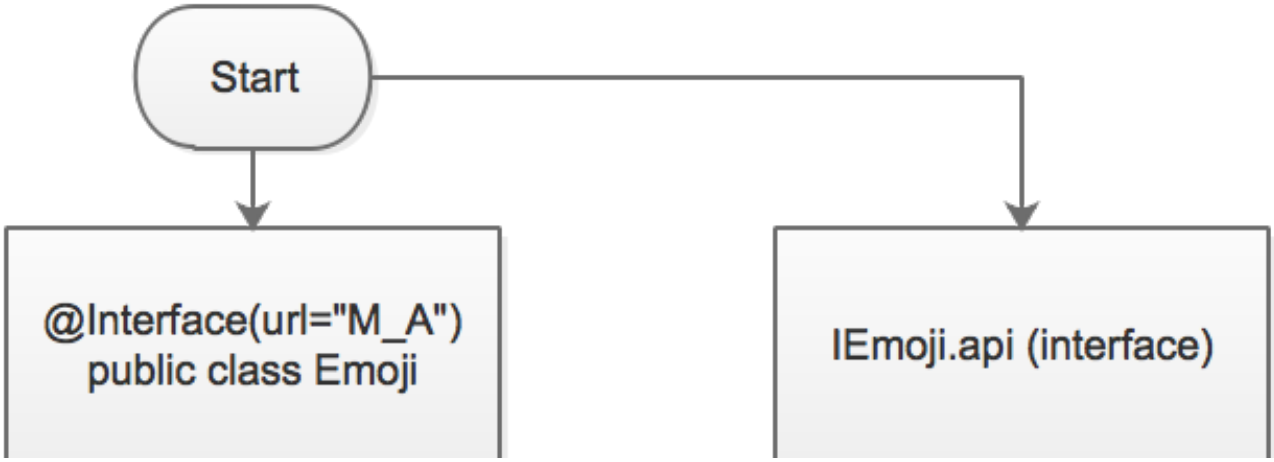
```
Emoji emoji = Coco.findModule(Emoji.class);
```

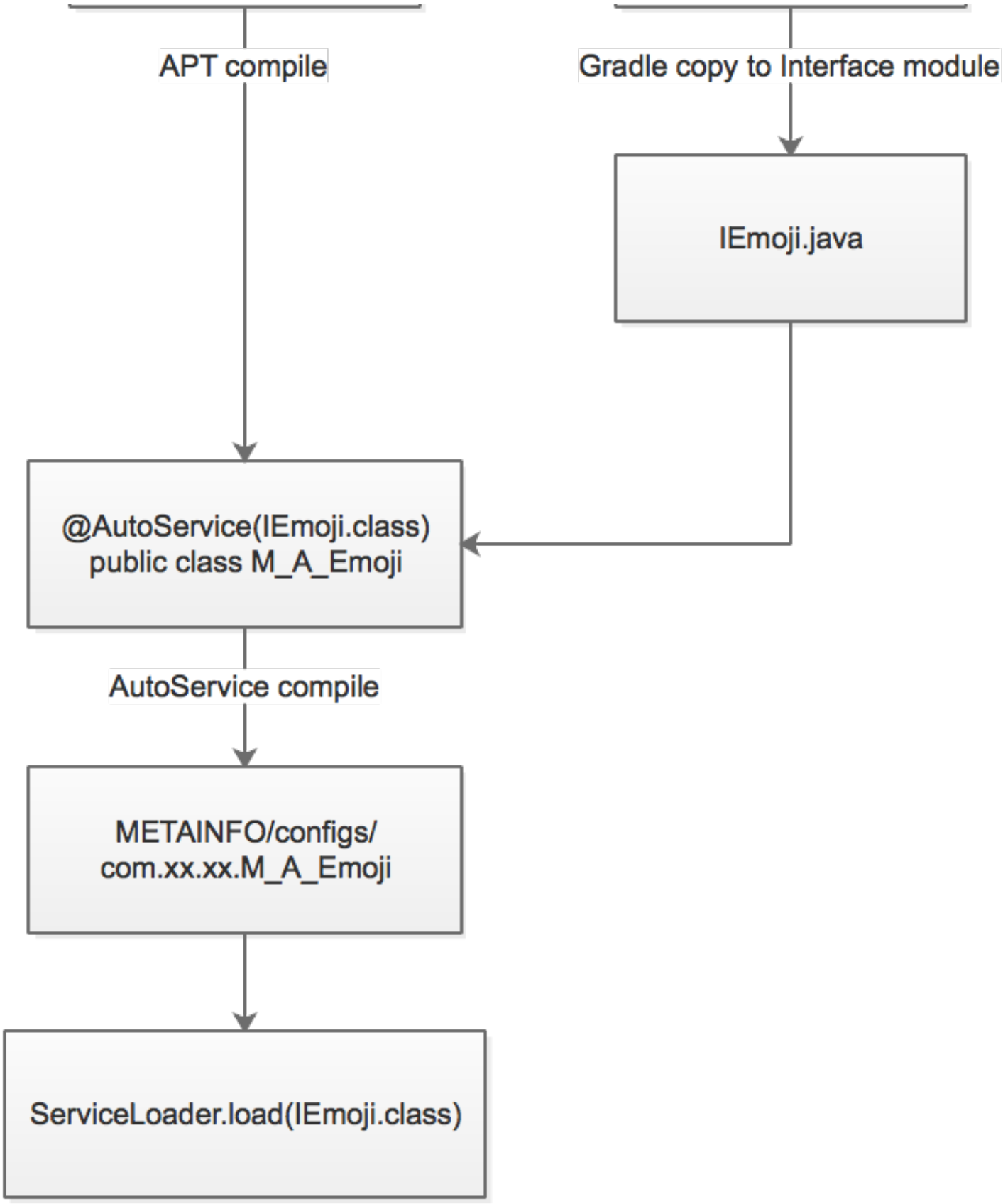
```
// 订阅
Coco.subscribe(Emoji.class, EmojiEvent.NAME, subscriber);

// 发布
CocoBaseModule.publish(CocoEvent event);
```

2.2. 技术细节

2.2.1 APT Plugin编译流程 - 组件注册





采用 `.api` 后缀文件作为需要暴露给外部模块使用的服务接口。编译时Gradle插件会扫描所有 `src/` 目录下所有该类型文件，自动将其下沉到 `Interface Module` 中。

同时提供 `@Interface` annotation，所有该标注类需要实现上述接口，编译时Gradle插件会自动生成一个对应的 `@AutoService` 标注的类。

示例如下：

Voice.api:

```
public interface Voice {
    public View getVoicePanel();
}
```

VoiceImpl.java:

```
@CocoInterface(url="voice")
public class VoiceImpl implement Voice {
    public View getVoicePanel() {
        return new VoicePanel();
    }
}
```

2.2.2. ServiceLoader优化与改进 - 组件调起

Service缺陷：

1. serviceloader没有缓存功能。因为对于服务来说，大部分我们都需要使用单例模式，而不会频繁的生成新的实例。
2. serviceloader使用无参的构造方法进行构建实例。
3. serviceloader是一种非显式的调用服务实现类方式，如果不在proguard中保护这些实现类，那么肯定会被shrink掉。
4. serviceloader配置文件需要手动注册的缺点

优化：

1. 提供缓存实现。
2. 要求所有服务必须继承CocoInterface基类，该类中定义了 `init(Context context)` 方法，对于serviceloader加载的类，都会首先调用该方法。
3. Proguard只需写 `* extends CocoInterface` 即可。
4. 通过AutoService实现自动注册

2.2.3. 组件管理与生命周期

框架提供组件的创建与销毁能力：

- `findInterface(String url)`，
- `removeInterface(String url)`，
- `destroyInterface(String url)`

同时提供如下生命周期：

```
/**
 * 组件被实例化时调用
 */
void onCreate();

/**
 * 组件remove后再次获取时调用
 */
void onReload();

/**
 * 组件remove时调用，释放module持有的相关缓存
 */
void onRelease();

/**
 * 组件销毁时调用，module被销毁，释放所有资源，保存必要信息
 */
void onDestroy();

/**
 * loadType采用Dependency方式时，在这里配置其它依赖模块，用于模块预加载实现
 */
void dependency();
```

组件框架同时提供一些Module调起的简单配置：

```
@CocoInterface(url="X", loadType = Lazy/Immediate/Dependency)
```

3. 工程编译控制

工程结构

```
Module: Emoji
    p_panel
        main
            java
            res
        huawei
            java
            res
    p_ar_creator
        main
            java
            res
    base // 模块内基础数据类，工具类
        main
            java
            res
```

P工程：

pins工程能在module之内再次构建完整的多子工程结构，通过project.properties来指定编译依赖关系。通过依赖关系在编译时找到所有的资源和源码路径。而对边界的约束需要配合code-check工具在编译期进行检查，杜绝依赖关系之外的代码引用。

粒度极小，一个pin工程也许只有一个源文件，只要它能表达一个独立职责。对于任何一个模块，从内部约束自己的功能结构，是对整体代码边界约束的极大补充。以前面插的结构为例，一个gallery业务可能提供了几种不同的产品功能，以及支撑能力。那么将其相互独立的代码进行区分，避免混杂，就会显得十分必要。清晰的结构，意味着后期维护成本的降低和开发效率的提高，留下了灵活性。

pins工程某种程度上能减少一些粒度太小的module工程，也一定程度的缓解太多module工程时的gradle编译性能问题。

```
sourceSets {
    main {
        def dirs = ['p_widget', 'p_theme',
                    'p_shop', 'p_shopcart',
                    'p_submit_order','p_multperson','p_again_order',
                    'p_location', 'p_log','p_ugc','p_im','p_share']
        dirs.each { dir ->
            java.srcDir("src/$dir/java")
            res.srcDir("src/$dir/res")
        }
    }
}
```

4. 基础SDK与平台能力

4.1. 基础SDK

1. Utils：整理/剥离
2. Stats：完善
3. Share：整理/剥离
4. Rx：引入Rx工具
5. ImageLoader：模块升级
6. Network：暂无工作
7. 基础容器：低优
8. Storage

4.2. 平台能力

1. Flywheel：完善
2. Permission：整理/剥离
3. 语音：整理/剥离
4. WhiteList：整理/剥离/完善
5. AR：整理/剥离
6. Installer：设计/实现
7. SkinRender
8. GlobalSetting
9.

4.3. 输入法框架能力

4.3.1. Lifecycle

Lifecycle模块提供整合的生命周期机制，包括

1. 组件自身生命周期：

```
void onCreate();
void onReload();
void onRelease();
void onDestroy();
void dependency();
```

2. 输入法系统生命周期 & 输入法扩展生命周期：

```
void onCreate
void onInitializeInterface
void onStartInput
void onStartInputView
...
```

3. 组件View级别的生命周期：

```
void onCreate();
void onVisible();
void onInVisible();
void onRemove();
```

4.3.2. ImeModule框架

ImeModule框架提供对InputMethodService的代理，将组件对APP模块的依赖进行解耦。这部分具体可见目前输入法中的代码。

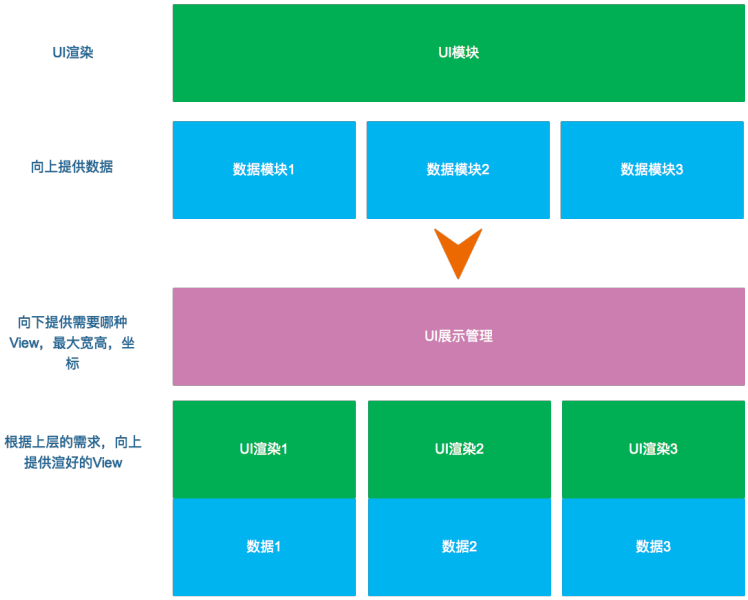
同时ImeModule还负责组件间事件通信机制。该机制基于RxBus传递一对多消息通信。

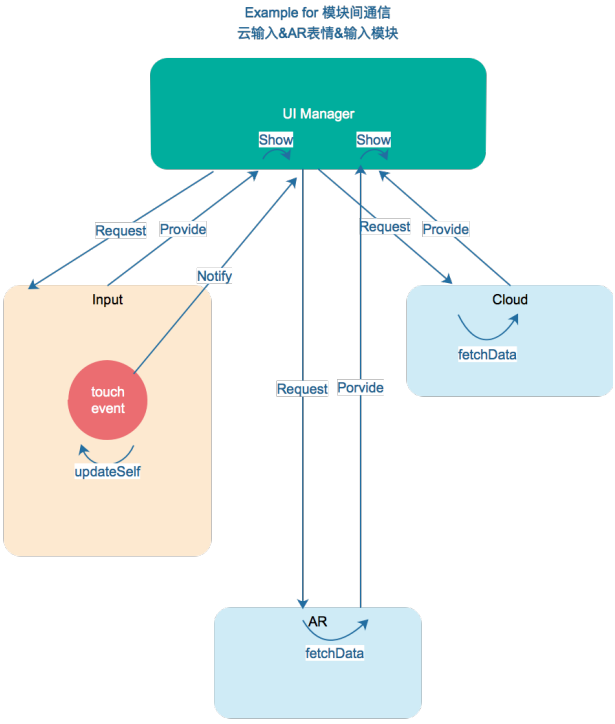
5. 业务组件框架

5.1. Lifecycle-MVP & Lifecycle-MVVM

6. 业务拆分方案

1. 从ImeService开始剥离，按功能模块，优先抽离AI输入法需要的模块。
2. 拆分方案如下：





UIManager：负责展现逻辑，module提供的view整合
Module：负责模块内部的View渲染，更新，事件处理，所有事件会通知UIManager转发给其它module

Request：渲染需求
Provide：View提供
Notify：事件通知