详解Componentization库注解

> 此内容重点分析注解定义和解析部分，关于插件部分单独分析，此处可能会略带提及。

一、注解定义

位置：`annotation`模块。

首先看模块gradle配置分析，下面为`build.gradle`文件内容

Groovy

```
// 注解类的定义比较纯粹，只涉及到语言本身，故只需要java基本开发支持插件即可
apply plugin: 'java-library'

// 依赖库
dependencies {
    // 以为会使用到androidx中的注解库，其实没用到，所以可选
    // 也应该需要注意到，androidx中的注解库其实也是一个纯java组件，不是aar归档格式
    api 'androidx.annotation:annotation:1.1.0'
}

// 开发的组件都需要发布到maven仓库，供主体项目使用，故而引入了相关maven构建上传逻辑
// 关于maven打包上传配置会在其他文档中介绍
apply from: "${rootProject.file('mvn-push.gradle')}"
```

此模块内容相对简单，下面简单讲下注解设计意图，

> 为了让注解有最全的可见性，我们声明其生命周期为运行时，这样gradle插件和反射都可以获取到，即`@Retention(RetentionPolicy.RUNTIME)`。

下面对注解进行分析一下，

- `@Api`，

Java

```java
/**
 * api接口声明注解，实现类必须被标记{@link Service}标记才可实现自动注册
 */
@Retention(RetentionPolicy.RUNTIME)
// 限制只能标记到接口或者虚拟类上
@Target(ElementType.TYPE)
public @interface Api {

  /**
   * 表示接口实现是否单例模式，可以和kotlin的object单例对象保持兼容，
   * 此机制也是考虑到一些情况，比如需要使用Application作为实例提供者，kotlin的object等
   * 关于单例的提供方式，需要关注@Provider注解和Componentization#newInstance()
   */
  boolean singleton() default false;

}
```

使用示例，先声明一个接口，并用`@Api`标记其实例模型为单例，

Kotlin

```kotlin
/**
 * 请注意，所有接口必须继承自API，此限制主要是为了统一到一种类型，而不是Object类型
 * 也是一种向后兼容的预留类型机制，以后加入共有逻辑接口就比较好处理
 */
@Api(singleton = true)
interface LibraryAPI: API {

  fun invokeMe()

}
```

> 关于接口的实现请查看下面`@Service`相关内容。

- **`@Service`**，继续看注解定义代码，

Java

```java
/**
 * API接口实现声明标记，其继承的接口必须被@Api注解标记
 */
@Retention(RetentionPolicy.RUNTIME)
// 限制只能标记到类型上
@Target(ElementType.TYPE)
public @interface Service {}
```

我们实现 `LibraryAPI` 接口，

Kotlin

```kotlin
/**
 * 由于在LibraryAPI中标记为单例，此处使用kotlin的object单例形式。对于此机制，
 * 了解到kotlin编译为LibraryService中的INSTANCE静态实例，在字节码层面直接获取即可。
 * 如果不使用object关键词机制，其实例模型为第一次调用时反射创建，内部通过Map进行缓存，
 * 具体可以查看componentization模块中Componentization相关getXXX实现。
 */
@Service
object LibraryService: LibraryAPI {
  override fun invokeMe() {
    Log.e("LibraryService", "调用invokeMe()接口")
  }
}
```

假设有自定义实例创建的需求怎么办，我们提供了@Provider注解标记机制，接着往下看，

- `@Provider`，

Java

```java
/**
 * 标记服务实例提供方式，一般用来在单例中使用，用于指定单例引用，只能标记到静态成员上。
 * 如果标记到方法上，方法返回值必须为API类型，如果标记到属性上，则属性也必须为API类型。
 */
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Provider {}
```

这样比较抽象，还是看下面的示例，

Kotlin

```kotlin
/**
 * 普通使用方式
 */
@Service
class LibraryService: LibraryAPI {

  companion object {
    /**
     * 静态属性
     */
    @Provider
    private lateinit var instance: LibraryAPI

    /**
     * 通过静态方法，可用来实现一些创建过程，跟静态块差不多
     */
    @Provider
    private fun getInstance(): LibraryAPI {
      // 可以提供一些更自定义的创建过程
      if (instance == null) {
        instance = LibraryService()
        ...
      }
      return instance
    }
  }

  override fun invokeMe() {
    Log.e("LibraryService", "调用invokeMe()接口")
  }
}
```

另外一个典型的场景，我们以全局的`Application`对象作为接口实例，此实例无法由开发者创建，

Kotlin

```kotlin
@Service
class TheApplication: Application(), LibraryAPI {

  companion object {

    @Provider
    private lateinit var applicationInstance: TheApplication

  }

  init {
    // 1. 注入对象初始化逻辑，或者在某些特定的生命周期比如onCreate中
    applicationInstance = this
  }

  override fun onCreate() {
    super.onCreate()
    // 2. 通常的初始化逻辑位置
  }

}
```

以上通过3个接口的配合用来阐明接口、实现、实例化这3者之间的关系。接下来是使用端注解，通过下面的分析来理解怎样使用以上声明的接口。首先看相关注解定义，

- `@AutoWired`，

Java

```java
/**
 * 说实话，这个注解的名称是从以前开发java后端Spring中抄过来的，自动绑定的含义。
 * 因为框架的目的是面向接口编程，所以声明都是接口API接口，此注解用来表明此接口实现无须自己创建
 */
@Retention(RetentionPolicy.RUNTIME)
// 接口实例肯定被绑定到属性成员，可以是静态成员，所以限制只能标记到属性上
@Target(ElementType.FIELD)
public @interface AutoWired {

  /**
   * 有些属性并不想或者不能第一时间初始化，但是可以保证在使用的位置一定可用，
   * 可以声明此处的接口为延迟初始化方式，这会把接口实现代理给一个延迟中间类，
   * 具体可以查看componentization模块的LazyDelegate，由注解处理器自动代理延迟逻辑
   * @return 默认延迟初始化
   */
  boolean lazy() default true;

}
```

模拟在Activity中使用，

Kotlin

```kotlin
class MainActivity: Activity() {

  /**
   * 声明api接口类型，并标记为自动绑定，最后通过gradle插件插入赋值代码
   */
  @AutoWired(lazy = true)
  private lateinit var libraryAPI: LibraryAPI

  companion object {
    /**
     * 支持在静态属性上使用
     */
    @AutoWired(lazy = true)
    private lateinit var libraryAPI: LibraryAPI
  }

  protected fun onCreate(saved: Bundle?) {
    super.onCreate(saved)
    libraryAPI.invokeMe()
  }

}
```

> 此实现需要配合gradle插件修改字节码完成，详细请参阅gradle的插件插装字节码实现。

下面注解为工具型注解，不针对开发者使用，用来辅助框架实现，

- `@Meta`，

Java

```java
/**
 * 关于组件辅助描述信息注解，便于非运行时理解关系，比如插件处理过程
 * 由于接口到实现可以是多对1的关系，故而一个service可以对应多个api接口类
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.CLASS)
@interface Meta {

  /**
   * 服务实现类全名
   * @return {@link Class#getName()}
   */
  String service();

  /**
   * 接口类型列表
   * @return {@link Class#getName()}
   */
  String[] api();

}
```

> 可以在注解处理器中间产物中找到使用示例，在辅助产物`ComponentRegister`上使用。

二、注解处理器

位置：`compiler`模块，下面从整个环境搭建到解析全过程逐步分析。

和`annotation`模块一样，首先看一下此模块的`build.gradle`文件，

Groovy

```groovy
import org.gradle.internal.jvm.Jvm

// 纯java代码处理，只需java标准插件支持即可
apply plugin: 'java-library'

// 配置java版本
sourceCompatibility = JavaVersion.VERSION_1_8
targetCompatibility = JavaVersion.VERSION_1_8

dependencies {
  // 因为需要解析处理注解，必须依赖注解模块
  implementation project(':annotation')
  // java代码生成框架
  implementation 'com.squareup:javapoet:1.13.0'
  // google的java注解处理器META-INF自动生成服务
  compileOnly 'com.google.auto.service:auto-service:1.0-rc7'
  // 注解处理器，可以通过注解方式完成注解处理器入口类声明
  annotationProcessor 'com.google.auto.service:auto-service:1.0-rc7'
  // 引入jdk扩展工具包
  compileOnly files(Jvm.current().getToolsJar())

  // gradle增量插件支持库
  implementation 'net.ltgt.gradle.incap:incap:0.2'
  // 通过注解方式支持生成gradle的META-INF信息
  annotationProcessor 'net.ltgt.gradle.incap:incap-processor:0.2'
}

// 此库也需要上传到maven端使用
apply from: "${rootProject.file('mvn-push.gradle')}"
```

这里需要注意一处隐式依赖，`com.google.auto.service:auto-service` 也会引入

Groovy

```groovy
dependencies {
  // google开源的jdk功能扩展支持工具包，国外非常出名，不过此项目不会使用
  compileOnly 'com.google.guava:guava:version'
  // google开源的注解处理器的测试和调试框架，方便注解处理器快速开发，下面会介绍到
  testImplementation 'com.google.testing.compile:compile-testing:version'
}
```

接下来从零开始编写注解处理器，我们将入口类定义为`ComponentizationProcessor`，如下：

第一步，定义类型

Java

```java
// 需要注意注解处理相关支持在javax.annotation包下，不属于标准库内容
import javax.annotation.processing.AbstractProcessor;
// java语法解析相关支持在javax.lang包下，不属于标准库内容
import javax.lang.model.element.Element;

import com.google.auto.service.AutoService;
import javax.annotation.processing.SupportedSourceVersion;
import net.ltgt.gradle.incap.IncrementalAnnotationProcessor;

// 标记此类是一个注解处理服务，auto-service工具会生成对应注解处理器相关META-INFO信息
@AutoService(Processor.class)
// 标记gradle注解增量策略类型为dynamic，incap工具会生成gradle相关META-INFO信息
@IncrementalAnnotationProcessor(IncrementalAnnotationProcessorType.DYNAMIC)
// 请看第2步，2选1方式
// @SupportedSourceVersion(SourceVersion.RELEASE_8)
// 请看第3步，2选1方式
// @SupportedOptions({"ISOLATING", "OPTION_LOG"})
// 请看第4步，2选1方式
// @SupportedAnnotationTypes({"com.bhb.android.componentization.Service"})
public final class ComponentizationProcessor extends AbstractProcessor {

  // 唯一需要强制实现的接口，接口参数中输入相关注解元素上下文，根据源码元素生成中间类
  @Override
  public boolean process(Set<? extends TypeElement> annotations,
RoundEnvironment env) {
  }

}
```

第二步，关注语言版本`getSupportedSourceVersion`，返回java版本，与编译器检查有关，

Java

```java
@Override public SourceVersion getSupportedSourceVersion() {
  return SourceVersion.RELEASE_8;
}
```

第三步，关注编译选项`getSupportedOptions`，注解处理为预编译处理，提供了编译参数注入，此api可以被声明在Processor上注解`@SupportedOptions(String[])`代替

Java

```java
public final class ComponentizationProcessor extends AbstractProcessor {

  // 定义日志输出的编译选项
  private static final String OPTION_LOG = "LOGGABLE"

  // 返回String集合定义了可被支持的编译选项
  @Override public Set<String> getSupportedOptions() {
    ImmutableSet.Builder<String> builder = ImmutableSet.builder();
    if (trees != null) {
      // 加入对gradle增量注解支持的编译选项，这样gradle就可以输入此特性支持

builder.add(IncrementalAnnotationProcessorType.ISOLATING.getProcessorOption());
    }
    // 我们把对日志输出控制的选项提供支持
    builder.add(OPTION_LOG);
    return builder.build();
  }
}
```

第四步，指定支持注解`getSupportedAnnotationTypes`，声明需要处理关注的注解集合，此api可以被声明在Processor上注解`@SupportedAnnotationTypes(String[])`代替

Java

```java
public final class ComponentizationProcessor extends AbstractProcessor {
  @Override public Set<String> getSupportedAnnotationTypes() {
    Set<String> types = new LinkedHashSet<>();
    // 添加Api注解
    // types.add(Api.class.getCanonicalName());
    // 添加Service注解
    types.add(Service.class.getCanonicalName());
    return types;
  }
}
```

> 需要注意的是，为保证注解处理性能，我们只需要跟踪入口注解，即Service注解。
>
> 从两点考虑：1. 没有实现的Api接口无需注册。2. 从Service可以反推出所有Api关系。
>
> 所以跟踪Service这一个注解即可实现处理需求。

第五步，关注初始化方法`init`

Java

```java
public final class ComponentizationProcessor extends AbstractProcessor {

  private Types typeUtils;
  private Filer filer;
  private @Nullable Trees trees;
  private Messager logger;
  private Map<String, String> options;

  @Override
  public synchronized void init(ProcessingEnvironment env) {
    super.init(env);
    // 预编译输入的编译选项，只支持从第三步提供的选项key，并可以携带value
    options = env.getOptions();
    // 编译输出日志工具，提供了不同的输出级别
    logger = env.getMessager();
    // javax.lang.model类型工具，后面process过程会用到
    typeUtils = env.getTypeUtils();
    // 输出产物输出流工具，支持创建java源码文件，在process过程用到
    filer = env.getFiler();
    try {
      // 语法树扫描解析工具，后面process过程会用到
      trees = Trees.instance(processingEnv);
    } catch (IllegalArgumentException ignored) {
      try {
        // Get original ProcessingEnvironment from Gradle-wrapped one or KAPT-wrapped one.
        for (Field field : processingEnv.getClass().getDeclaredFields()) {
          if (field.getName().equals("delegate") || field.getName().equals("processingEnv")) {
            field.setAccessible(true);
            ProcessingEnvironment javacEnv = (ProcessingEnvironment) field.get(processingEnv);
            trees = Trees.instance(javacEnv);
            break;
          }
        }
      } catch (Throwable ignored2) {
      }
    }
  }
}
```

第六步，真正的核心处理`process`接口，

Java

```
1  public final class ComponentizationProcessor extends AbstractProcessor {
2    @Override
3    public boolean process(Set<? extends TypeElement> annotations,
   RoundEnvironment env) {
4      for (Element element : env.getElementsAnnotatedWith(Service.class)) {
5        if (!SuperficialValidation.validateElement(element)) {
6          continue;
7        }
8        try {
9          generateRegisterClassFile(element);
10       } catch (Exception e) {
11         e.printStackTrace();
12         logger.printMessage(Diagnostic.Kind.ERROR, e.getMessage());
13         return false;
14       }
15     }
16     return false;
17   }
18 }
```

关于返回Boolean值含义，下面为从java官方api参考中描述，

Processes a set of annotation types on type elements originating from the prior round and returns whether or not these annotation types are claimed by this processor. If `true` is returned, the annotation types are claimed and subsequent processors will not be asked to process them; if `false` is returned, the annotation types are unclaimed and subsequent processors may be asked to process them. A processor may always return the same boolean value or may vary the result based on chosen criteria.The input set will be empty if the processor supports `"*"` and the root elements have no annotations. A `Processor` must gracefully handle an empty set of annotations.

翻译过来的含义，

处理一个从上一轮处理流转来的类型元素上的注解类型集合，并且返回这些注解是否被此处理器认领。如果返回`true`，表示这个注解被认领，且其他子处理器不会再次请求处理；如果返回`false`，表示这个注解不被认领，子处理器可能会再次请求处理。一个处理器可能返回同样的值或者可以基于

不同处理策略返回不同值。如果处理器支持`*`通配符注解集合，并且根`Element`没有任何注解声明，则输入的注解类型集合为空。一个处理器必须优雅（合理）得处理一个空注解类型集合。

总结一下，

注解处理器设计为链式处理结构，多个处理器之间有子从关系。当一个处理器完成处理之后，可以决定是否完整处理(true)，如果需要其他处理器继续关注，则需要返回false。开发者可以根据自己的处理策略动态决定。

是不是有点像客户端的事件处理机制，自己处理还可以决定是否完全拦截，否则分发其他View。

这里返回false，每个模块处理器都需要执行。下面开始解析，也分为几部分讲，单独拿出来比较好。

三、解析和生产

第一步，遍历指定元素

Kotlin

```kotlin
// 查找所有被Service注解的元素，此注解标注到实现类上，所以得到的元素为class类型，
// 比如，可以返回LibraryService类
for (Element element : env.getElementsAnnotatedWith(Service.class)) {
  // 此工具为auto-service框架提供，用于验证元素是否合规，不知道哪些元素不正确?
  if (!SuperficialValidation.validateElement(element)) {
    continue;
  }
}
```

第二步，接下来要生成产物，首先来分析设计需求，

1. 验证Service和Api接口声明合规性。
2. 接口的延迟代理，用于AutoWired过程。
3. 对于之后的插装操作，进行尽可能的编码简化。

对于第1个需求，我们需要分析`Service`类结构，需要符合以下要求：

1. 必须是实现class，且不能是抽象的(abstract)。
2. 必须有父接口，且父接口中至少包含一个`API`接口，且接口必须被`@Api`注解修饰。

> 通过第1步结构分析中，我们可以得到多Api和Service之间的对应关系。

对于第2个需求，我们也需要借助第1个需求的多对一关系解决。对于代理类应该这样设计：

1. 代理接口和Service类必须有共同的Api接口组合，这样对外的类型系统一致。
2. 延迟初始化需求，实现类必须在某一个接口调用处再获取实例。

我们通过简单的定义使用演示一下需要的设计条件。

首先定义了2个Api接口和一个实现Service，

Java

```java
/**
 * 天使的能力
 */
@Api
public interface AngelApi: API {
  void fly(int height);
}

/**
 * 宝宝的能力
 */
@Api
public interface BabyApi: API {
  void sleep(int time);
}

/**
 * 安吉拉宝贝
 */
@Service
public class AngelBaby implements AngelApi, BabyApi {
  @Override public void fly(int height) {}
  @Override public void sleep(int time) {}
}
```

对应的代理类大概是这样，

Java

```java
class AngelBabyLazyDelegate implements AngelApi, BabyApi {

  private AngelApi angelApiService() {
    return 获取AngelBaby实例;
  }

  private BabyApi babyApiService() {
    return 获取AngelBaby实例;
  }

  @Override public void fly(int height) {
    // 调用时再获取
    angelApiService().fly(height);
  }
  @Override public void sleep(int time) {
    // 调用时再获取
    babyApiService().sleep(time);
  }

}
```

> 关于某一个Api对应Service实例的获取，可以通过第一步的分析反向查找得到。
>
> 具体考虑到实例模型等一系列处理，我们通过辅助运行时框架中的工具类方法 `Componentization.getXXX(ApiClass.class)` 得到，此工具在 `componentization` 模块，这个模块的提供了一些运行期的工具方法，更是提供了手动注册等方式。

对于第3个需求，为了最后插装自动化注册，我们生成注册辅助类，先看接口设计，此辅助类在 `componentization` 模块中可以找到：

Java

```java
/**
 * 组件注册辅助接口
 */
interface ComponentRegister {

  /**
   * 生成的代理类后缀
   */
  String SUFFIX = "_Register";

  Item register();

  class Item {
    final List<Class<? extends API>> apis;
    final Class<? extends API> service;

    Item(List<Class<? extends API>> apis, Class<? extends API> service) {
      this.apis = apis;
      this.service = service;
    }
  }

}
```

对应上面的`AngelBabyService`类，其辅助注册类大概是这样，

Java

```java
@Meta(
    service = "com.bhb.android.componentization.AngelBabyService",
    api = {"com.bhb.android.componentization.AngelApi",
"com.bhb.android.componentization.BabyApi"}
)
class AngelBabyService_Register implements ComponentRegister {
  @Override public Item register() {
    final ArrayList<Class<? extends API>> apis = new ArrayList<>(2);
    apis.add(AngelApi.class);
    apis.add(BabyApi.class);
    return new ComponentRegister.Item(apis, AngelBabyService.class);
  }
}
```

> 注意，为了保护中间产物的封闭性，我们都限制为`com.bhb.android.componentization`包内，并且可以和`Componentization`进行互操作。

第三步，生成两个类的实现，

在具体的编码实现中，1和3需求是可以合并进行的，通过源码展示，

Java

```java
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment env) {
  for (Element element : env.getElementsAnnotatedWith(Service.class)) {
    if (!SuperficialValidation.validateElement(element)) {
      continue;
    }
    try {
      // 1. 生成运行时辅助注册工具，对应第3个设计需求，同时进行检查用作第1个设计需求
      generateRegisterClassFile(element);
      // 2. 生成用于AutoWired注解延迟代理类，满足第2个设计需求
      generateLazyClassFile(element);
    } catch (Exception e) {
      e.printStackTrace();
      logger.printMessage(Diagnostic.Kind.ERROR, e.getMessage());
      return false;
    }
  }
  return false;
}
```

下面的生成Java源码都使用了JavaPoet工具，需要自行了解。

1. 生成辅助注册类，

Java

```java
/**
 * 生成注册类文件
 * @param element 元素
 * @throws IOException 写入异常
 */
private void generateRegisterClassFile(Element element) throws IOException {
  TypeSpec.Builder typeBuilder = TypeSpec.classBuilder(
          // a 指定类名为serviceName + ComponentRegister_SUFFIX。
          element.getSimpleName() + ComponentRegister_SUFFIX)
          .addModifiers(Modifier.FINAL)
          // b 添加父接口ComponentRegister。
          .addSuperinterface(ComponentRegisterType)
          // c 添加接口ComponentRegister.register实现，
          // 收集一对多关系信息返回，具体请查看接口设计
          .addMethod(buildRegisterMethod(element))
          // d 添加一对多的辅助查询注解信息，便于其他过程查询。
          .addAnnotation(buildRegisterMeta(element));
  // e 确定类包名为PACKAGE_SPACE，写入文件注释信息，结果输出给filer流写入到指定文件。
  JavaFile file = JavaFile.builder(PACKAGE_SPACE, typeBuilder.build())
          .addFileComment("此文件为自动生成，用于组件化辅助注册").build();
  file.writeTo(filer);
}
```

生成其类的过程大体一致，分为两个部分，

第一部分：定义类名，添加父接口/类，添加成员属性，添加成员方法，添加注解。

第二部分：指定包名，写入文件注释，输出为文件流。

目标类：

Java

```java
// 此文件为自动生成，用于组件化辅助注册
package com.bhb.android.componentization;

@Meta(
    service = "com.bhb.android.componentization.AngelBabyService",
    api = {"com.bhb.android.componentization.AngelApi",
"com.bhb.android.componentization.BabyApi"}
)
class AngelBabyService_Register implements ComponentRegister {
  @Override public Item register() {
    final ArrayList<Class<? extends API>> apis = new ArrayList<>(2);
    apis.add(AngelApi.class);
    apis.add(BabyApi.class);
    return new ComponentRegister.Item(apis, AngelBabyService.class);
  }
}
```

2. 生成延迟代理类

Java

```java
/**
 * 生成懒初始化代理类
 * @param element Service元素
 * @throws IOException 写入异常
 */
private void generateLazyClassFile(Element element) throws IOException {
  Type serviceType = ((Symbol.ClassSymbol) element).asType();
  // a 通过Service向上找到所有的Api接口类
  List<Type> apis = getApiTypes(element);
  // b 新建延迟代理类名 ServiceName_Lazy
  TypeSpec.Builder typeBuilder = TypeSpec.classBuilder(
          element.getSimpleName() + LazyDelegate_SUFFIX);
  TypeName apiTypeName;
  List<Type> interfaces;
  // c 遍历所有的api接口，开始添加代理接口实现
  for (Type api : apis) {
    apiTypeName = TypeName.get(api);
    // d 获取某一个api接口向上查询的其他接口
    interfaces = getAllInterfaces(api);
    // e 添加Api接口到代理类
```

```
20      // e 添加Api接口到代理类
21      typeBuilder.addSuperinterface(apiTypeName);
22      // 代理类变量
23      TypeName delegateType = ParameterizedTypeName.get(LazyDelegateType,
    apiTypeName);
24      // 定义某个代理Api实现成员名称 ApiNameDelegate
25      String fieldName = ((ClassName) getRawType(apiTypeName)).simpleName()
26              + LazyDelegate_Field_DELEGATE_SUFFIX;
27      // f 创建代理Api接口属性成员
28      FieldSpec.Builder fieldBuilder = FieldSpec.builder(
29              delegateType, fieldName)
30              .addModifiers(Modifier.PRIVATE)
31              // g 创建代理成员初始化逻辑 = new LazyDelegateImpl<>(){}
32              .initializer("new $T<$T>() {}", LazyDelegateImplType, apiTypeName);
33      // h 把api代理属性添加代理类
34      typeBuilder.addField(fieldBuilder.build());
35      // i 遍历每个Api接口的所有父接口类型
36      for (Type intf : interfaces) {
37        // j 复制每个接口方法定义并表达为JavaPoet模型,
38        // 指定方法体为ApiDelegate.get().apiName(params...)
39        for (MethodSpec method : generateMethod(serviceType, intf, fieldName)) {
40          // k 把每个接口方法添加到代理类
41          typeBuilder.addMethod(method);
42        }
43      }
44    }
45
46    // l 写入文件
47    JavaFile.builder(PACKAGE_SPACE, typeBuilder.build())
48            .addFileComment("此文件为自动生成，用于组件化延迟代理")
49            .build()
50            .writeTo(filer);
51  }
```

中间出现的几个转换细节实现需要自行通过库代码理解。

目标类：

Java

```java
// 此文件为自动生成，用于组件化延迟代理
package com.bhb.android.componentization;


class AngelBabyService_Lazy implements AngelApi, BabyApi {
  private LazyDelegate<LibraryAPI> AngelApiDelegate = new LazyDelegateImpl<AngelApi>() {};
  private LazyDelegate<BabyApi> BabyApiDelegate = new LazyDelegateImpl<BabyApi>() {};

  @Override
  public void fly(int arg0) {
    // Element中无法获取到方法参数名称，故采用arg+索引的方式解决
    AngelApiDelegate.get().fly(arg0);
  }

  @Override
  public void sleep(int arg0) {
    AngelApiDelegate.get().sleep(arg0);
  }

}
```

其中的核心辅助类LazyDelegateImpl，get方法实现采用单实例按需创建方式，一看即明白，

Java

```java
package com.bhb.android.componentization;

import java.lang.reflect.ParameterizedType;

/**
 * 这样的空构造是为了反射到实际类型参数，具体使用的地方构造一个匿名类来保留类型信息
 * @param <C> API类型
 */
abstract class LazyDelegateImpl<C extends API> implements LazyDelegate<C> {

  /**
   * 组件实例
   */
  private volatile C api;

  @SuppressWarnings("unchecked")
  private Class<C> getAPIClass() {
    return (Class<C>) ((ParameterizedType) getClass()
        .getGenericSuperclass()).getActualTypeArguments()[0];
  }

  @Override
  public C create() {
    return Componentization.getSafely(getAPIClass());
  }

  @Override
  public synchronized C get() {
    if (null == api) {
      api = create();
    }
    return api;
  }
}
```

四、辅助框架

位置：`componetization`模块

作用：辅助运行期注册和查询，上面所有的注解处理或者其gradle插件都会关联到此框架，此框架作为开发者的唯一操作入口。

核心：`Componentization`工具类，简化了一些实现过程，其大体代码结构如下，

Java

```java
public final class Componentization {

  /**
   * 全包名
   */
  private static final String PACKAGE =
Componentization.class.getPackage().getName();

  /**
   * 收集到的组件注册信息
   */
  private final static Map<Class<? extends API>, Class<? extends API>>
          sComponentProvider = new HashMap<>();
  /**
   * 单例组件存储
   */
  private final static Map<Class<? extends API>, API> sComponents = new
HashMap<>();

  /**
   * 手动注册
   * @param api     api
   * @param service service实例
   */
  public static void register(Class<? extends API> api, Class<? extends API>
service) {
    sComponentProvider.put(api, service);
  }

  /**
   * 手动注册
   * @param api     api
   * @param service service实例
   */
  public static void register(Class<? extends API> api, API service) {
    sComponents.put(api, service);
  }

  /**
   * 执行指定组件器
   */
```

```
     */
    private static void register(Class<? extends ComponentRegister> register) {
        try {
            ComponentRegister.Item registerItem = register.newInstance().register();
            for (Class<? extends API> api : registerItem.apis) {
                sComponentProvider.put(api, registerItem.service);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 尝试获取指定api实现
     * @param type api接口
     * @param <T>  类型
     * @return     api实现：必须被AService注解修饰
     */
    public static <T extends API> T getSafely(Class<T> type) {
        try {
            return get(type);
        } catch (ComponentException e) {
            e.printStackTrace();
        }
        return null;
    }

    /**
     * 尝试获取指定api实现
     * @param type api接口
     * @param <T>  类型
     * @return     api实现：必须被AService注解修饰
     * @throws ComponentException 相关异常
     */
    @SuppressWarnings("unchecked")
    public static <T extends API> T get(Class<T> type) throws ComponentException {

    }

    /**
     * 尝试获取指定api延迟初始化实现
     * @param apiType api接口
     * @param <T>  类型
     * @return     api实现：必须被AService注解修饰
     */
```

```
83    @SuppressWarnings("unchecked")
84    public static <T extends API> T getLazy(Class<T> apiType) throws
      ComponentException {
85        Class<T> type = apiType;
86        try {
87            type = null != type.getAnnotation(Service.class)
88                    ? type : (Class<T>) sComponentProvider.get(type);
89            Class<? extends LazyDelegate<T>> lazyClazz
90                    = loadClass(PACKAGE + "." + type.getSimpleName() +
      LazyDelegate.SUFFIX);
91            return (T) lazyClazz.newInstance();
92        } catch (Exception e) {
93            e.printStackTrace();
94            throw new ComponentException("组件[" + apiType.getCanonicalName() + "]无法
      支持延迟初始化特性");
95        }
96    }
97
98    /**
99     * 尝试获取指定api延迟初始化实现，自动向下降低为非延迟初始化
100    * @param type api接口
101    * @param <T>  类型
102    * @return     api实现：必须被AService注解修饰
103    */
104   public static <T extends API> T getLazySafely(Class<T> type) {
105       try {
106           return getLazy(type);
107       } catch (ComponentException e) {
108           e.printStackTrace();
109           Log.e(TAG, Log.getStackTraceString(e));
110           return getSafely(type);
111       }
112   }
113
114   @SuppressWarnings("unchecked")
115   private static <T extends API> T makeInstance(Class<T> service, boolean
      singleton)
116           throws ComponentException {
117
118   }
119
120   @SuppressWarnings("unchecked")
121   private static <T extends API> T newInstance(Class<T> service) throws
      ComponentException {
122       try {
```

```
123         Constructor<? extends API> constructor = service.getDeclaredConstructor();
124         constructor.setAccessible(true);
125         return (T) constructor.newInstance();
126       } catch (Exception e) {
127         e.printStackTrace();
128       }
129       throw new ComponentException("对于Service类" + service.getName() + "而言，没有找到合适的实例构造器");
130     }
131
132 }
```

可以通过此工具完成手动注册和使用，

Java

```java
// 手动注册Api ---> Service关系
Componentization.register(AngelApi.class, AngelBabyService.class);
Componentization.register(BabyApi.class, AngelBabyService.class);
// 获取接口实现，非延迟初始化方式，即获取到真实的Service实例。
AngelApi angel = Componentization.get(AngelApi.class);
// 获取接口实现，延迟初始化方式，即获取到接口的延迟代理类。
BabyApi baby = Componentization.getLazy(BabyApi.class);
// 当调用其中的接口时获取Service实例。
baby.sleep(0);
```

关于接口自动化注册内容请查阅gradle插件文档： 详解Componentization库插件