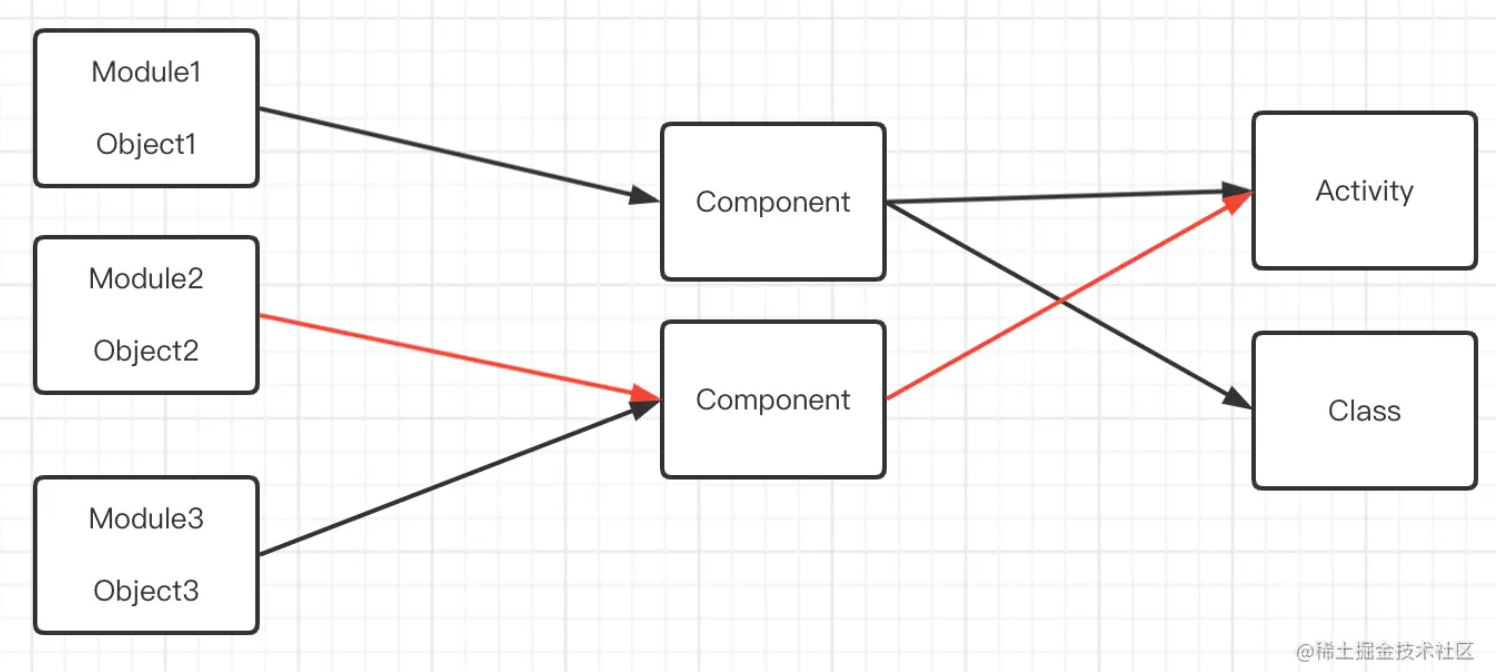
对于dagger2来说，属于移动端的依赖注入框架，在Hilt之前可以说使用最多的一个框架，其主要核心思想可看下图



我们创建了多个实体对象类，我们想将其注入到Activity或者一些类中，传统的方式通过new关键字创建，会导致实现层与调用层完全耦合在一起；因此dagger2就是在调用层与实现层之间加了一层Component，用于提供注入的类对象。

dagger2三板斧

dagger2的几个重要组件：Module、Component

@Module

class HttpModule {

/\*\*提供数据支持\*/

@Provides

fun provideHttpObject(): HttpObject {

return HttpObject()

}

}

@Component(modules = arrayOf(HttpModule::class))

interface HttpComponent {

/\*\*标记这个Component要注入到哪个组件上\*/

fun injectSplashActivity(activity: AppCompatActivity)

}

对于Component，在使用的时候需要注意两点，首先它一定得是一个接口，另外在注入的时候，一定得是一个具体的Activity，不能像上面写的那样是一个笼统的Activity。

APT生成代码讲解 - create

根据Component生成的一个类，这个类实现了HttpComponent接口

public final class DaggerHttpComponent implements HttpComponent {

private Provider<HttpObject> provideHttpObjectProvider;

private MembersInjector<SplashActivity> splashActivityMembersInjector;

private DaggerHttpComponent(Builder builder) {

assert builder != null;

initialize(builder);

}

public static Builder builder() {

return new Builder();

}

//将HttpComponent实现类初始化

public static HttpComponent create() {

return builder().build();

}

@SuppressWarnings("unchecked")

private void initialize(final Builder builder) {

this.provideHttpObjectProvider = HttpModule\_ProvideHttpObjectFactory.create(builder.httpModule);

this.splashActivityMembersInjector =

SplashActivity\_MembersInjector.create(provideHttpObjectProvider);

}

@Override

public void injectSplashActivity(SplashActivity activity) {

splashActivityMembersInjector.injectMembers(activity);

}

public static final class Builder {

private HttpModule httpModule;

private Builder() {}

public HttpComponent build() {

if (httpModule == null) {

this.httpModule = new HttpModule();

}

return new DaggerHttpComponent(this);

}

public Builder httpModule(HttpModule httpModule) {

this.httpModule = Preconditions.checkNotNull(httpModule);

return this;

}

}

}

首先我们在使用的时候，是通过下面这种方式使用的，首先我们看下create方法

DaggerHttpComponent.create().injectSplashActivity(this)

在这个方法中，是调用了内部类Builder的build方法，在这个方法中，首先我们看下会判断httpModule是否为空，如果为空，那么就首先初始化HttpModule，然后创建一个DaggerHttpComponent对象。

public HttpComponent build() {

if (httpModule == null) {

this.httpModule = new HttpModule();

}

return new DaggerHttpComponent(this);

}

也就是说，调用DaggerHttpComponent.create()这段代码就是做一些初始化操作：

（1）如果Component中注册的Module没有被初始化，那么首先会在build方法中进行初始化；

（2）创建Component接口实现类对象。

APT生成代码讲解 - inject

接下来我们看下核心方法inject是如何实现对象注入的；

private DaggerHttpComponent(Builder builder) {

assert builder != null;

initialize(builder);

}

当创建DaggerHttpComponent对象的时候，我们可以看到，在DaggerHttpComponent的构造方法中，有一个initialize方法

private void initialize(final Builder builder) {

this.provideHttpObjectProvider = HttpModule\_ProvideHttpObjectFactory.create(builder.httpModule);

this.splashActivityMembersInjector =

SplashActivity\_MembersInjector.create(provideHttpObjectProvider);

}

在这个方法中，我们可以看到分两步走：

（1）首先是创建一个HttpModule\_ProvideHttpObjectFactory工厂类对象，传入了在Builder中初始化的Module对象。

public final class HttpModule\_ProvideHttpObjectFactory implements Factory<HttpObject> {

private final HttpModule module;

public HttpModule\_ProvideHttpObjectFactory(HttpModule module) {

assert module != null;

this.module = module;

}

/\*\*这里返回的就是Module中定义的Provides方法\*/

@Override

public HttpObject get() {

return Preconditions.checkNotNull(

module.provideHttpObject(), "Cannot return null from a non-@Nullable @Provides method");

}

public static Factory<HttpObject> create(HttpModule module) {

return new HttpModule\_ProvideHttpObjectFactory(module);

}

}

我们重点关注一下get方法，这个方法返回的就是在Module中定义的Provides方法，其实就是实例化某个对象。

（2）第二步就是创建SplashActivity\_MembersInjector，并把工厂实现类传进去了。所以在调用injectSplashActivity方法的时候，其实就是调用了SplashActivity\_MembersInjector的injectMembers方法，在这个方法中，将会拿到在SplashActivity中的变量，并给赋值（这里就是调用了HttpModule\_ProvideHttpObjectFactory类中的get方法）

public final class SplashActivity\_MembersInjector implements MembersInjector<SplashActivity> {

private final Provider<HttpObject> httpObjectProvider;

public SplashActivity\_MembersInjector(Provider<HttpObject> httpObjectProvider) {

assert httpObjectProvider != null;

this.httpObjectProvider = httpObjectProvider;

}

public static MembersInjector<SplashActivity> create(Provider<HttpObject> httpObjectProvider) {

return new SplashActivity\_MembersInjector(httpObjectProvider);

}

@Override

public void injectMembers(SplashActivity instance) {

if (instance == null) {

throw new NullPointerException("Cannot inject members into a null reference");

}

instance.httpObject = httpObjectProvider.get();

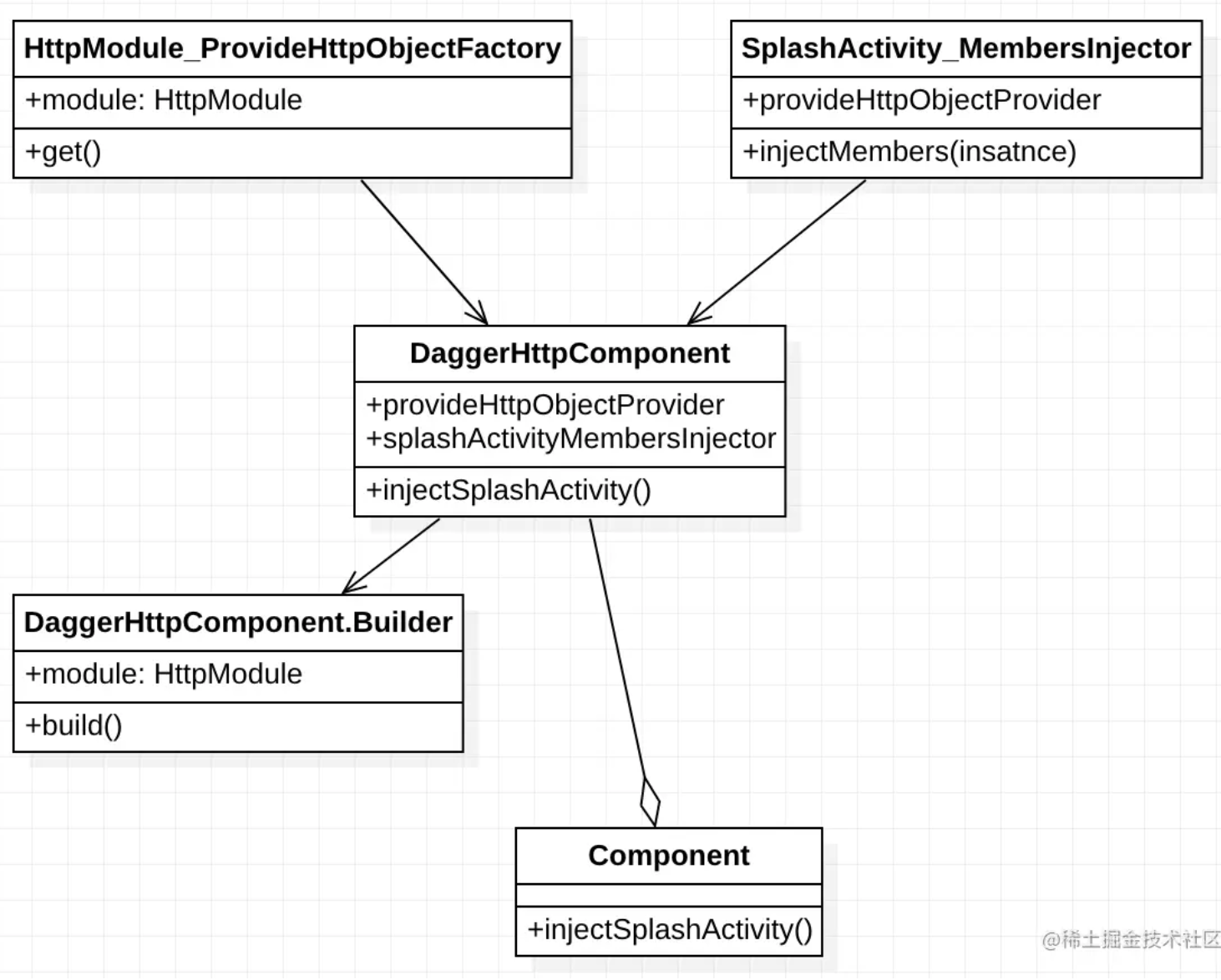
}

}

所以为什么在注入的时候，不能设置为Activity而需要一个具体的类，就是因为在赋值的时候，需要精确到这个类中具体成员变量，而且不能为私有变量。

小结

所以从dagge2的源码我们大概能够明白了，其实APT生成了几个关键类，可以看下面的类图



在DaggerHttpComponent中会持有两个核心类引用，分别是持有Module的Factory类，以及实现注入的memberInjector，我们看下面的用法：

@Inject

@JvmField

var httpObject: HttpObject? = null

override fun initView() {

JUCTest.test()

Singleton.getInstance().increment()

testProxy()

DaggerHttpComponent.create().injectSplashActivity(this)

}

当创建DaggerHttpComponent的时候，其实Module还有Factory、MemberInjector都已经完成了初始化，当调用injectSplashActivity方法的时候，其实就是将httpObject与Module中的provideHttpObject方法的返回值做了关联。

@Inject

@JvmField

var httpObject: HttpObject? = null

=

httpObject = httpObjectProvider.get()

=

httpObject = module.provideHttpObject()

=

httpObject = HttpObject()

dagger2实现单例模式

因为在Module中，提供对象的方式是通过新建一个对象

@Inject

@JvmField

var httpObject: HttpObject? = null

@Inject

@JvmField

var httpObject2: HttpObject? = null

所以如果是上述这种方式，那么通过源码我们其实能知道，这两个对象都是新创建一个HttpObject对象，其实我们对于这样的场景肯定是希望是一个单例，那么在dagge2中也提供了这种单例的注解@Singleton。

@Singleton

@Module

class HttpModule {

/\*\*提供数据支持\*/

@Provides

@Singleton

fun provideHttpObject(): HttpObject {

return HttpObject()

}

}

****这里需要注意，如果要设置成单例，那么在Module和Component上也需要声明单例注解@Singleton，否则编译会报错。****

****局部单例****

通过@Singleton配置之后，确实如我们所见，这个对象已经是一个单例了

****E/TAG: httpObject==>37015167 httpObject2==>37015167****

那么当页面跳转之后，我们在另一个页面上，会发现这个对象又是一个新建的对象了。

****2022-11-27 17:37:07.463 8568-8568/com.lay.mvi E/TAG: httpObject==>37015167 httpObject2==>37015167****

****2022-11-27 17:37:12.112 8568-8568/com.lay.mvi E/TAG: 调用前处理--****

****2022-11-27 17:37:12.473 8568-8568/com.lay.mvi E/TAG: httpObject==>193636890****

所以dagger2中的@Singleton只是一个局部单例，它的作用域只是在当前页面上是单例的，那么如何做到全局的单例呢？

全局单例

为什么是局部的单例，我们应该也知道，****因为每个页面都是会创建新的Component，所以对于Module和Component来说，每个页面都是不一样的****，所以想要做到全局的单例，就需要将作用域提到全局的位置。

companion object{

var component: HttpComponent? = null

}

override fun onCreate() {

super.onCreate()

Log.e("TAG", "Application onCreate")

component = DaggerHttpComponent.create()

}

这样Component的作用域就提到了App的层，所有页面只要拿到这个Component都会是一个单例的形式。

MyApp.component?.injectMainActivity2(this)

那么单例是如何实现的呢？通过源码可以发现，跟之前不同的是，多了一个DoubleCheck.provider

@SuppressWarnings("unchecked")

private void initialize(final Builder builder) {

this.provideHttpObjectProvider =

DoubleCheck.provider(HttpModule\_ProvideHttpObjectFactory.create(builder.httpModule));

this.splashActivityMembersInjector =

SplashActivity\_MembersInjector.create(provideHttpObjectProvider);

this.mainActivity2MembersInjector =

MainActivity2\_MembersInjector.create(provideHttpObjectProvider);

}

这个DoubleCheck.provider其实就是****实现了HttpModule\_ProvideHttpObjectFactory的单例，这样在注入给变量赋值的时候，其实拿到的就是同一个Factory中的同一个Module****（也是一个单例）。

多Component注入

假设我们现在又有一个模块想要实现注入，而且也需要注入到同一个Activity中，那么可以新建一个Module和Component。

@Component(modules = arrayOf(ImageModule::class))

interface ImageComponent {

fun inject(activity: SplashActivity)

}

这里我们写了一个Component，同样也是要注入到SplashActivity中，之前的HttpComponent也是注入到SplashActivity中，但是在编译的时候报错了

com.lay.mvi.dagger.HttpObject cannot be provided without an @Inject constructor or from an @Provides- or @Produces-annotated method.

public abstract void inject(@org.jetbrains.annotations.NotNull()

^

com.lay.mvi.dagger.HttpObject is injected at

com.lay.mvi.SplashActivity.httpObject

com.lay.mvi.SplashActivity is injected at

com.lay.mvi.dagger.ImageComponent.inject(activity)

报这种错误就是对于同一个Activity或者Fragment，只能注入一次。

那么如果想在一个Activity中注入多个Component，该怎么办呢，就得用Component的依赖了。

@Component(modules = arrayOf(ImageModule::class))

interface ImageComponent {

fun provideImage(): ImageObject

}

首先，在ImageComponent中不再进行注入的处理，而是提供一个ImageObject对象获取方法。

@Singleton

@Component(modules = arrayOf(HttpModule::class), dependencies = arrayOf(ImageComponent::class))

interface HttpComponent {

/\*\*标记这个Component要注入到哪个组件上\*/

fun injectSplashActivity(activity: SplashActivity)

fun injectMainActivity2(activity:MainActivity2)

}

在之前的组件中，设置dependencies属性，将ImageComponent作为HttpComponent的依赖传入进去，那么这个时候，再创建DaggerHttpComponent的时候，就不能简单地create，而是需要把所有的依赖项手动配置进去

component = DaggerHttpComponent.builder()

.imageComponent(DaggerImageComponent.create())

.httpModule(HttpModule())

.build()

这样我们在项目中就可以使用ImageComponent中定义的Module注入了。像我们之前写的ImageComponent并不是单例，我们加上@Singleton注解之后，发现报错了。

@Singleton

@Component(modules = arrayOf(ImageModule::class))

interface ImageComponent {

fun provideImage(): ImageObject

}

Scope作用域规则

This @Singleton component cannot depend on scoped components:

@dagger.Component(modules = {com.lay.mvi.dagger.HttpModule.class}, dependencies = {com.lay.mvi.dagger.ImageComponent.class})

^

@Singleton com.lay.mvi.dagger.ImageComponent

我们看下上面的这个错误，会发现错误是@Singleton不能在ImageComponent上使用，那这块就跟Scope的使用有关系了。

其实我们在使用Scope的时候，需要关注两点：

（1）在Component依赖的时候，两个Component上的作用域不能一样。@Singleton就是一个scope，而HttpComponent和ImageComponent上都加了这个描述，所以就会报错。

（2）没有scope的Component不能依赖一个有scope的Component；就比如ImageComponent上有@Singleton注解，而HttpComponent上没有，那么ImageComponent就不能作为HttpComponent的依赖。

那这种情况怎么处理呢？其实很简单，照猫画虎，根据@Singleton自定义作用域即可。

@Scope

@Documented

@Retention(RUNTIME)

public @interface AppScope {}

@Scope

@Documented

@Retention(RUNTIME)

public @interface UserScope {}

这里是定义了两个scope，其中AppScope可以代表全局的scope，他们都可以代替@Singleton从而实现单例模式。

@AppScope

@Component(modules = arrayOf(HttpModule::class), dependencies = arrayOf(ImageComponent::class))

interface HttpComponent {

/\*\*标记这个Component要注入到哪个组件上\*/

fun injectSplashActivity(activity: SplashActivity)

fun injectMainActivity2(activity:MainActivity2)

}

@UserScope

@Component(modules = arrayOf(ImageModule::class))

interface ImageComponent {

fun provideImage(): ImageObject

}

SubComponent

其实上面的这种写法，感觉并不是很清晰，还需要配置各种依赖关系，那么有没有相应的组件能够清晰的划分主次，当然有了，就是SubComponent。

在上面的实例中，ImageComponent就是子组件，那么就可以这么写

@UserScope

@Subcomponent(modules = arrayOf(ImageModule::class))

interface ImageComponent {

fun injectSplashActivity(activity: SplashActivity)

fun injectMainActivity2(activity: MainActivity2)

}

注意这里是反的，之前dependencies的那种方式是主组件写注入。

@AppScope

@Component(modules = arrayOf(HttpModule::class))

interface HttpComponent {

/\*\*标记这个Component要注入到哪个组件上\*/

fun getImageComponent(): ImageComponent

}

那么主组件中，就是用来拿到这些子组件，而且也非常清晰了，在主组件中能够清晰的看到有哪些子组件。

DaggerHttpComponent.create().getImageComponent().injectSplashActivity(this)

最终通过子组件的注入方式来完成。

dagger2热门注解使用

Named

例如当前有一个场景，使用HttpModule来提供一些用户信息，这里提供了2个信息，分别是小明和小王

@AppScope

@Module

class HttpModule {

/\*\*提供数据支持\*/

@Provides

fun providerUser1():User{

return User("小明",15)

}

@Provides

fun providerUser2():User{

return User("小王",25)

}

}

在实现依赖注入的时候，比如user1想要拿到小明的用户信息，user2想要拿到小王的用户信息，那么在dagger2内部想要做这样的分配，显然当前HttpModule中的做法是不够的。

@Inject

@JvmField

var user1: User? = null

@Inject

@JvmField

var user2: User? = null

这里其实就可以使用@Named("key1")限定操作符来实现一一映射关系

@AppScope

@Module

class HttpModule {

/\*\*提供数据支持\*/

@Provides

@Named("key1")

fun providerUser1():User{

return User("小明",15)

}

@Provides

@Named("key2")

fun providerUser2():User{

return User("小王",25)

}

}

这样在内部进行对象分配的时候，就能够找到对应关系。

@NamedUser("key1")

@Inject

@JvmField

var user1: User? = null

@NamedUser2("key2")

@Inject

@JvmField

var user2: User? = null

但是实际的用户场景中，这种情况肯定是少数，因为涉及到多个对象通常会使用列表，而不是一个一个地去做映射。

4.2 懒加载 Lazy Provider

@Inject

@JvmField

var lazy:Lazy<HttpObject>? = null

@Inject

@JvmField

var provider:Provider<HttpObject>? = null

像在前面，我们在注入一个对象的时候，都是直接进行赋值，在dagger2中提供了2个懒加载的泛型操作符Lazy和Provider

@Override

public void injectMembers(SplashActivity instance) {

if (instance == null) {

throw new NullPointerException("Cannot inject members into a null reference");

}

instance.httpObject = httpObjectAndHttpObject2AndProviderAndLazyProvider.get();

instance.httpObject2 = httpObjectAndHttpObject2AndProviderAndLazyProvider.get();

instance.imageObject = imageObjectProvider.get();

instance.lazy = DoubleCheck.lazy(httpObjectAndHttpObject2AndProviderAndLazyProvider);

instance.provider = httpObjectAndHttpObject2AndProviderAndLazyProvider;

}

从源码中我们可以看出，在对lazy进行赋值的时候使用了DoubleCheck，说明lazy是一个单例，而何为懒加载呢？其实就是在调用get方法的时候，才会取到值。

Log.e("TAG", "lazy==>${lazy.hashCode()} ${lazy?.get()}")

Log.e("TAG", "provider==>${provider.hashCode()} ${provider?.get()}")