

# 第6章 Java反射类加载与动态代理面试题汇总

## 第6章 Java反射类加载与动态代理面试题汇总

### 6.1 PathClassLoader与DexClassLoader的区别是什么？

详细讲解

这道题想考察什么？

考察的知识点

考生如何回答

ClassLoader简介

DexClassLoader与PathClassLoader

总结

### 6.2 什么是双亲委托机制，为什么需要双亲委托机制？

详细讲解

这道题想考察什么？

考察的知识点

考生如何回答

双亲委托机制

双亲委托机制的作用

### 6.3 Android中加载类的方法有哪些？有什么区别？

详细讲解

这道题想考察什么？

考察的知识点

考生应该如何回答

类加载时机

### 6.4 ClassNotFoundException的有可能的原因是什么？（阿里）

这道题想考察什么？

考察的知识点

考生应该如何回答

### 6.5 odex了解吗？解释型和编译型有什么区别？

详细讲解

这道题想考察什么？

考察的知识点

考生应该如何回答

odex

解释型语言和编译型语言

### 6.6 说说反射的应用场景，哪些框架？

详细讲解

这道题想考察什么？

考察的知识点

考生应该如何回答

### 6.7 反射为什么慢？(滴滴)

这道题想考察什么？

考察的知识点

考生如何回答

### 6.8 动态代理是什么？如何实现？

详细讲解

这道题想考察什么？

考察的知识点

考生如何回答

静态代理

动态代理

Java动态代理原理

### 6.9 动态代理的方法怎么初始化的？（字节跳动）

详细讲解

这道题想考察什么？

考察的知识点

考生应该如何回答

#### 6.10 CGLIB动态代理（字节跳动）

这道题想考察什么？

考察的知识点

考生如何回答

CGLIB动态代理原理

总结

#### 6.11 为什么IO是耗时操作？

详细讲解

这道题想考察什么？

考察的知识点

考生应该如何回答

## 6.1 PathClassLoader与DexClassLoader的区别是什么？

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Android虚拟机类加载机制》

### 这道题想考察什么？

Android中类加载机制的原理

### 考察的知识点

ClassLoader类加载机制

### 考生如何回答

ClassLoader就是我们常说的类加载器。

### ClassLoader简介

任何一个 Java 程序都是由一个或多个 class 文件组成，在程序运行时，需要将 class 文件加载到 JVM 中才可以使用，负责加载这些 class 文件的就是 Java 的类加载机制。ClassLoader 的作用简单来说就是加载 class 文件，提供给程序运行时使用。每个 Class 对象的内部都有一个 classLoader 字段来标识自己是由哪个 ClassLoader 加载的。

```
class Class<T> {  
    ...  
    private transient ClassLoader classLoader;  
    ...  
}
```

ClassLoader是一个抽象类，而它的具体实现类很多，最为主要被使用的有：

- `BootClassLoader`

用于加载Android Framework层class文件。

- `PathClassLoader`

用于Android应用程序类加载器。可以加载指定的dex，以及jar、zip、apk中的classes.dex

- `DexClassLoader`

用于加载指定的dex，以及jar、zip、apk中的classes.dex

- `InMemoryDexClassLoader`

Android8.0新增，用于加载内存中的dex

## DexClassLoader与PathClassLoader

在Android 5.0以下的版本中，两者之间的区别为：

1. `DexClassLoader`：可加载jar、apk和dex，可以从SD卡中加载
2. `PathClassLoader`：只能加载已安装到系统中（即/data/app目录下）的apk文件

但是随着Android版本的升级，到Android 5.0及以后就已经不是这样了。

我们先来看看在Android 中的 `PathClassLoader` 与 `DexClassLoader`：

```
public class DexClassLoader extends BaseDexClassLoader {

    public DexClassLoader(String dexPath, String optimizedDirectory,
        String libraryPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), libraryPath, parent);
    }
}

public class PathClassLoader extends BaseDexClassLoader {

    public PathClassLoader(String dexPath, ClassLoader parent) {
        super(dexPath, null, null, parent);
    }

    public PathClassLoader(String dexPath, String libraryPath,
        ClassLoader parent) {
        super(dexPath, null, libraryPath, parent);
    }
}
```

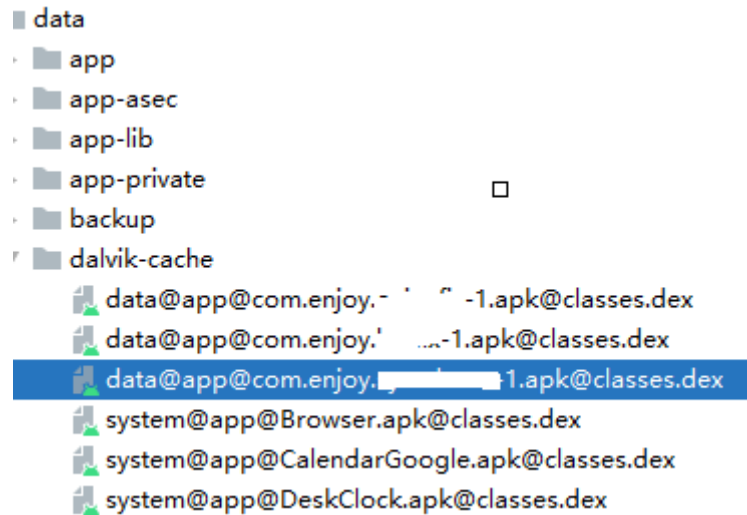
其中 `PathClassLoader` 与 `DexClassLoader` 都是继承自同一个父类：`BaseDexClassLoader`，而两者的区别则是 `DexClassLoader` 必须传递一个 **optimizedDirectory** 用于存放dexopt的结果，后者则不用。

dexopt:

在**Dalvik**中虚拟机在加载一个dex文件时，对 dex 文件 进行 验证 和 优化的操作，其对 dex 文件的优化结果变成了 odex(Optimized dex) 文件，这个文件使用了一些优化操作码，和 dex 文件很像。

其实无论是 `PathClassLoader` 还是 `DexClassLoader`，大家可以看到并没有重写父类的其他方法。如果 **optimizedDirectory** 优化目录为Null，即 `PathClassLoader`，则Android5.0以下，则会使用默认的优化目录：`/data/dalvik-cache/`。

而这个目录在安装某个APK时，系统会自动在其中存放odex文件：[data@app@packagename.apk@classes.dex](#)



而在使用 `PathClassLoader` 加载时，如果加载的不是已经安装在手机中的APK，则会报出：**Dex cache directory isn't writable: /data/dalvik-cache**，这个目录我们的应用自身并不具备写的权限。因此 `PathClassLoader` 只能加载已经安装的APK中的dex文件。

而到了ART下，加载方式发生了截然不同的变化，在安装时对 dex 文件执行dex2oat（AOT 提前编译操作），编译为OAT（实际上是ELF文件）可执行文件（机器码）。而如果在加载时，无法成功加载oat文件，仍然会尝试从原dex中加载，因此ART下，`PathClassLoader` 与 `DexClassLoader` 都能加载任意指定的dex，以及jar、zip、apk中的classes.dex。但是从原dex加载会导致无法dex2oat，加快加载速度，降低运行效率。

到了Android N之后采用解释，AOT与JIT 混合模式。

到了Android 8.1及以后，`DexClassLoader` 变为：

```
public class DexClassLoader extends BaseDexClassLoader {
35     /**
36      * Creates a {@code DexClassLoader} that finds interpreted and native
37      * code. Interpreted classes are found in a set of DEX files contained
38      * in Jar or APK files.
39      *
40      * <p>The path lists are separated using the character specified by the
41      * {@code path.separator} system property, which defaults to {@code :}.
42      *
43      * @param dexPath the list of jar/apk files containing classes and
44      *     resources, delimited by {@code File.pathSeparator}, which
45      *     defaults to {@code ":"} on Android
46      * @param optimizedDirectory this parameter is deprecated and has no
47      *     effect
48      * @param librarySearchPath the list of directories containing native
49      *     libraries, delimited by {@code File.pathSeparator}; may be
50      *     {@code null}
51      * @param parent the parent class loader
52      */
53     public DexClassLoader(String dexPath, String optimizedDirectory,
54                          String librarySearchPath, ClassLoader parent) {
55         super(dexPath, null, librarySearchPath, parent);
56     }
57 }
```

此时DexClassLoader中optimizedDirectory同样固定传递null，因此两者没有任何区别了。

## 总结

- Android 4.4及以下：
  - DexClassLoader：可加载jar、apk和dex，可以从SD卡中加载
  - PathClassLoader：只能加载已安装到系统中（即/data/app目录下）的apk文件
- Android 5.0~Android 8.0：
  - DexClassLoader：可加载jar、apk和dex，可以从SD卡中加载
  - PathClassLoader：可加载jar、apk和dex，可以从SD卡中加载，但会导致无法进行dex2oat操作
- Android 8.1及以上：
  - DexClassLoader 与 PathClassLoader 完全一致

---

## 6.2 什么是双亲委托机制，为什么需要双亲委托机制？

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Android虚拟机类加载机制》

### 这道题想考察什么？

类加载机制原理

### 考察的知识点

类加载机制

### 考生如何回答

#### 双亲委托机制

双亲委托机制是指当一个类加载器收到一个类加载请求时，该类加载器首先会把请求委派给父类加载器。每个类加载器都是如此，只有在父类加载器在自己的搜索范围内找不到指定类时，子类加载器才会尝试自己去加载。

```
public abstract class ClassLoader{
    //父类加载器
    ClassLoader parent;

    protected ClassLoader(ClassLoader parentLoader) {
        this(parentLoader, false);
    }

    ClassLoader(ClassLoader parentLoader, boolean nullAllowed) {
        if (parentLoader == null && !nullAllowed) {
            throw new NullPointerException("parentLoader == null && !nullAllowed");
        }
        parent = parentLoader;
    }
}
```

```

protected Class<?> loadClass(String className, boolean resolve) throws
ClassNotFoundException {
    //先找缓存
    Class<?> clazz = findLoadedClass(className);
    if (clazz == null) {
        ClassNotFoundException suppressed = null;
        if(parent !=null){
            try {
                //交给父类加载器加载
                clazz = parent.loadClass(className, false);
            } catch (ClassNotFoundException e) {
                suppressed = e;
            }
        }
        if (clazz == null) {
            try {
                //父类加载器加载不到，自己加载
                clazz = findClass(className);
            } catch (ClassNotFoundException e) {
                e.addSuppressed(suppressed);
                throw e;
            }
        }
    }

    return clazz;
}

```

## 双亲委托机制的作用

- 1、防止重复加载
- 2、安全，保证系统类不能被篡改。

Java虚拟机只会在不同的类的类名相同且加载该类的加载器均相同的情况下才会判定这是一个类。如果没有双亲委派机制，同一个类可能就会被多个类加载器加载，如此类就可能会被识别为两个不同的类，相互赋值时问题就会出现。

双亲委派机制能够保证多加载器加载某个类时，最终都是由一个加载器加载，确保最终加载结果相同。

没有双亲委派模型，让所有类加载器自行加载的话，假如用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，系统就会出现多个不同的Object类，Java类型体系中基础行为就无法保证，应用程序就会变得一片混乱。

## 6.3 Android中加载类的方法有哪些？有什么区别？

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Android虚拟机类加载机制》

## 这道题想考察什么？

类加载机制中类加载的过程

### 考察的知识点

1. 类加载过程
2. 类加载时机

### 考生应该如何回答

Android加载类的方式实际上就是Java的类加载。虚拟机把描述类的信息从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终才能够变成可以被虚拟机直接使用的 Java 类型。类加载过程见《5.1 描述JVM类加载过程》。

### 类加载时机

在以下情况下，类会自动加载：

1. 使用 new 实例化对象，创建子类的实例会首先加载其父类
2. 访问类的静态方法
3. 访问类的静态属性
4. 对类进行反射调用
5. Java程序中定义了main的类，在启动main方法时该类会被加载

以上五种情况都会触发类的加载并且完成对类的初始化。除了以上情况之外，我们也可以主动调用

`ClassLoader#loadClass(name)` 或者 `Class.forName(name)` 进行加载。实际上

`Class.forName(name)` 也是通过ClassLoader完成对指定类的加载。

```
public static Class<?> forName(String className) throws ClassNotFoundException {  
    //得到调用者的类,如main方法所在类  
    Class<?> caller = Reflection.getCallerClass();  
    //ClassLoader.getClassLoader(caller): 获得main方法所在类的类加载器，使用其完成  
    className的加载  
    return forName0(className, true, ClassLoader.getClassLoader(caller),  
        caller);  
}
```

但是 `ClassLoader#loadClass(name)` 与 `Class.forName(name)` 不同的是，后者除了完成类加载之外，还会对类进行初始化，执行类中的static块。而 `ClassLoader#loadClass` 只会完成类的加载。

当然，我们也可以使用：`Class.forName(String name, boolean initialize, ClassLoader loader)`。通过第二个参数initialize，从而选择是否需要对类进行初始化。

## 6.4 ClassNotFoundException的有可能的原因是什么？（阿里）

### 这道题想考察什么？

类加载流程

## 考察的知识点

1. 反射机制
2. 反射在框架中的应用
3. 获取Class类的主要方式

## 考生应该如何回答

ClassNotFoundException表示虚拟机加载指定的类时，如果没有找到对应的Class文件时，抛出的异常。以Android为例，出现此异常的原因可能有：

- 1、APK中dex未包含需要加载的类数据，导致类加载器无法加载；
- 2、Android Dalvik下，使用dex分包，在Secondary DEX 加载之前，需要使用不在主dex中的类
- 3、类数据不合法验证失败导致加载失败。此时能够在Log中看到：

```
Failure to verify dex file '/data/app/xxxxx/base.apk'
```

## 6.5 odex了解吗？解释型和编译型有什么区别？

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Android虚拟机类加载机制》

### 这道题想考察什么？

是否了解Android 类处理过程，有编程语言的基础认识

## 考察的知识点

1. APK安装过程
2. 编程语言的分类

## 考生应该如何回答

### odex

odex全称：OptimizedDEX，表示经过优化的dex文件。

Android APK就是一个Zip压缩包，Android 虚拟机每次需要从Zip压缩中读取classes.dex文件完成类加载。而odex就是把 .dex 文件从Zip压缩包中的提取，这样运行时无需再次从 APK 提取。

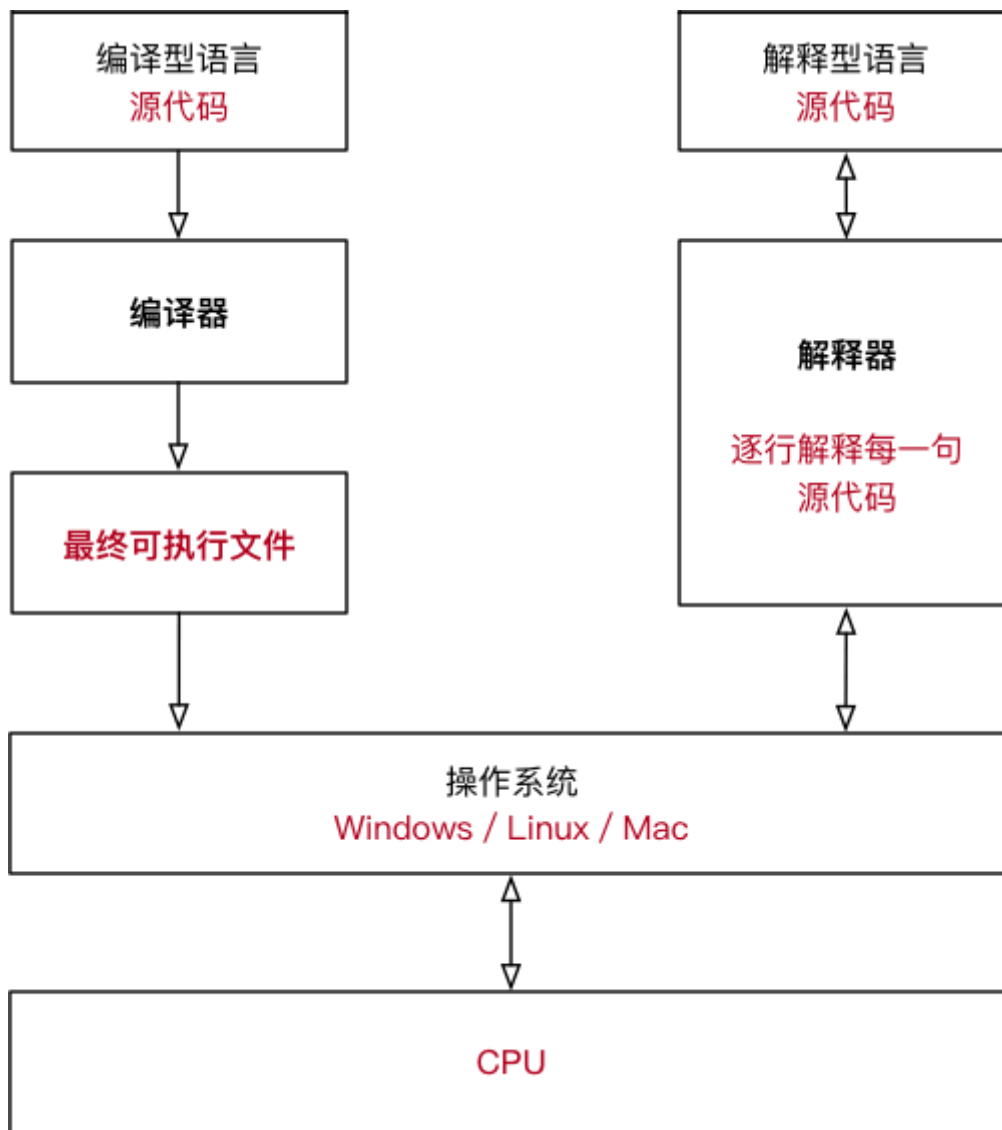
### 解释型语言和编译型语言

在开发中，无论我们使用Java、C++还是其他语言，实际上我们写的代码都是给人看的。然后我们的运行设备并没有人这么聪明，只能够看懂一些特定的简单的指令。因此在程序真正运行之前需要先将源代码转换成这种运行设备能够识别的二进制指令，也就是机器码。

但是不同的语言对什么时候、怎么转换成机器码又有不同的处理，我们按照对不同处理方式可以把语言分类为：**解释型语言和编译型语言**：

- **编译型语言**：利用**编译器**将源码一次性全部转换成二进制指令，生成可执行程序，如C/C++等。
- **解释型语言**：边执行，边使用**解释器**转换，如JavaScript、PHP等





不管是**编译器**还是**解释器**实际上都是把人能看懂的代码，翻译成CPU能够识别的指令(机器码)。但是编译型语言需要在开发完成以后将所有的源代码都转换成可执行程序，而可执行程序里面包含的就是机器码。不同操作系统对可执行文件的内部结构有不同的要求，彼此之间无法兼容，**因此编译型语言一般是不能跨平台的**，需要在不同平台运行，需要根据不同的平台生成对应平台的可执行文件。

相比于编译型语言，**解释型语言几乎都能跨平台**。解释型语言之所以能够跨平台，是因为有了解释器这个中间层。在不同的平台下，解释器会将相同的源代码转换成不同的机器码。

## 6.6 说说反射的应用场景，哪些框架？

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Java高级语言特性，注解与反射》

### 这道题想考察什么？

是否了解反射相关的理论知识

## 考察的知识点

1. 反射机制
2. 反射在框架中的应用
3. 获取Class类的主要方式

## 考生应该如何回答

反射是Java程序开发语言的特征之一，它允许动态地发现和绑定类、方法、字段，以及所有其他的由于有所产生的元素。通过反射，能够在需要时完成创建实例、调用方法和访问字段的工作。

反射机制主要提供功能有：

- 在运行时判断任意一个对象所属的类
- 在运行时构造任意一个类的对象
- 在运行时判断任意一个类所具有的成员变量和方法
- 在运行时调用任意一个对象的方法，通过反射甚至可以调用到private修饰的方法
- 生成动态代理

反射在框架中用的非常多。我们可以利用反射机制在Java程序中，动态的去调用一些protected甚至是private的方法或类，这样可以很大程度上满足我们的一些比较特殊需求。例如Activity的启动过程中Activity的对象的创建。

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent
customIntent) {

    //省略
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
        activity = mInstrumentation.newActivity(
            cl, component.getClassName(), r.intent);
        StrictMode.incrementExpectedActivityCount(activity.getClass());
        r.intent.setExtrasClassLoader(cl);
        r.intent.prepareToEnterProcess();
        if (r.state != null) {
            r.state.setClassLoader(cl);
        }
    }

    //上面代码可知Activity在创建对象的时候调用了mInstrumentation.newActivity();

    public Activity newActivity(ClassLoader cl, String className,
        Intent intent)
        throws InstantiationException, IllegalAccessException,
        ClassNotFoundException {
        //这里的className就是在manifest中注册的Activity name.
        return (Activity)cl.loadClass(className).newInstance();
    }

    //Activity对象的创建是通过反射完成的。java程序可以动态加载类定义，而这个动态加载的机制就是
    通过ClassLoader来实现的。
```

Java的每个.class文件里承载了类的所有信息内容，包含父类、接口、属性、构造、方法等；这些class文件会在程序运行状态下通过ClassLoader类机制加载到虚拟机内存中并产生一个对象实例，然后这个对象就可以对这块内存地址进行引用；一般我们通过New一个对象创建就行，而利用反射我们通过JVM查找并指定加载的类名，就可以创建对象实例，而在Java中，类的对象被实例化可以在内存中被引用时，

就可以获取该类的所有信息。可以说反射是所有插件化框架的基础。

```
//反射获取类的实例化对象
Class<?> cls=Class.forName("com.fanshe.Person"); //forName(包名.类名)
Person p=(Person)cls.newInstance();
//或
Person p = new Person();
Class<?> cls=p.getClass();
Person p2=(Person)cls.newInstance();
//或
Class<?> cls=Person.Class();
Person p=(Person)cls.newInstance();

//反射获取类的属性的值
Field getField = cls.getDeclaredField(fieldName); //传入属性名
Object getValue =getField.get(newClass); //传入实例值
//静态属性不用借助实例
Field getField = cls.getDeclaredField(fieldName);
Object getValue =getField.get(null);
```

反射在插件化、热修复、Retrofit等等几乎所有的常见框架中均有使用。

## 6.7 反射为什么慢？(滴滴)

### 这道题想考察什么？

是否了解反射的代价

### 考察的知识点

反射

### 考生如何回答

以反射方法为例，在使用反射时，首先需要获得对应的 Class 对象，然后获取需要反射执行的 Method 对象，调用 invoke 方法。

Java 反射效率低主要原因是：

1. 在反射执行方法时，参数需要拆装箱<sup>3</sup>

invoke 方法的参数是 Object[] 类型，当方法参数是基本数据类型时候，需要在此转化成 对应的包装类型，如 int需要转换为Integer，也就会额外生成Integer 对象。接着还需要将这些参数封装为 Object数组。

2. 需要检查方法可见性

反射时每次调用都必须检查方法的可见性，比如反射private方法时必须设置

`setAccessible(true)`，在执行时，invoke方法需要根据作用域与flag进行检查

3. 需要校验参数

反射时也必须检查每个实际参数与形式参数的类型匹配性

4. 反射方法难以内联

正常调用的方法能够进行内联优化，而反射将本来允许内联的方法变得复杂，无法内联

5. JIT 无法优化

反射涉及到动态加载，无法进行优化

## 6.8 动态代理是什么？如何实现？

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Retrofit中的注解反射与动态代理》

### 这道题想考察什么？

面试者对设计模式中的代理模式掌握情况，是否了解并能合理运用静态代理与动态代理，知道两者的区别；动态代理原理与其所涉及到的知识点

### 考察的知识点

代理模式，反射

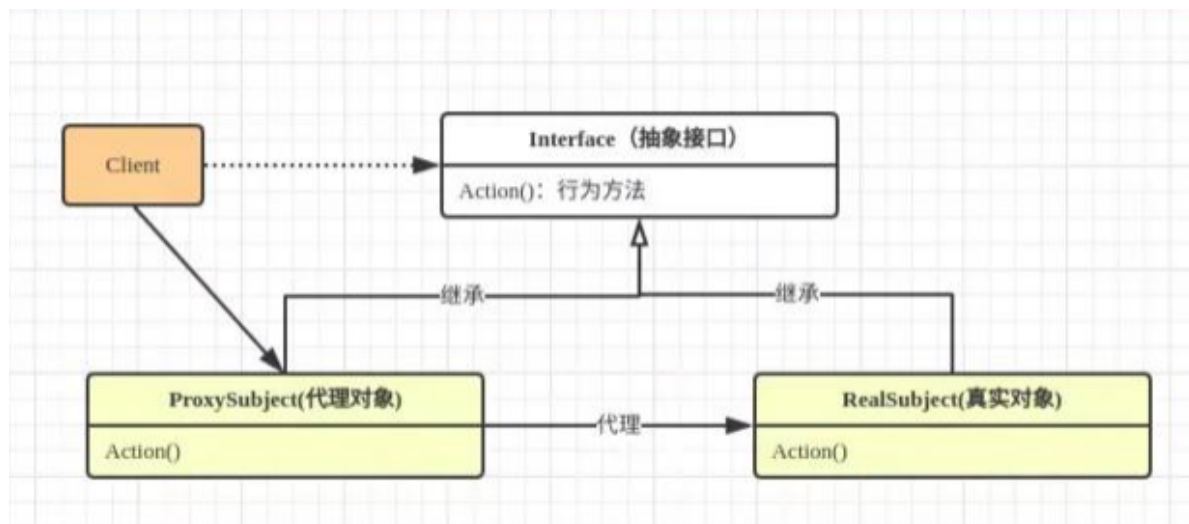
### 考生如何回答

代理模式，属于结构型模式。使用一个类代表另一个类的功能，在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。比如生活中常见的中介与租房，租房者不需要直接与房东交互，房东把房屋租赁交给中介代理。代理模式的目的有：

（1）通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来的不必要复杂性；

（2）通过代理对象对访问进行控制；

代理模式一般会有三个角色：



- 抽象角色：

- 指代理角色和真实角色对外提供的公共方法，一般为一个接口

- 真实角色：

- 需要实现抽象角色接口，定义了真实角色所要实现的业务逻辑，以便供代理角色调用。也就是真正的业务逻辑在此。

- 代理角色：

- 要实现抽象角色接口，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附

加自己的操作。将统一的流程控制都放到代理角色中处理！

## 静态代理

静态代理在进行一对一代理时，会出现时静态代理对象量多，可维护性差，而在一对多时，也会出现扩展能力差的问题。比如在进行开发时，同学们如果遇到NDK的问题，可以找我们享学课堂提问。因此我们创建对应的接口：

```
/**
 * 抽象角色： 定义了服务的接口
 */
public interface NDK {
    void ndk();
}
```

Lance老师擅长NDK问题：

```
/**
 * 真实角色： 需要实现抽象角色接口
 */
public class Lance implements NDK {

    @Override
    public void ndk() {
        System.out.println("回答问题");
    }
}
```

同学们一旦有NDK问题，就可以找到我们的助教老师：

```
/**
 * 代理角色： 需要实现抽象角色接口，代理真实角色
 */
public class Kiso implements NDK {

    private final NDK ndk;

    public VV(NDK ndk) {
        this.ndk = ndk;
    }

    //....前置处理
    public void before() {
        System.out.println("前置处理：了解同学问题");
    }

    //....后置处理
    public void after() {
        System.out.println("后置处理：回访答疑体验");
    }

    @Override
    public void ndk() {
```

```

        before();
        ndk.ndk();
        after();
    }
}

```

遇到了NDK问题直接向代理对象VV询问。

```

//创建真实对象
NDK lance = new Lance();
//Kiso代理真实对象
Kiso kiso = new Kiso(lance);
//向Kiso询问NDK问题
kiso.ndk();

```

在上面这个案例中，代理（Kiso）使客户端（学生）不需要知道实现类是什么（擅长NDK的是哪个老师），而客户端只需知道代理即可（解耦合）。

## 动态代理

作为一个传授系统性知识体系的线上教育机构，享学课堂不仅仅只能解决同学们的NDK问题，如果同学们遇到了其他的问题比如UI，享学课堂也要能够为同学们提供答疑服务。而此时静态代理的局限性就体现出来了：当我们的系统更新或者增加新的业务需求，可能需要新增很多目标接口和代理类。

而动态代理就能够很好的解决这个问题：

而动态代理就能够很好的解决这个问题：

```

//真实对象
NDK lance = new Lance();
//JAVA动态代理
NDK ndk = (NDK) Proxy.newProxyInstance(lance.getClass().getClassLoader(),
    lance.getClass().getInterfaces(), new
    ProxyInvokeHandler(lance));
ndk.ndk();

UI alvin = new Alvin();
UI ui = (UI) Proxy.newProxyInstance(alvin.getClass().getClassLoader(),
    alvin.getClass().getInterfaces(), new
    ProxyInvokeHandler(alvin));
ui.ui();

//代理多个接口
Object proxy = Proxy.newProxyInstance(alvin.getClass().getClassLoader(),
    new Class[]{NDK.class, UI.class}, new
    ProxyInvokeHandler(lance, alvin));
((UI)proxy).ui();
((NDK)proxy).ndk();

```

```

// 在Proxy.newProxyInstance创建的对象上调用任何方法都会回调此处的invoke方法
public class ProxyInvokeHandler implements InvocationHandler {
    //真实对象
    private Object realObject;

```

```

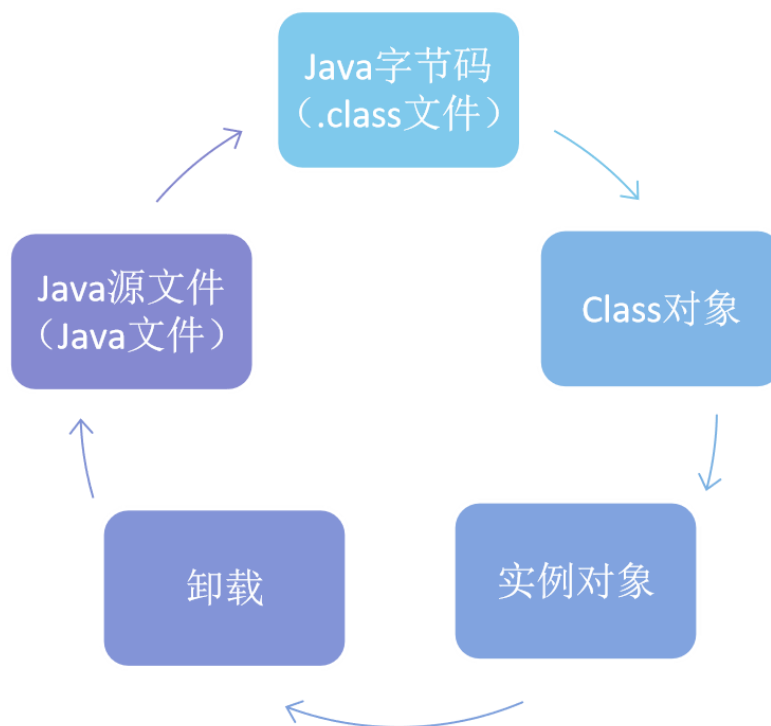
public ProxyInvokeHandler(Object realObject) {
    this.realObject = realObject;
}

/**
 *
 * @param o      代理对象
 * @param method 调用的方法
 * @param objects 方法的参数
 * @return
 * @throws Throwable
 */
@Override
public Object invoke(Object o, Method method, Object[] objects) throws
Throwable {
    //反射执行真实对象方法
    return method.invoke(realObject, objects);
}
}

```

## Java动态代理原理

我们的类一般由Java源文件编译出Java字节码.class文件，然后经过类加载器ClassLoader加载使用。



字节码的数据主要由一个真实存在的.class记录，类加载器读取这个.class文件中的字节码数据。而动态代理，则是在运行时通过：`Proxy.newProxyInstance` 在内存中直接生成类的字节码数据，然后创建此类实例对象返回。Java动态代理在内存中生成出来的字节码数据，我们可以写出到文件中去查看：

```
//生成NDK代理类:com.enjoy.lib.Proxy$0
byte[] bytes = ProxyGenerator.generateProxyClass("com.enjoy.lib.Proxy$0",
    new Class[]{NDK.class});

try {
    FileOutputStream fos = new FileOutputStream("/xxx/Proxy$0.class");
    fos.write(bytes);
    fos.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

## Android中没有ProxyGenerator，可以创建Java工程（使用JDK）

最后反编译查看 `Proxy.newProxyInstance` 创建的对象类型 `Proxy$0.class` 内容如下：

```
public final class Proxy$0 extends Proxy implements NDK {
    private static Method m1;
    private static Method m2;
    private static Method m3;
    private static Method m0;
    //InvocationHandler Proxy.newProxyInstance最后一个参数
    public Proxy$0(InvocationHandler var1) throws {
        super(var1);
    }

    //.....

    //在动态代理对象上调用ndk方法,通过InvocationHandler回调出去
    public final void ndk() throws {
        try {
            super.h.invoke(this, m3, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    //静态代码块，加载就会找到对应的method，比如ndk方法
    static {
        try {
            //.....
            m3 = Class.forName("com.xxx.NDK").getMethod("ndk");
            //.....
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```



## 6.9 动态代理的方法怎么初始化的？（字节跳动）

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Retrofit中的注解反射与动态代理》

### 这道题想考察什么？

面试者对设计模式中的代理模式掌握情况，是否了解并能合理运用静态代理与动态代理，知道两者的区别；动态代理原理与其所涉及到的知识点

### 考察的知识点

1. 代理模式
2. 反射

### 考生应该如何回答

动态代理原理见：《5.8 动态代理是什么？动态代理的原理是怎样的？》

在Java动态代理中会根据代理的接口生成Proxy派生类作为代理类。此类中会生成一段静态代码块，在静态代码块中使用反射获取到被代理类的所有方法并赋值给成员属性，以供后续使用。

在执行方法时，会通过 `InvocationHandler` 将此方法对应的被代理类方法(静态代码块获得的Method)对象以及参数回调给使用者。

```
public final class Proxy$0 extends Proxy implements NDK {

    //在动态代理对象上调用ndk方法,通过InvocationHandler回调出去
    public final void ndk() throws {
        try {
            super.h.invoke(this, m3, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    //静态代码块，加载就会找到对应的method，比如ndk方法
    static {
        try {
            //.....
            m3 = Class.forName("com.xxx.NDK").getMethod("ndk");
            //.....
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```

## 6.10 CGLIB动态代理（字节跳动）

## 这道题想考察什么？

代理思想与CGLIB实现的动态代理的原理及其中涉及到的知识点。

## 考察的知识点

ASM、字节码增强技术

## 考生如何回答

这个问题可能会与JDK动态代理一起讨论，所以在回答此问题之前需要首先清楚代理模式-动态代理的特点以及JDK动态代理实现原理：第五章—《动态代理是什么？动态代理的原理是怎样的？》

## CGLIB动态代理原理

CGLIB(Code Generation Library)是一个基于ASM的字节码生成库，它允许我们在运行时对字节码进行修改和动态生成。CGLIB通过继承方式实现代理，在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势织入横切逻辑。

```
//cglib和jdk动态代理不同，不止只能代理接口
public class HelloService {

    public void sayHello() {
        System.out.println("HelloService:sayHello");
    }

}

public class CglibMethod implements MethodInterceptor{
    /**
     * sub: cglib生成的代理类的对象
     * method: 被代理对象的方法
     * objects: 方法入参
     * methodProxy: 代理方法
     */
    @Override
    public Object intercept(Object sub, Method method, Object[] objects,
        MethodProxy methodProxy) throws Throwable{
        System.out.println("====插入前置处理====");
        Object object = methodProxy.invokeSuper(sub, objects);
        System.out.println("====插入后置处理====");
        return object;
    }
}

static void main(String[] args){
    Enhancer enhancer = new Enhancer();
    //设置要代理的类型
    enhancer.setSuperclass(HelloService.class);
    //设置代理对象调用方法回调，代理对象上调用方法回调: CglibMethod#intercept
    enhancer.setCallback(new CglibMethod());
    //创建代理对象
    HelloService proxy= (HelloService)enhancer.create();
    // 通过代理对象调用目标方法
    proxy.sayHello();
}
```

JDK代理只能代理接口，而CGLIB动态代理则没有此类强制性要求。CGLIB会生成继承被代理类的一个子类（代理类），并在这个类中对代理方法进行强化处理(前置处理、后置处理等)。在CGLIB底层，其实是借助了ASM这个非常强大的Java字节码生成框架。

创建代理对象的几个步骤：

- 生成代理类的二进制字节码文件；
- 加载二进制字节码，生成Class对象；
- 通过反射机制获得实例构造，并创建代理类对象

生成的代理类Class文件反编译之后的Java代码如下：

```
public class HelloService$$EnhancerByCGLIB$$1f1876d extends HelloService
implements Factory {
    private static final Method CGLIB$sayHello$0$Method; //被代理方法
    private static final MethodProxy CGLIB$sayHello$0$Proxy; //代理方法
    static {
        CGLIB$STATICHOOK1();
    }
    static void CGLIB$STATICHOOK1() {
        //.....
        Class var0 =
        Class.forName("com.enjoy.compile.HelloService$$EnhancerByCGLIB$$1f1876d");
        Class var1;
        // 反射获得被代理类的Method: sayHello
        CGLIB$sayHello$0$Method = ReflectUtils.findMethods(new String[]
{"sayHello", "()V"}, (var1 =
Class.forName("com.enjoy.compile.HelloService")).getDeclaredMethods())[0];
        // 创建 HelloService#sayHello的代理方法
        CGLIB$sayHello$0$Proxy = MethodProxy.create(var1, var0, "()V",
"sayHello", "CGLIB$sayHello$0");
        // .....
    }

    final void CGLIB$sayHello$0() {
        super.sayHello();
    }

    public final void sayHello() {
        // CglibMethod
        MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
        if (var10000 == null) {
            CGLIB$BIND_CALLBACKS(this);
            var10000 = this.CGLIB$CALLBACK_0;
        }
        // 回调CglibMethod#intercept 方法
        if (var10000 != null) {
            var10000.intercept(this, CGLIB$sayHello$0$Method, CGLIB$emptyArgs,
CGLIB$sayHello$0$Proxy);
        } else {
            super.sayHello();
        }
    }
    //.....
}
```

通过CGLIB生成的代理类可以看出，实际上我们创建的代理对象，是由CGLIB生成的一个代理类的子类对象，而这个子类中也和父类一样实现了 `sayHello` 方法。而调用 `sayHello` 方法，则会回调我们设置的 `Callback`：`CglibMethod#intercept` 方法。

```
@Override
    public Object intercept(Object sub, Method method, Object[] objects,
        MethodProxy methodProxy) throws Throwable{
        System.out.println("====插入前置处理====");
        Object object = methodProxy.invokeSuper(sub, objects);
        System.out.println("====插入后置处理====");
        return object;
    }
```

在回调中我们进行执方法的前置处理与后置处理，使用 `methodProxy.invokeSuper` 去调起被代理类中实现的 `sayHello` 方法，此时会调用到CGLIB生成代理类中的：

```
final void CGLIB$sayHello$0() {
    //执行父类: HelloService#sayHello
    super.sayHello();
}
```

## 总结

- 1、CGLIB在运行时生成代理类 `HelloService$$EnhancerByCGLIB$$YYYY` 继承被代理类 `HelloService`。注意：`final`方法无法代理；
- 2、代理类会为委托方法生成两个方法，一个是重写的 `sayHello` 方法，另一个是 `CGLIB$sayHello$0` 方法；
- 3、当执行代理对象的 `sayHello` 方法时，则将调用创建代理对象时 `Enhancer.setCallback` 设置的 `MethodInterceptor.intercept` 方法，在 `intercept` 方法中调用 `methodProxy.invokeSuper` 则回调到 `CGLIB$sayHello$0` 方法。

---

## 6.11 为什么IO是耗时操作？

### 详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《Java IO在Android中的运用》

### 这道题想考察什么？

Java中常用的IO流程，以及项目中使用的注意事项

## 考察的知识点

Java IO原理

## 考生应该如何回答

计算机硬件上使用DMA(Direct Memory Access，直接内存存取)来访问磁盘等IO，也就是请求发出后，CPU就不再管了，直到DMA处理器完成任务，再通过**中断**告诉CPU完成了。

所以，单独的一个IO时间，对CPU的占用是很少的，阻塞了就更不会占用CPU了，因为程序都不继续运行了，CPU时间交给其它线程和进程了。

虽然IO不会占用大量的CPU时间，但是"量变引起质变"，所以面对大量IO的任务，有时候是需要算法来合并IO，或者通过cache来缓解IO压力的。