

Android 开发相关源码精编解析

Android 开发相关源码精编解析.....	1
1.深入解析微信 MMKV 源码.....	6
初始化.....	7
获取.....	8
获取 MMKV 对象.....	8
构造 MMKV 对象.....	11
从文件加载数据.....	12
修改.....	15
数据写入.....	15
内存重整.....	18
删除.....	21
读取.....	22
文件回写.....	23
Protobuf 处理.....	25
Protobuf 编码.....	25
Protobuf 实现.....	27
跨进程锁实现.....	32
跨进程锁的选择.....	32
文件锁.....	33
文件锁封装.....	34
状态同步.....	37
总结.....	39
2.深入解析阿里巴巴路由框架 ARouter 源码.....	41
初始化.....	41
解析 ClassName.....	44
初始化 Warehouse.....	46
路由跳转.....	48
Postcard 的创建.....	48
Postcard 的补全.....	52
执行跳转.....	55
Service 的获取.....	57
拦截器机制.....	59
注解处理.....	61
总结.....	61
编译期注解处理.....	62
初始化.....	62
路由.....	62
3.深入解析 AsyncTask 源码（一款 Android 内置的异步任务执行库）.....	63
功能概述.....	64
创建.....	65
执行.....	67

取消.....	70
线程池.....	71
不足之处.....	73
总结.....	74
4.深入解析 Volley 源码（一款 Google 推出的网络请求框架）.....	75
Volley.....	75
创建 RequestQueue.....	75
RequestQueue.....	79
创建.....	79
启动.....	80
入队.....	81
停止.....	82
结束.....	82
ExecutorDelivery.....	83
ResponseDeliveryRunnable.....	84
NetworkDispatcher.....	86
CacheDispatcher.....	89
Request.....	93
构建.....	93
取消.....	94
Response 转换.....	94
Response 的交付.....	95
Error 的交付.....	95
结束.....	96
缓存.....	97
Event 机制.....	98
Response.....	98
Network.....	99
HttpStack.....	103
缓存机制.....	104
总结.....	105
5.深入解析 Retrofit 源码.....	106
前言.....	106
目录.....	107
2. 创建网络请求接口的实例.....	135
2.1 使用步骤.....	135
2.2 源码分析.....	137
1. 外观模式.....	140
2. 代理模式.....	141
下面看源码分析.....	142

6.深入解析 OkHttp 源码.....	178
6.1 OkHttp 3.7 源码分析（一）——整体架构.....	178
简单使用.....	179
总体架构.....	181
6.2OkHttp 3.7 源码分析（二）——拦截器&一个实际网络请求的实现.....	185
1.构造 Demo.....	186
2. 发起请求.....	186
3. 构建拦截器链.....	188
4 RetryAndFollowUpInterceptor.....	195
5 BridgeInterceptor.....	195
6. CacheInterceptor.....	198
7 ConnectInterceptor.....	203
8. CallServerInterceptor.....	204
8.整体流程.....	206
6.3OkHttp 3.7 源码分析（三）——任务队列.....	207
1. 线程池的优点.....	208
2. OkHttp 的任务队列.....	209
3. Dispatcher 分发器.....	211
4. 总结.....	217
6.4OkHttp 3.7 源码分析（四）——缓存策略.....	217
1. HTTP 缓存策略.....	218
2. Cache 源码分析.....	219
3. DiskLruCache.....	230
6.5OkHttp 3.7 源码分析（五）——连接池.....	249
1. 背景.....	250
2. 连接池的使用与分析.....	252
7.深入解析 ButterKnife 源码.....	270
概述版.....	270
分析版.....	270
8.深入解析 Okio 源码（一套简洁高效的 I/O 库）.....	271
Sink.....	271
Source.....	273
Buffer.....	275
Segment.....	276
数据转移.....	281
BufferedSource.....	282
BufferedSink.....	285
write.....	285
flush.....	287
emit.....	287
Timeout 超时机制.....	288
Timeout.....	288
AsyncTimeout.....	290
总结.....	298

9.深入解析 SharedPreferences 源码.....	298
获取 SharedPreferences.....	298
根据名称获取 SP.....	299
获取 SP 名称对应的 File 对象.....	300
根据创建的 File 对象获取 SP.....	300
缓存机制.....	302
SharedPreferencesImpl.....	303
从 Disk 加载数据.....	303
编辑 SharedPreferences.....	305
获取 Editor.....	306
等待读取机制.....	306
EditorImpl.....	307
提交 SharedPreferences.....	308
apply.....	308
commit.....	310
同步数据至内存.....	311
写入数据至硬盘.....	312
总结.....	313
10.深入解析 EventBus 源码.....	314
getDefault 方法.....	314
register 方法.....	316
搜寻过程.....	317
订阅过程.....	322
post 方法.....	324
存放线程信息.....	325
搜寻 Subscription.....	328
未找到对应 Subscription 的处理.....	329
执行对应 Subscription.....	330
postSticky 方法.....	332
unregister 方法.....	332
Poster.....	333
HandlerPoster.....	334
AsyncPoster.....	335
BackgroundPoster.....	336
PendingPost 池.....	337
总结.....	338
11.Android 自定义注解初探.....	339
什么是注解.....	339
元注解.....	340
自定义注解.....	343
运行时注解的处理.....	345
编译时注解的处理.....	347
在 Android 中使用自定义注解.....	353
gradle 配置.....	353

定义注解.....	353
实现注解处理器.....	354
在项目中使用.....	356
ButterKnife 的流程.....	359
12.View 的工作机制源码分析.....	359
13.Android 触摸事件分发机制源码分析.....	360
14.Android 按键事件分发机制源码分析.....	361
15.深入解析 Handler 源码.....	363
发送消息.....	363
post & sendMessage.....	363
消息入队.....	366
enqueueMessage.....	366
消息循环.....	368
消息遍历.....	370
消息的处理.....	374
同步屏障机制.....	374
加入同步屏障.....	374
移除同步屏障.....	375
同步屏障的作用.....	377
阻塞唤醒机制.....	377
epoll.....	377
native 初始化.....	380
native 阻塞实现.....	381
native 唤醒.....	383
总结.....	384
相关问题.....	385
问题 1.....	385
问题 2.....	386
16.深入解析 Binder 源码.....	387
简介.....	387
整体图.....	389
设计架构.....	389
数据结构鸟瞰.....	390
Binder 设计基础.....	391
ioctl(): 内核/用户空间调用.....	391
mmap(): 内核/用户空间内存映射.....	392
Binder 中的 ONEWAY 与非 ONEWAY 调用.....	393
Binder 中的生产者与消费者.....	395
Binder 代理对象的 handle 句柄.....	399
Binder 内核中的红黑树.....	404
辅助功能：实名服务的注册与获取.....	405
核心功能：跨进程数据传输.....	409
辅助功能：匿名服务的跨进程传输与回调.....	419
辅助功能：死亡回调的注册与获取.....	424

附：内核基础知识.....	427
17.深入解析 JNI 源码.....	431
简介.....	431
架构图.....	432
示例.....	432
在 JAVA 中调用 Native 方法.....	432
在 Native 中调用 JAVA 方法.....	434
开机 JNI 初始化.....	435
System.loadLibrary()原理.....	442
18.深入解析 Glide 源码.....	445
功能介绍.....	446
Glide 主入口.....	446
RequestManager (Glide.with(...))	446
RequestBuilder.....	447
代码结构.....	448
ModelLoader.....	449
DataFetcher.....	450
Target.....	451
Resource.....	452
ResourceTransformation.....	452
Pool.....	453
Cache.....	453
Decoder.....	454
Encoder.....	455
执行流程.....	455
4.2.1 缓存 target，并启动 Request.....	462
4.2.2 数据下载完成后的缓存处理 SourceGenerator.onDataReady.....	477
4.2.3 装载流程 回调通知这里就不打算多讲了，主要线路如下：	482
装载流程 Debug 流程图.....	483
装载流程 Debug 流程图 2.....	483
结束语.....	485
思考.....	486

1. 深入解析微信 MMKV 源码

MMKV 是微信于 2018 年 9 月 20 日开源的一个 K-V 存储库，它与 SharedPreferences 相似，但又在更高的效率下解决了其不支持跨进程读写等弊端。

一年前的自己因对它非常感兴趣写下了一篇 [【Android】 MMKV 源码浅析](#)。不过由于当时还是大二，知识的储备还不够丰富，因此整体的分析在某些细节上还比较稚嫩。由于对这个库很感兴趣，因此尝试重新对它进行一次源码解析，对以前分析不够到位的地方进行补充，并且将以前没有研究的部分细致研究一下。

初始化

通过 MMKV.initialize 方法可以实现 MMKV 的初始化：

```
public static String initialize(Context context) {  
  
    String root = context.getFilesDir().getAbsolutePath() + "/mmkv";  
  
    return initialize(root);  
  
}
```

它采用了内部存储空间下的 `mmkv` 文件夹作为根目录，之后调用了 `initialize` 方法。

```
public static String initialize(String rootDir) {  
    MMKV.rootDir = rootDir;  
    jniInitialize(MMKV.rootDir);  
    return rootDir;  
}
```

调用到了 `jniInitialize` 这个 Native 方法进行 Native 层的初始化：

```
extern "C" JNIEXPORT JNICALL void  
Java_com_tencent_mmkv_MMKV_jniInitialize(JNIEnv *env, jobject obj, jstring roo  
tDir) {  
    if (!rootDir) {  
        return;  
    }  
    const char *kstr = env->GetStringUTFChars(rootDir, nullptr);  
    if (kstr) {  
        MMKV::initializeMMKV(kstr);  
        env->ReleaseStringUTFChars(rootDir, kstr);  
    }  
}
```

```
    }
}
```

这里通过 `MMKV::initializeMMKV` 对 `MMKV` 类进行了初始化：

```
void MMKV::initializeMMKV(const std::string &rootDir) {
    static pthread_once_t once_control = PTHREAD_ONCE_INIT;
    pthread_once(&once_control, initialize);
    g_rootDir = rootDir;
    char *path = strdup(g_rootDir.c_str());
    mkPath(path);
    free(path);
    MMKVInfo("root dir: %s", g_rootDir.c_str());
}
```

实际上就是记录下了 `rootDir` 并创建对应的根目录，由于 `mkPath` 方法创建目录时会修改字符串的内容，因此需要复制一份字符串进行。

获取

获取 `MMKV` 对象

通过 `mmkvWithID` 方法可以获取 `MMKV` 对象，它传入的 `mmapID` 就对应了 `SharedPreferences` 中的 `name`，代表了一个文件对应的 `name`，而 `relativePath` 则对应了一个相对根目录的相对路径。

```
@Nullable
public static MMKV mmkvWithID(String mmapID, String relativePath) {
    if (rootDir == null) {
        throw new IllegalStateException("You should Call MMKV.initialize() first.");
    }
    long handle = getMMKVWithID(mmapID, SINGLE_PROCESS_MODE, null, relativePath);
    if (handle == 0) {
        return null;
    }
    return new MMKV(handle);
```

}

它调用到了 `getMMKVWithId` 这个 Native 方法，并获取到了一个 `handle` 构造了 Java 层的 MMKV 对象返回。这是一种很常见的手法，Java 层通过持有 Native 层对象的地址从而与 Native 对象通信（例如 Android 中的 Surface 就采用了这种方式）。

```
extern "C" JNIEXPORT JNICALL jlong Java_com_tencent_mmkv_MMKV_getMMKVWithID(
    JNIEnv *env, jobject obj, jstring mmapID, jint mode, jstring cryptKey, jstring relativePath) {
    MMKV *kv = nullptr;
    // mmapID 为 null 返回空指针
    if (!mmapID) {
        return (jlong) kv;
    }
    string str = jstring2string(env, mmapID);
    bool done = false;
    // 如果需要进行加密, 获取用于加密的 key, 最后调用 MMKV::mmkvWithID
    if (cryptKey) {
        string crypt = jstring2string(env, cryptKey);
        if (crypt.length() > 0) {
            if (relativePath) {
                string path = jstring2string(env, relativePath);
                kv = MMKV::mmkvWithID(str, DEFAULT_MMAP_SIZE, (MMKVMODE) mode,
&crypt, &path);
            } else {
                kv = MMKV::mmkvWithID(str, DEFAULT_MMAP_SIZE, (MMKVMODE) mode,
&crypt, nullptr);
            }
            done = true;
        }
    }
    // 如果不需要加密, 则调用 mmkvWithID 不传入加密 key, 表示不进行加密
    if (!done) {
        if (relativePath) {
            string path = jstring2string(env, relativePath);
            kv = MMKV::mmkvWithID(str, DEFAULT_MMAP_SIZE, (MMKVMODE) mode, nullptr,
&ptr, &path);
        } else {
            kv = MMKV::mmkvWithID(str, DEFAULT_MMAP_SIZE, (MMKVMODE) mode, nullptr,
&ptr, nullptr);
        }
    }
}
```

```
    }
    return (jlong) kv;
}
```

这里实际上调用了 `MMKV::mmkvWithID` 方法，它根据是否传入用于加密的 `key` 以及是否使用相对路径调用了不同的方法。

```
MMKV *MMKV::mmkvWithID(
    const std::string &mmapID, int size, MMKVMODE mode, string *cryptKey, string *relativePath) {
    if (mmapID.empty()) {
        return nullptr;
    }
    // 加锁
    SCOPEDLOCK(g_instanceLock);
    // 将 mmapID 与 relativePath 结合生成 mmapKey
    auto mmapKey = mappedKVKey(mmapID, relativePath);
    // 通过 mmapKey 在 map 中查找对应的 MMKV 对象并返回
    auto itr = g_instanceDic->find(mmapKey);
    if (itr != g_instanceDic->end()) {
        MMKV *kv = itr->second;
        return kv;
    }
    // 如果找不到，构建路径后构建 MMKV 对象并加入 map
    if (relativePath) {
        auto filePath = mappedKVPathWithID(mmapID, mode, relativePath);
        if (!isFileExist(filePath)) {
            if (!createFile(filePath)) {
                return nullptr;
            }
        }
        MMKVInfo("prepare to load %s (id %s) from relativePath %s", mmapID.c_str(),
r(), mmapKey.c_str(),
relativePath->c_str());
    }
    auto kv = new MMKV(mmapID, size, mode, cryptKey, relativePath);
    (*g_instanceDic)[mmapKey] = kv;
    return kv;
}
```

这里的步骤如下：

1. 通过 `mmapedKVKey` 方法对 `mmapID` 及 `relativePath` 进行结合生成了对应的 `mmapKey`, 它会将它们两者的结合经过 `md5` 从而生成对应的 `key`, 主要目的是为了支持不同相对路径下的同名 `mmapID`。
2. 通过 `mmapKey` 在 `g_instanceDic` 这个 `map` 中查找对应的 `MMKV` 对象, 如果找到直接返回。
3. 如果找不到对应的 `MMKV` 对象, 构建一个新的 `MMKV` 对象, 加入 `map` 后返回。

构造 `MMKV` 对象

我们可以看看在 `MMKV` 的构造函数中做了什么:

```
MMKV::MMKV(  
    const std::string &mmapID, int size, MMKVMode mode, string *cryptKey, string *relativePath)  
    : m_mmapID(mmapedKVKey(mmapID, relativePath))  
    // ...)  
    // ...  
  
    if (m_isAshmem) {  
        m_ashmemFile = new MmapedFile(m_mmapID, static_cast<size_t>(size), MMAP_ASHEM);  
        m_fd = m_ashmemFile->getFd();  
    } else {  
        m_ashmemFile = nullptr;  
    }  
    // 通过加密 key 构建 AES 加密对象 AESCrypt  
    if (cryptKey && cryptKey->length() > 0) {  
        m_crypter = new AESCrypt((const unsigned char *) cryptKey->data(), cryptKey->length());  
    }  
    // 赋值操作  
    // 加锁后调用 LoadFromFile 加载数据  
    {  
        SCOPEDLOCK(m_sharedProcessLock);  
        loadFromFile();  
    }
```

```
    }
}
```

这里进行了一些赋值操作，之后如果需要加密则根据用于加密的 `cryptKey` 生成对应的 `AESCrypt` 对象用于 `AES` 加密。最后，加锁后通过 `loadFromFile` 方法从文件中读取数据，这里的锁是一个跨进程的文件共享锁。

从文件加载数据

我们都知道，MMKV 是基于 `mmap` 实现的，通过内存映射在高效率的同时保证了数据的同步写入文件，`loadFromFile` 中就会真正进行内存映射：

```
void MMKV::loadFromFile() {
    // ...
    // 打开对应的文件
    m_fd = open(m_path.c_str(), O_RDWR | O_CREAT, S_IRWXU);
    if (m_fd < 0) {
        MMKVError("fail to open:%s, %s", m_path.c_str(), strerror(errno));
    } else {
        // 获取文件大小
        m_size = 0;
        struct stat st = {0};
        if (fstat(m_fd, &st) != -1) {
            m_size = static_cast<size_t>(st.st_size);
        }
        // 将文件大小对齐到页大小的整数倍，用 0 填充不足的部分
        if (m_size < DEFAULT_MMAP_SIZE || (m_size % DEFAULT_MMAP_SIZE != 0)) {
            size_t oldSize = m_size;
            m_size = ((m_size / DEFAULT_MMAP_SIZE) + 1) * DEFAULT_MMAP_SIZE;
            if (ftruncate(m_fd, m_size) != 0) {
                MMKVError("fail to truncate [%s] to size %zu, %s", m mmapID.c_st
r(), m_size,
                           strerror(errno));
            }
            m_size = static_cast<size_t>(st.st_size);
        }
        zeroFillFile(m_fd, oldSize, m_size - oldSize);
    }
    // 通过 mmap 将文件映射到内存
}
```

```

        m_ptr = (char *) mmap(nullptr, m_size, PROT_READ | PROT_WRITE, MAP_SHARED, m_fd, 0);
        if (m_ptr == MAP_FAILED) {
            MMKVError("fail to mmap [%s], %s", m_mmapID.c_str(), strerror(errno));
        } else {
            memcpy(&m_actualSize, m_ptr, Fixed32Size);
            MMKVInfo("loading [%s] with %zu size in total, file size is %zu", m_mmapID.c_str(),
                      m_actualSize, m_size);
            bool loadFromFile = false, needFullWriteback = false;
            if (m_actualSize > 0) {
                if (m_actualSize < m_size && m_actualSize + Fixed32Size <= m_size) {
                    // 对文件进行 CRC 校验, 如果失败根据策略进行不同对处理
                    if (checkFileCRCValid()) {
                        loadFromFile = true;
                    } else {
                        // CRC 校验失败, 如果策略是错误时恢复, 则继续读取, 并且最后
                        // 需要进行回写
                        auto strategic = onMMKVCRCCheckFail(m_mmapID);
                        if (strategic == OnErrorRecover) {
                            loadFromFile = true;
                            needFullWriteback = true;
                        }
                    }
                } else {
                    // 文件大小有误, 若策略是错误时恢复, 则继续读取, 并且最后需要进
                    // 行回写
                    auto strategic = onMMKVFileLengthError(m_mmapID);
                    if (strategic == OnErrorRecover) {
                        loadFromFile = true;
                        needFullWriteback = true;
                    }
                }
            }
            // 从文件中读取内容
            if (loadFromFile) {
                MMKVInfo("loading [%s] with crc %u sequence %u", m_mmapID.c_str(),
                          m_metaInfo.m_crcDigest, m_metaInfo.m_sequence);
                // 读取 MMBuffer
            }
        }
    }
}

```

```

        MMBuffer inputBuffer(m_ptr + Fixed32Size, m_actualSize, MMBufferNoCopy);
        // 如果需要解密，对文件进行解密
        if (m_crypter) {
            decryptBuffer(*m_crypter, inputBuffer);
        }
        // 通过 MiniPBCoder 将 MMBuffer 转换为 Map
        m_dic.clear();
        MiniPBCoder::decodeMap(m_dic, inputBuffer);
        // 构造用于输出的 CodeOutputData
        m_output = new CodedOutputData(m_ptr + Fixed32Size + m_actualSize,
                                       m_size - Fixed32Size - m_actualSize);
        if (needFullWriteback) {
            fullWriteback();
        } else {
            SCOPEDLOCK(m_exclusiveProcessLock);
            if (m_actualSize > 0) {
                writeActualSize(0);
            }
            m_output = new CodedOutputData(m_ptr + Fixed32Size, m_size - Fixed32Size);
            recalculateCRCDigest();
        }
        MMKVInfo("loaded [%s] with %zu values", m_mmapID.c_str(), m_dic.size());
    }
}
if (!isValid()) {
    MMKVWarning("[%s] file not valid", m_mmapID.c_str());
}
m_needLoadFromFile = false;
}

```

这里的代码虽然长，但逻辑还是非常清晰的，步骤如下：

1. 打开文件并获取文件大小，将文件的大小对齐到页的整数倍，不足则补 0
(与内存映射的原理有关，内存映射是基于页的换入换出机制实现的)
2. 通过 `mmap` 函数将文件映射到内存中，得到指向该区域的指针 `m_ptr`。

3. 对文件进行长度校验及 CRC 校验（循环冗余校验，可以校验文件完整性），在失败的情况下会根据当前策略进行抉择，如果策略是失败时恢复，则继续读取，并且在最后将 map 中的内容回写到文件。
4. 通过 `m_ptr` 构造出一块用于管理 MMKV 映射内存的 `MMBuffer` 对象，如果需要解密，通过之前构造的 `AESCrypt` 进行解密。
5. 由于 **MMKV** 使用了 **protobuf** 进行序列化，通过 `MiniPBCoder::decodeMap` 方法将 protobuf 转换成对应的 map。
6. 构造用于输出的 `CodedOutputData` 类，如果需要回写（CRC 校验或文件长度校验失败），则调用 `fullWriteback` 方法将 map 中的数据回写到文件。

修改

数据写入

Java 层的 MMKV 对象继承了 `SharedPreferences` 及 `SharedPreferences.Editor` 接口并实现了一系列如 `.putInt`、`putLong` 的方法用于对存储的数据进行修改，我们以 `.putInt` 为例：

```
@Override  
public Editor putInt(String key, int value) {  
    encodeInt(nativeHandle, key, value);  
    return this;  
}
```

它调用到了 `encodeInt` 这个 Native 方法：

```
extern "C" JNIREPORT JNICALL jboolean Java_com_tencent_mmkv_MMKV_encodeInt(  
    JNIEnv *env, jobject obj, jlong handle, jstring oKey, jint value) {  
    MMKV *kv = reinterpret_cast<MMKV *>(handle);  
    if (kv && oKey) {
```

```
        string key = jstring2string(env, oKey);
        return (jboolean) kv->setInt32(value, key);
    }
    return (jboolean) false;
}
```

这里将 Java 层持有的 NativeHandle 转为了对应的 MMKV 对象，之后调用了其 `setInt32` 方法：

```
bool MMKV::setInt32(int32_t value, const std::string &key) {
    if (key.empty()) {
        return false;
    }
    // 构造值对应的 MMBuffer，通过 CodedOutputData 将其写入 Buffer
    size_t size = pbInt32Size(value);
    MMBuffer data(size);
    CodedOutputData output(data.getPtr(), size);
    output.writeInt32(value);
    return setDataForKey(std::move(data), key);
}
```

这里首先获取到了写入的 `value` 在 `protobuf` 中所占据的大小，之后为其构造了对应的 `MMBuffer` 并将数据写入了这段 `Buffer`，最后调用到了 `setDataForKey` 方法（`std::move` 是 C++ 11 的特性，我们可以简单理解成赋值，它通过直接移动内存减少了拷贝）。

同时可以发现 `CodedOutputData` 是与 `Buffer` 交互的桥梁，可以通过它实现向 `MMBuffer` 中写入数据。

```
bool MMKV::setDataForKey(MMBuffer &&data, const std::string &key) {
    if (data.length() == 0 || key.empty()) {
        return false;
    }
    // 获取写锁
    SCOPEDLOCK(m_lock);
    SCOPEDLOCK(m_exclusiveProcessLock);
    // 确保数据已读入内存
    checkLoadData();
```

```

    // 将 data 写入 map 中
    auto itr = m_dic.find(key);
    if (itr == m_dic.end()) {
        itr = m_dic.emplace(key, std::move(data)).first;
    } else {
        itr->second = std::move(data);
    }
    m_hasFullWriteback = false;
    return appendDataWithKey(itr->second, key);
}

```

这里在确保数据已读入内存的情况下将 data 写入了对应的 map，之后调用了 appendDataWithKey 方法：

```

bool MMKV::appendDataWithKey(const MMBuffer &data, const std::string &key) {
    size_t keyLength = key.length();
    // 计算写入到映射空间中的 size
    size_t size = keyLength + pbRawVarint32Size((int32_t) keyLength);
    size += data.length() + pbRawVarint32Size((int32_t) data.length());
    // 要写入，获取写锁
    SCOPEDLOCK(m_exclusiveProcessLock);
    // 确定剩余映射空间足够
    bool hasEnoughSize = ensureMemorySize(size);
    if (!hasEnoughSize || !isValid()) {
        return false;
    }
    if (m_actualSize == 0) {
        auto allData = MiniPBCoder::encodeDataWithObject(m_dic);
        if (allData.length() > 0) {
            if (m_crypter) {
                m_crypter->reset();
                auto ptr = (unsigned char *) allData.getPtr();
                m_crypter->encrypt(ptr, ptr, allData.length());
            }
            writeActualSize(allData.length());
            m_output->writeRawData(allData); // note: don't write size of data
            recalculateCRCDigest();
            return true;
        }
        return false;
    } else {
        writeActualSize(m_actualSize + size);
        m_output->writeString(key);
    }
}

```

```
m_output->writeData(data); // note: write size of data
auto ptr = (uint8_t *) m_ptr + Fixed32Size + m_actualSize - size;
if (m_crypter) {
    m_crypter->encrypt(ptr, ptr, size)
}
updateCRCDigest(ptr, size, KeepSequence);
return true;
}
```

这里首先计算了即将写入到映射空间的内容大小，之后调用了 `ensureMemorySize` 方法确保剩余映射空间足够。

如果 `m_actualSize` 为 0，则会通过 `MiniPBCoder::encodeDataWithObject` 将整个 `map` 转换为对应的 `MMBuffer`，加密后通过 `CodedOutputData` 写入，最后重新计算 CRC 校验码。否则会将 `key` 和对应 `data` 写入，最后更新 CRC 校验码。

`m_actualSize` 是位于文件的首部的，因此是否为 0 取决于文件对应位置。

同时值得注意的是：由于 `protobuf` 不支持增量更新，为了避免全量写入带来的性能问题，`MMKV` 在文件中的写入并不是通过修改文件对应的位置，而是直接在后面 `append` 一条新的数据，即使是修改了已存在的 `key`。而读取时只记录最后一条对应 `key` 的数据，这样显然会在文件中存在冗余的数据。这样设计的原因我认为是出于性能的考量，**MMKV 中存在着一套内存重整机制用于对冗余的 key-value 数据进行处理**。它正是在确保内存充足时实现的。

内存重整

我们接下来看看 `ensureMemorySize` 是如何确保映射空间是否足够的：

```

bool MMKV::ensureMemorySize(size_t newSize) {
    // ...
    if (newSize >= m_output->spaceLeft()) {
        // 如果内存剩余大小不足以写入，尝试进行内存重整，将 map 中的数据重新写入 protoBuf 文件
        static const int offset = pbFixed32Size(0);
        MMBuffer data = MiniPBCoder::encodeDataWithObject(m_dic);
        size_t lenNeeded = data.length() + offset + newSize;
        if (m_isAshmem) {
            if (lenNeeded > m_size) {
                MMKVWarning("ashmem %s reach size limit:%zu, consider configure with larger size",
                           m_mmapID.c_str(), m_size);
                return false;
            }
        } else {
            size_t avgItemSize = lenNeeded / std::max<size_t>(1, m_dic.size());
            size_t futureUsage = avgItemSize * std::max<size_t>(8, (m_dic.size() + 1) / 2);
            // 如果内存重整后仍不足以写入，则将大小不断乘2直至足够写入，最后通过 mmap 重新映射文件
            if (lenNeeded >= m_size || (lenNeeded + futureUsage) >= m_size) {
                size_t oldSize = m_size;
                do {
                    // double 空间直至足够
                    m_size *= 2;
                } while (lenNeeded + futureUsage >= m_size);
                // ...
                if (ftruncate(m_fd, m_size) != 0) {
                    MMKVError("fail to truncate [%s] to size %zu, %s", m_mmapID.c_str(), m_size,
                               strerror(errno));
                    m_size = oldSize;
                    return false;
                }
                // 用零填充不足部分
                if (!zeroFillFile(m_fd, oldSize, m_size - oldSize)) {
                    MMKVError("fail to zeroFile [%s] to size %zu, %s", m_mmapID.c_str(), m_size,
                               strerror(errno));
                    m_size = oldSize;
                    return false;
                }
            } else {
                // unmap
            }
        }
    }
}

```

```

        if (munmap(m_ptr, oldSize) != 0) {
            MMKVError("fail to munmap [%s], %s", m mmapID.c_str(), strerror(errno));
        }

        // 重新通过 mmap 映射
        m_ptr = (char *) mmap(m_ptr, m_size, PROT_READ | PROT_WRITE, MAP_SHARED, m_fd, 0);
        if (m_ptr == MAP_FAILED) {
            MMKVError("fail to mmap [%s], %s", m mmapID.c_str(), strerror(errno));
        }

        // check if we fail to make more space
        if (!isValid()) {
            MMKWarning("[%s] file not valid", m mmapID.c_str());
            return false;
        }
    }

    // 加密数据
    if (m_crypter) {
        m_crypter->reset();
        auto ptr = (unsigned char *) data.getPtr();
        m_crypter->encrypt(ptr, ptr, data.length());
    }

    // 重新构建并写入数据
    writeAcutalSize(data.length());
    delete m_output;
    m_output = new CodedOutputData(m_ptr + offset, m_size - offset);
    m_output->writeRawData(data);
    recalculateCRCDigest();
    m_hasFullWriteback = true;
}

return true;
}

```

这里代码看起来也比较长，它对 MMKV 的内存重整进行了实现，步骤如下：

1. 当剩余映射空间不足以写入需要写入的内容，尝试进行内存重整
2. 内存重整会将文件清空，将 map 中的数据重新写入文件，从而去除冗余数据

- 若内存重整合后剩余映射空间仍然不足，不断将映射空间 double 直到足够，并用 mmap 重新映射

删除

通过 Java 层 MMKV 的 remove 方法可以实现删除操作：

```
@Override  
public Editor remove(String key) {  
    removeValueForKey(key);  
    return this;  
}
```

它调用了 removeValueForKey 这个 Native 方法：

```
extern "C" JNIRETURN JNICALL void Java_com_tencent_mmkv_MMKV_removeValueForKey  
(JNIEnv *env,  
                                     jobject  
                                     instance,  
                                     jlong  
                                     handle,  
                                     jstring  
                                     oKey) {  
    MMKV *kv = reinterpret_cast<MMKV *>(handle);  
    if (kv && oKey) {  
        string key = jstring2string(env, oKey);  
        kv->removeValueForKey(key);  
    }  
}
```

这里调用了 Native 层 MMKV 的 removeValueForKey 方法：

```
void MMKV::removeValueForKey(const std::string &key) {  
    if (key.empty()) {  
        return;  
    }  
    SCOPEDLOCK(m_lock);  
    SCOPEDLOCK(m_exclusiveProcessLock);  
    checkLoadData();
```

```
    removeDataForKey(key);
}
```

它在数据读入内存的前提下，调用了 `removeDataForKey` 方法：

```
bool MMKV::removeDataForKey(const std::string &key) {
    if (key.empty()) {
        return false;
    }
    auto deleteCount = m_dic.erase(key);
    if (deleteCount > 0) {
        m_hasFullWriteback = false;
        static MMBuffer nan(0);
        return appendDataWithKey(nan, key);
    }
    return false;
}
```

这里实际上是构造了一条 `size` 为 0 的 `MMBuffer` 并调用 `appendDataWithKey` 将其 `append` 到 `protobuf` 文件中，并将 `key` 对应的内容从 `map` 中删除。读取时发现它的 `size` 为 0，则会认为这条数据已经删除。

读取

我们通过 `getInt`、`getLong` 等操作可以实现对数据的读取，我们以 `getInt` 为例：

```
@Override
public int getInt(String key, int defValue) {
    return decodeInt(nativeHandle, key, defValue);
}
```

它调用到了 `decodeInt` 这个 Native 方法：

```
extern "C" JNICALL jint Java_com_tencent_mmkv_MMKV_decodeInt(
    JNIEnv *env, jobject obj, jlong handle, jstring oKey, jint defaultValue) {
    MMKV *kv = reinterpret_cast<MMKV *>(handle);
    if (kv && oKey) {
        string key = jstring2string(env, oKey);
        return (jint) kv->getInt32ForKey(key, defaultValue);
    }
}
```

```
    }
    return defaultValue;
}
```

它调用到了 MMKV.getInt32ForKey 方法：

```
int32_t MMKV::getInt32ForKey(const std::string &key, int32_t defaultValue) {
    if (key.empty()) {
        return defaultValue;
    }
    SCOPEDLOCK(m_lock);
    auto &data = getDataForKey(key);
    if (data.length() > 0) {
        CodedInputData input(data.getPtr(), data.length());
        return input.readInt32();
    }
    return defaultValue;
}
```

它首先调用了 getDataForKey 方法获取到了 key 对应的 MMBuffer，之后通过 CodedInputData 将数据读出并返回。可以发现，长度为 0 时会将其视为不存在，返回默认值。

```
const MMBuffer &MMKV::getDataForKey(const std::string &key) {
    checkLoadData();
    auto itr = m_dic.find(key);
    if (itr != m_dic.end()) {
        return itr->second;
    }
    static MMBuffer nan(0);
    return nan;
}
```

这里实际上是通过在 Map 中寻找从而实现，找不到会返回 size 为 0 的 Buffer。

文件回写

MMKV 中，在一些特定的情景下，会通过 fullWriteback 方法立即将 map 的内容回写到文件。

回写时机主要有以下几个：

1. 通过 MMKV.reKey 方法修改加密的 key。
2. 删 除一系列的 key 时（通过 removeValuesForKeys 方法）
3. 读取文件时文件校验或 CRC 校验失败。

```
bool MMKV::fullWriteback() {
    if (m_hasFullWriteback) {
        return true;
    }
    if (m_needLoadFromFile) {
        return true;
    }
    if (!isValid()) {
        MMKVWarning("[%s] file not valid", m mmapID.c_str());
        return false;
    }
    // 如果 map 空了，直接清空文件
    if (m_dic.empty()) {
        clearAll();
        return true;
    }
    // 将 m_dic 转换为对应的 MMBuffer
    auto allData = MiniPBCoder::encodeDataWithObject(m_dic);
    SCOPEDLOCK(m_exclusiveProcessLock);
    if (allData.length() > 0) {
        if (allData.length() + Fixed32Size <= m_size) {
            // 如果足够写入，直接写入
            if (m_crypter) {
                m_crypter->reset();
                auto ptr = (unsigned char *) allData.getPtr();
                m_crypter->encrypt(ptr, ptr, allData.length());
            }
            writeAcutalSize(allData.length());
            delete m_output;
            m_output = new CodedOutputData(m_ptr + Fixed32Size, m_size - Fixed32Size);
            m_output->writeRawData(allData); // note: don't write size of data
            recalculateCRCDigest();
            m_hasFullWriteback = true;
        }
        return true;
    }
}
```

```

    } else {
        // 如果剩余空间不够写入，调用 ensureMemorySize 从而进行内存重整与扩容
        return ensureMemorySize(allData.length() + Fixed32Size - m_size);
    }
}
return false;
}

```

这里首先在 map 为空的情况下，由于代表了所有数据已被删除，因此通过 `clearAll` 清除了文件与数据。

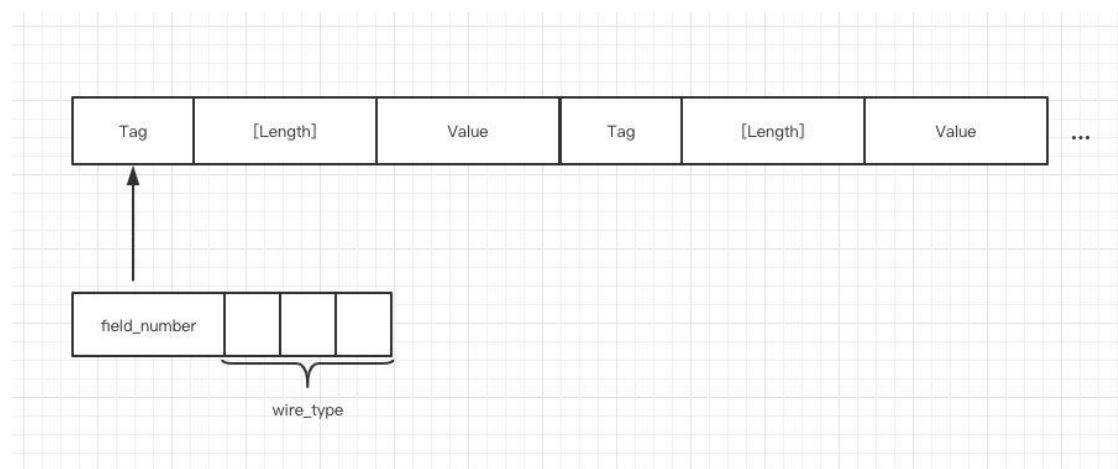
否则它会对当前映射空间是否足够写入 map 中回写的数据，如果足够则会将数据写入，否则会调用 `ensureMemorySize` 从而进行内存重整与扩容。

Protobuf 处理

Protobuf 编码

在我们开始对 Protobuf 部分代码进行研究前，让我们先研究一下 Protobuf 编码的格式。

Protobuf 采用了一种 TLV (Tag-Length-Value) 的格式进行编码，其格式如下：



可以看到，每条字段都由 Tag、Length、Value 三部分组成，其中 **Length** 是可选的。

Tag

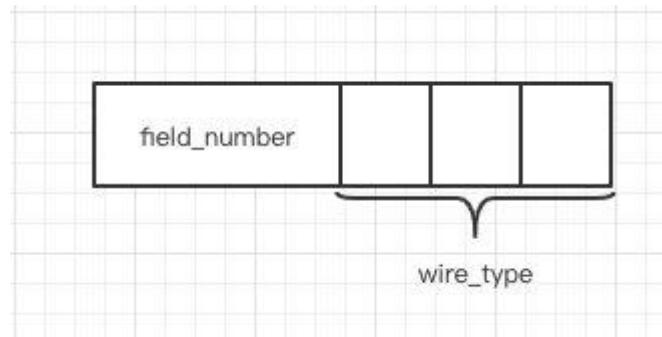


image-20200229101120808

Tag 由 field_number 和 wire_type 两部分组成，其中：

- field_number: 字段编号
- wire_type: protobuf 编码类型

并且 **Tag** 采用了 **Varints** 编码，它是一种可变长的 int 编码（类似 dex 文件的 LEB128）。

wire_type 共有 3 位，可以存放 8 种编码格式，目前已经实现了如下 6 种：

值	含义	用途
0	Varint	可变整型
1	64-bit	固定 64 位
2	Length-delimited	string、bytes 等

值	含义	用途
3	Start group (已废弃)	group 开始
4	End group (已废弃)	group 结束
5	32-bit	固定 32 位

可以发现，**Start group** 与 **End group** 已经废弃，对于 Length 这个字段，只有 Length-delimited 用到，其余的 Varint、64-bit、32-bit 等都不需要 Length 字段。

Varints 编码

Varints 编码是一种可变长的 int 编码，它的编码规则如下：

1. 第一位标明了是否需要读取下一字节
2. 存储了数值的补码，且低位在前高位在后。

解码过程

可以简单模拟一下解码的过程，我们接收到一串二进制数据，我们可以先读取一个 Varints 编码块，其后面 3 位为 wire_type，而前面的表示 field_number。之后它会根据 wire_type 来决定是根据 Length 读取固定大小的 Value 还是采用 Varint 等方式读取后面的 Value。

Protobuf 实现

在 MMKV 中通过 `MiniPBCoder` 完成了 Protobuf 的序列化及反序列化。我们可以通过 `MiniPBCoder::decodeMap` 将 MMKV 存储的 protobuf 文件反序列化为

对应的 Map，可以通过 `MiniPBCoder::encodeDataWithObject` 将 Map 序列化为对应存储的字节流。

序列化

我们先看看它是如何完成序列化的过程的：

```
static MMBuffer encodeDataWithObject(const T &obj) {
    MiniPBCoder pbcoder;
    return pbcoder.getEncodeData(obj);
}
```

它调用到了 `getEncodeData` 方法，并传入了对应的 Map：

```
MMBuffer MiniPBCoder::getEncodeData(const unordered_map<string, MMBuffer> &map)
{
    m_encodeItems = new vector<PBEncodeItem>();
    // 准备 PBEncodeItem 数组
    size_t index = prepareObjectForEncode(map);
    PBEncodeItem *oItem = (index < m_encodeItems->size()) ? &(*m_encodeItems)
[index] : nullptr;
    if (oItem && oItem->compiledSize > 0) {
        m_outputBuffer = new MMBuffer(oItem->compiledSize);
        m_outputData = new CodedOutputData(m_outputBuffer->getPtr(), m_outputBuffer->length());
        writeRootObject();
    }
    return std::move(*m_outputBuffer);
}
```

可以看到，它首先通过 `prepareObjectForEncode` 方法将 Map 中的键值对转为了对应的 `PBEncodeItem` 对象数组，之后构造了对应的用于写入的 `CodedOutputData` 以及写入的 `m_outputBuffer`，然后调用了 `writeRootObject` 方法将数据通过 `CodedOutputData` 写入到 `m_outputBuffer` 中。

PBEncodeItem 数组的准备

我们先看到 `prepareObjectForEncode` 方法:

```
size_t MiniPBCoder::prepareObjectForEncode(const unordered_map<string, MMBuffer> &map) {
    // 放入一个新的 EncodeItem
    m_encodeItems->push_back(PBEncodeItem());
    // 获取刚刚的 Item 以及其对应的 index
    PBEncodeItem *encodeItem = &(m_encodeItems->back());
    size_t index = m_encodeItems->size() - 1;
{
    // 将该 EncodeItem 作为一个 Container
    encodeItem->type = PBEncodeItemType_Container;
    encodeItem->value.strValue = nullptr;
    // 遍历 Map
    for (const auto &itr : map) {
        const auto &key = itr.first;
        const auto &value = itr.second;
        if (key.length() <= 0) {
            continue;
        }
        // 将 key 作为一个 EncodeItem 放入数组
        size_t keyIndex = prepareObjectForEncode(key);
        if (keyIndex < m_encodeItems->size()) {
            // 将 value 作为一个 EncodeItem 放入数组
            size_t valueIndex = prepareObjectForEncode(value);
            if (valueIndex < m_encodeItems->size()) {
                // 计算 container 添加 key 和 value 后的 size
                (*m_encodeItems)[index].valueSize += (*m_encodeItems)[keyIndex].compiledSize;
                (*m_encodeItems)[index].valueSize += (*m_encodeItems)[valueIndex].compiledSize;
            } else {
                m_encodeItems->pop_back(); // pop key
            }
        }
        encodeItem = &(*m_encodeItems)[index];
    }
    encodeItem->compiledSize = pbRawVarint32Size(encodeItem->valueSize) + encodeItem->valueSize;
```

```
    return index;  
}
```

可以看到，这里实际上会首先在 `m_encodeItems` 数组中先放入一个作为 Container 的 `PBEncodeItem`，之后遍历 Map，对每个 Key 和 Value 分别构建对应的 `PBEncodeItem` 并放入，并且将其 size 计算入 Container 的 `valueSize`。最后会返回该 Container 的 index。

对于 Key 其会写入一个 String 类型的 `PBEncodeItem`

对于 Value 其会写入一个 Data 类型存储 `MMBuffer` 的 `PBEncodeItem`。

将数据写入 `MMBuffer`

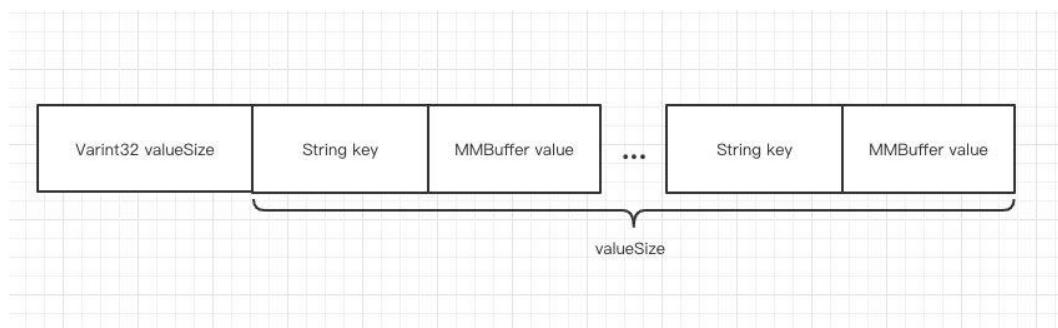
接着我们看看它是如何实现将数据写入的，我们看到 `writeRootObject` 方法：

```
void MiniPBCoder::writeRootObject() {  
    for (size_t index = 0, total = m_encodeItems->size(); index < total; index++) {  
        PBEncodeItem *encodeItem = &(*m_encodeItems)[index];  
        switch (encodeItem->type) {  
            case PBEncodeItemType_String: {  
                m_outputData->writeString(*(encodeItem->value.strValue));  
                break;  
            }  
            case PBEncodeItemType_Data: {  
                m_outputData->writeData(*(encodeItem->value.bufferValue));  
                break;  
            }  
            case PBEncodeItemType_Container: {  
                m_outputData->writeRawVarint32(encodeItem->valueSize);  
                break;  
            }  
            case PBEncodeItemType_None: {  
                MMKVEError("%d", encodeItem->type);  
                break;  
            }  
        }  
    }  
}
```

```
    }  
}
```

这里的实现非常简单，根据是 String 类型还是 Data 类型还是 Container 类型，分别写入 String、MMBuffer 以及 Varint32。其中 Container 写入的就是后面的 size 大小。

因此写入到文件后文件最后的格式如下：



反序列化

我们可以通过 MiniPBCoder.decodeMap 将其反序列化为 Map，我们可以看看它是如何实现的：

```
void MiniPBCoder::decodeMap(unordered_map<string, MMBuffer> &dic,  
                           const MMBuffer &oData,  
                           size_t size) {  
    MiniPBCoder oCoder(&oData);  
    oCoder.decodeOneMap(dic, size);  
}
```

它调用到了 decodeOnMap 方法：

```
void MiniPBCoder::decodeOneMap(unordered_map<string, MMBuffer> &dic, size_t size) {  
    if (size == 0) {  
        auto length = m_inputData->readInt32();  
    }  
    while (!m_inputData->isAtEnd()) {
```

```
const auto &key = m_inputData->readString();
if (key.length() > 0) {
    auto value = m_inputData->readData();
    if (value.length() > 0) {
        dic[key] = move(value);
    } else {
        dic.erase(key);
    }
}
}
```

可以看到，它的实现非常简单，先读取了一个 Varint32 的 valueSize，之后不断通过 CodedInputStream 分别读取 key 和 value，这对我们前面的猜想进行了印证，并且当遇到 Length 为 0 的 value 时，会将对应的项删掉。

跨进程锁实现

本部分主要参考自官方文档：[MMKV for Android 多进程设计与实现](#)

跨进程锁的选择

SharedPreferences 在 Android 7.0 之后便不再对跨进程模式进行支持，原因是跨进程无法保证线程安全，而 MMKV 则通过了文件锁解决了这个问题。

其实本来是可以采用在共享内存中创建 `pthread_mutex` 实现两端的线程同步，但由于 Android 对 Linux 的部分机制进行了阉割，它无法保证获取锁的进程被杀死后，系统会对锁的信息进行清理。这就会导致等待锁的进程饿死。

因此 MMKV 采用了文件锁的设计，它的缺点在于不支持递归加锁，不支持锁的升级/降级，因此 MMKV 自行对这两个功能进行了实现。

文件锁

文件锁是 Linux 中基于文件实现的跨进程锁，我们需要维护一个 `flock` 结构体，它的结构如下：

```
struct flock {
    short l_type; /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence; /* How to interpret l_start: SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start; /* Starting offset for lock */
    off_t l_len; /* Number of bytes to lock */
    pid_t l_pid; /* PID of process blocking our lock (F_GETLK only) */
};
```

其中我们重点关注 `l_type`，它表达了锁的类型，它有三种状态：

- `F_RDLCK`: 也就是读锁，是一种共享锁
- `F_WRLCK`: 也就是写锁，是一种互斥锁
- `F_UNLCK`: 也就是无锁，代表要对其进行解锁

我们通过 `fcntl` 函数可以提交对 `flock` 的修改：

```
int fcntl(int fd, int cmd, struct flock lock)
```

其中 `fd` 也就是文件描述符，`cmd` 表达了要进行的操作，`flock` 表示 `flock` 结构体，它里面包含了对锁进行操作的类型。

`cmd` 有以下三种取值：

- `F_GETLK`: 获取文件锁
- `F_SETLK`: 设置文件锁（非阻塞），设置不成功直接返回

- `F_SETLKW`: 设置文件锁（阻塞），阻塞等到设置成功

文件锁存在着一定缺点：

1. 不支持递归加锁（重入锁）：如果我们重复加锁会导致阻塞，如果我们解锁会把所有的锁都给解除。
2. 存在着死锁问题：如果我们两个进程同时将读锁升级为死锁，可能会陷入互相等待从而发生死锁。

文件锁封装

MMKV 中对文件锁的**递归锁和锁升级/降级**机制进行了实现。

- 递归锁（可重入）

若一个进程/线程已经拥有了锁，那么后续的加锁操作不会导致卡死，并且解锁也不会导致外层的锁被解掉。由于文件锁是基于状态的，没有计数器，因此在解锁时会导致外层的锁也被解掉。

- 锁升级/降级

锁升级是指将已经持有的共享锁，升级为互斥锁，也就是将读锁升级为写锁，锁降级则是反过来。文件锁支持锁升级，但是容易死锁：假如 A、B 进程都持有了读锁，现在都想升级到写锁，就会发生死锁。另外，由于文件锁不支持递归锁，也导致了锁降级一降就降到没有锁。

MMKV 中基于文件锁实现了上述的递归锁以及锁的升级、降级功能。

加锁

调用 `FileLock.lock` 或 `FileLock.try_lock` 方法会调用到 `FileLock.doLock` 方法，他们两者的区别是前者是阻塞式获取锁，会等待到锁的释放，后者则是非阻塞式获取锁。在 `FileLock.doLock` 中完成了锁的获取：

```

bool FileLock::doLock(LockType lockType, int cmd) {
    bool unLockFirstIfNeeded = false;
    // 加读锁（共享锁）
    if (lockType == SharedLockType) {
        // 读锁数量++
        m_sharedLockCount++;
        // 有其他锁的情况下，不需要真正再加一次锁
        if (m_sharedLockCount > 1 || m_exclusiveLockCount > 0) {
            return true;
        }
    } else {
        m_exclusiveLockCount++;
        // 之前加过写锁，则不需要再重新加锁
        if (m_exclusiveLockCount > 1) {
            return true;
        }
        // 要加写锁，如果已经存在读锁，可能是其他进程获取的，如果是则需要先将自己的读
        // 锁释放掉，再加写锁
        if (m_sharedLockCount > 0) {
            unLockFirstIfNeeded = true;
        }
    }
}

```

```

// 加读锁或写锁获取到的锁类型 F_RDLCK 或 F_WRLCK
m_lockInfo.l_type = LockType2FlockType(lockType);
if (unLockFirstIfNeeded) {
    // 如果已经存在读锁，先看看能否获取写锁
    auto ret = fcntl(m_fd, F_SETLK, &m_lockInfo);
    if (ret == 0) {
        return true;
    }
    // 不能获取写锁说明其他线程获取了读锁，则将自己的读锁释放避免死锁
    auto type = m_lockInfo.l_type;
    // 执行解锁
    m_lockInfo.l_type = F_UNLCK;
    ret = fcntl(m_fd, F_SETLK, &m_lockInfo);
    if (ret != 0) {
        MMKVError("fail to try unlock first fd=%d, ret=%d, error:%s", m_fd,
ret,
                     strerror(errno));
    }
    m_lockInfo.l_type = type;
}
// 执行对应的加锁（读锁或写锁）

```

```

        auto ret = fcntl(m_fd, cmd, &m_lockInfo);
        if (ret != 0) {
            MMKVError("fail to lock fd=%d, ret=%d, error:%s", m_fd, ret, strerror(errno));
            return false;
        } else {
            return true;
        }
    }
}

```

可以看到，上面的步骤对于写锁而言，在加写锁时，如果当前进程持有了读锁，那我们需要尝试加写锁。如果加写锁失败说明其他线程持有了读锁，我们需要将目前的读锁释放掉，再加写锁，从而避免死锁（这种情况说明两个进程的读锁都想升级为写锁）。

同时可以发现，MMKV 中通过维护了 `m_sharedLockCount` 以及 `m_exclusiveLockCount` 从而实现了递归加锁，如果存在其他锁时，就不再需要真正第二次加锁了。

解锁

通过 `FileLock.unlock` 可以完成对锁的解锁：

```

bool FileLock::unlock(LockType lockType) {
    bool unlockToSharedLock = false;
    if (lockType == SharedLockType) {
        if (m_sharedLockCount == 0) {
            return false;
        }
        m_sharedLockCount--;
        // 解读锁，只需要减少 count 即可，如果此时存在其他的锁就不需要真正解锁了
        if (m_sharedLockCount > 0 || m_exclusiveLockCount > 0) {
            return true;
        }
    } else {
        if (m_exclusiveLockCount == 0) {
            return false;
        }
        // 解写锁
    }
}

```

```

        m_exclusiveLockCount--;
        if (m_exclusiveLockCount > 0) {
            return true;
        }
        // 如果之前我们是存在写锁的，则只是降级为读锁，因为我们之前将读锁升级为了写锁
        if (m_sharedLockCount > 0) {
            unlockToSharedLock = true;
        }
    }
    m_lockInfo.l_type = static_cast<short>(unlockToSharedLock ? F_RDLCK : F_UNLCK);
    auto ret = fcntl(m_fd, F_SETLK, &m_lockInfo);
    if (ret != 0) {
        MMKVEError("fail to unlock fd=%d, ret=%d, error:%s", m_fd, ret, strerror(errno));
        return false;
    } else {
        return true;
    }
}

```

在解锁时，对于解写锁时，如果我们的写锁是由读锁升级而来，则不会真的进行解锁，而是改为加读锁，从而实现将写锁降级为读锁（因为读锁还没解除）。

状态同步

跨进程共享 MMKV 文件面临着状态同步问题：写指针同步、内存重整同步、内存增长同步。

写指针同步：其他进程可能写入了新的键值，此时需要更新写指针的位置。它通过在文件头部保存了有效内存的大小 `m_actualSize`，每次都对其进行比较从而实现写指针的同步。

内存重整同步：如果发生了内存重整，可能导致前面的键值全部失效，需要全部抛弃重新加载。为了实现内存重整同步，是通过使用一个单调递增的序列号 `m_sequence` 进行比较，每进行一次内存重整将其 + 1 从而实现。

内存增长同步：通过文件大小的比较从而实现。

MMKV 中的状态同步通过 `checkLoadData` 方法实现：

```

void MMKV::checkLoadData() {
    if (m_needLoadFromFile) {
        SCOPEDLOCK(m_sharedProcessLock);
        m_needLoadFromFile = false;
        loadFromFile();
        return;
    }
    if (!m_isInterProcess) {
        return;
    }
    // TODO: atomic lock m_metaFile?
    MMKVMetaInfo metaInfo;
    metaInfo.read(m_metaFile.getMemory());
    if (m_metaInfo.m_sequence != metaInfo.m_sequence) {
        // 序列号不同, 说明发生了内存重整, 清空后重新加载
        MMKVInfo("[%s] oldSeq %u, newSeq %u", m_mmapID.c_str(), m_metaInfo.m_sequence,
                 metaInfo.m_sequence);
        SCOPEDLOCK(m_sharedProcessLock);

        clearMemoryState();
        loadFromFile();
    } else if (m_metaInfo.m_crcDigest != metaInfo.m_crcDigest) {
        // CRC 不同, 说明发生了改变
        MMKVDebug("[%s] oldCrc %u, newCrc %u", m_mmapID.c_str(), m_metaInfo.m_crcDigest,
                  metaInfo.m_crcDigest);
        SCOPEDLOCK(m_sharedProcessLock);
        size_t fileSize = 0;
        if (m_isAshmem) {
            fileSize = m_size;
        } else {
            struct stat st = {0};
            if (fstat(m_fd, &st) != -1) {
                fileSize = (size_t) st.st_size;
            }
        }
        if (m_size != fileSize) {
            // 如果 size 相同, 说明发生了文件增长
            MMKVInfo("file size has changed [%s] from %zu to %zu", m_mmapID.c_str(),
                     m_size, fileSize);
            clearMemoryState();
            loadFromFile();
        }
    }
}

```

```
        } else {
            // size 相同, 说明需要进行写指针同步, 只需要部分进行 LoadFile
            partialLoadFromFile();
        }
    }
}
```

可以看到，除了写指针同步的情况，其余情况都是重新读取文件实现同步。

总结

MMKV 是一个基于 `mmap` 实现的 K-V 存储工具，它的序列化基于 `protobuf` 实现，引入了 `CRC` 校验从而对文件完整性进行校验，并且它支持了通过 `AES` 算法对 `protobuf` 文件进行加密。

- MMKV 的初始化过程主要完成了对 `rootDir` 的初始化及创建，它位于应用的内部存储 `file` 下的 `mmkv` 文件夹。
- MMKV 的获取需要通过 `mmapWithID` 完成，它会结合传入的 `mmapId` 与 `relativePath` 通过 `md5` 生成一个唯一的 `mmapKey`，通过它查找 `map` 获取对应的 MMKV 实例，若找不到对应的实例会构建一个新的 MMKV 对象。Java 层通过持有 Native 层对象的地址从而实现与 Native 对象进行通信。
- 在 MMKV 对象创建时，会创建用于 AES 加密的 `AESCrypt` 对象，并且会调用 `loadFromFile` 方法将文件的内容通过 `mmap` 映射到内存中，映射会以页的整数倍进行，若不足的地方会补 0。映射完成后会构造对应的 `MMBuffer` 对映射区域进行管理并创建对应的 `CodedOutputData` 对象，之后会通过 `MiniPBCoder` 将其读入到 `_dic` 这个 Map 中，它以 `String` 为 key，`MMBuffer` 为 value。

- MMKV 在数据写入前会调用 `checkLoadData` 方法确保数据已读入并且对跨进程的信息进行同步，之后会将数据转换为 `MMBuffer` 对象并写入 `map` 中，然后调用 `ensureMemorySize` 确保映射空间足够的情况下，通过 构造 MMKV 对象时创建的 `CodedOutputData` 将数据写入 `protobuf` 文件。并且 MMKV 的数据更新和写入都是通过在文件后进行 `append`，会造成存在冗余 `key-value` 数据。
- `ensureMemorySize` 方法在内存不足的情况下首先进行内存重整，它会清空文件，从 `map` 重新将数据写入文件，从而清理冗余数据，如果仍然不够则会以每次两倍对文件大小进行扩容，并重新通过 `mmap` 进行映射。
- MMKV 的删除操作实际上是在文件中对同样的 `key` 写入长度为 0 的 `MMBuffer` 实现，当读取时发现其长度为 0，则将其视为已删除。
- MMKV 的读取是通过 `CodedInputStream` 实现，它在读如的 `MMBuffer` 长度为 0 时会将其视为不存在。实际上 `CodedInputStream` 与 `CodedOutputStream` 就是与 `MMBuffer` 进行交互的桥梁。
- MMKV 还存在着文件回写机制，在以下的时机会将 `map` 中的数据立即写入文件，空间不足则会进行内存重整：
 1. 通过 `MMKV.reKey` 方法修改加密的 `key`。
 2. 删除一系列的 `key` 时（通过 `removeValuesForKeys` 方法）
 3. 读取文件时文件校验或 CRC 校验失败。
- MMKV 对跨进程读写进行了支持，它通过文件锁实现跨进程加锁，并且通过对文件锁引入读锁和写锁的计数，从而解决了其存在的不支持递归锁和锁升级/降级问题。不使用 `pthread_mutex` 通过共享内存加锁的原因是 Android 对

Linux 进行了阉割，如果持有锁的进程被杀死无法保证清除锁的信息，可能导致等待锁的其他进程饿死。

- 加写锁时，如果当前进程持有了读锁，那我们需要尝试将其升级为写锁。如果升级写锁失败说明其他线程持有了读锁，我们需要将当前进程的读锁释放掉，再加写锁，从而避免死锁（这种情况说明两个进程的读锁都想升级为写锁）。
- 解写锁时，如果我们的写锁是由读锁升级而来，则不会真的进行解锁，而是改为加读锁，从而实现将写锁降级为读锁（因为读锁还没解除）。
- MMKV 解决了写指针同步、内存重整同步以及内存增长同步问题，写指针同步通过在文件的起始处添加一个写指针值，在 `checkLoadData` 中会对它进行比较，从而获取最新的写指针 `m_actualSize`，而内存重整同步通过一个序号 `m_sequence` 来实现，每当发生一次内存重整对其 + 1，通过比较即可确定。而内存增长同步则通过比较文件大小实现。

2. 深入解析阿里巴巴路由框架 ARouter 源码

ARouter 是阿里推出的一款页面路由框架。由于项目中采用了组件化架构进行开发，通过 ARouter 实现了页面的跳转，之前看它的源码时忘了写笔记，因此今天来重新对它的源码进行一次分析。

本篇源码解析基于 ARouter 1.2.4

初始化

ARouter 在使用前需要通过调用 `Arouter.init` 方法并传入 `Application` 进行初始化：

```
/**
```

```
* Init, it must be call before used router.  
*/  
public static void init(Application application) {  
    if (!hasInit) {  
        logger = _ARouter.logger;  
        _ARouter.logger.info(Consts.TAG, "ARouter init start.");  
        hasInit = _ARouter.init(application);  
        if (hasInit) {  
            _ARouter.afterInit();  
        }  
        _ARouter.logger.info(Consts.TAG, "ARouter init over.");  
    }  
}
```

这里调用到了 `_ARouter.init`, 这个 `_ARouter` 类才是 `ARouter` 的核心类:

```
protected static synchronized boolean init(Application application) {  
    mContext = application;  
    LogisticsCenter.init(mContext, executor);  
    logger.info(Consts.TAG, "ARouter init success!");  
    hasInit = true;  
    // It's not a good idea.  
    // if (Build.VERSION.SDK_INT > Build.VERSION_CODES.ICE_CREAM_SANDWICH) {  
    //     application.registerActivityLifecycleCallbacks(new AutowiredLifecycleCallback());  
    // }  
    return true;  
}
```

这里实际上调用到了 `LogisticsCenter.init`:

```
public synchronized static void init(Context context, ThreadPoolExecutor tpe)  
throws HandlerException {  
    mContext = context;  
    executor = tpe;  
    try {  
        long startInit = System.currentTimeMillis();  
        Set<String> routerMap;  
        // 获取存储 ClassName 集合的 routerMap (debug 模式下每次都会拿最新的)  
        if (ARouter.debuggable() || PackageUtils.isNewVersion(context)) {  
            logger.info(TAG, "Run with debug mode or new install, rebuild router map.");  
            // 根据指定的 packageName 获取 package 下的所有 ClassName
```

```
        routerMap = ClassUtils.getFileNameByPackageName(mContext, ROUTE_ROOT_PACKAGE);
        if (!routerMap.isEmpty()) {
            // 存入 SP 缓存
            context.getSharedPreferences(AROUTER_SP_CACHE_KEY, Context.MODE_PRIVATE).edit().putStringSet(AROUTER_SP_KEY_MAP, routerMap).apply();
        }
    } else {
        logger.info(TAG, "Load router map from cache.");
        // release 模式下, 已经缓存了 ClassName 列表
        routerMap = new HashSet<>(context.getSharedPreferences(AROUTER_SP_CACHE_KEY, Context.MODE_PRIVATE).getStringSet(AROUTER_SP_KEY_MAP, new HashSet<String>()));
    }
    logger.info(TAG, "Find router map finished, map size = " + routerMap.size() + ", cost " + (System.currentTimeMillis() - startInit) + " ms.");
    startInit = System.currentTimeMillis();
    // 遍历 ClassName
    for (String className : routerMap) {
        if (className.startsWith(ROUTE_ROOT_PACKAGE + DOT + SDK_NAME + SEPARATOR + SUFFIX_ROOT)) {
            // 发现是 Root, 加载类构建对象后通过 LoadInto 加载进 Warehouse.groupsIndex
            ((IRouteRoot) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.groupsIndex);
        } else if (className.startsWith(ROUTE_ROOT_PACKAGE + DOT + SDK_NAME + SEPARATOR + SUFFIX_INTERCEPTORS)) {
            // 发现是 Interceptor, 加载类构建对象后通过 LoadInto 加载进 Warehouse.interceptorsIndex
            ((IInterceptorGroup) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.interceptorsIndex);
        } else if (className.startsWith(ROUTE_ROOT_PACKAGE + DOT + SDK_NAME + SEPARATOR + SUFFIX_PROVIDERS)) {
            // 发现是 ProviderGroup, 加载类构建对象后通过 LoadInto 加载进 Warehouse.providersIndex
            ((IProviderGroup) (Class.forName(className).getConstructor().newInstance())).loadInto(Warehouse.providersIndex);
        }
    }
    // ...
} catch (Exception e) {
    throw new HandlerException(TAG + "ARouter init logistics center exception! [" + e.getMessage() + "]");
}
```

```
}
```

这里主要有如下几步：

1. 获取 com.alibaba.android.arouter.routes 下存储 ClassName 的集合 routerMap。
2. 若为 debug 模式或之前没有解析过 routerMap，则通过 ClassUtils.getFileNameByPackageName 方法对指定 package 下的所有 ClassName 进行解析并存入 SP。
(debug 每次都需要更新，因为类会随着代码的修改而变动)
3. 若并非 debug 模式，并且之前已经解析过，则直接从 SP 中取出。
4. 遍历 routerMap 中的 ClassName。

如果是 RouteRoot，则加载类构建对象后通过 loadInto 加载进 Warehouse.groupsIndex。

- a. 如果是 InterceptorGroup，则加载类构建对象后通过 loadInto 加载进 Warehouse.interceptorsIndex。
- b. 如果是 ProviderGroup，则加载类构建对象后通过 loadInto 加载进 Warehouse.providersIndex。

解析 ClassName

我们先看看 ClassUtils.getFileNameByPackageName 是如何对指定 package 下的 ClassName 集合进行解析的：

```
public static Set<String> getFileNameByPackageName(Context context, final String packageName) {
```

```
final Set<String> classNames = new HashSet<>();
// 通过 getSourcePaths 方法获取 dex 文件 path 集合
List<String> paths = getSourcePaths(context);
// 通过 CountDownLatch 对 path 的遍历处理进行控制
final CountDownLatch parserCtl = new CountDownLatch(paths.size());
// 遍历 path, 通过 DefaultPoolExecutor 并发对 path 进行处理
for (final String path : paths) {
    DefaultPoolExecutor.getInstance().execute(new Runnable() {
        @Override
        public void run() {
            // 加载 path 对应的 dex 文件
            DexFile dexfile = null;
            try {
                if (path.endsWith(EXTRACTED_SUFFIX)) {
                    // zip 结尾通过 DexFile.LoadDex 进行加载
                    dexfile = DexFile.loadDex(path, path + ".tmp", 0);
                } else {
                    // 否则通过 new DexFile 加载
                    dexfile = new DexFile(path);
                }
                // 遍历 dex 中的 Entry
                Enumeration<String> dexEntries = dexfile.entries();
                while (dexEntries.hasMoreElements()) {
                    // 如果是对应的 package 下的类, 则添加其 className
                    String className = dexEntries.nextElement();
                    if (className.startsWith(packageName)) {
                        classNames.add(className);
                    }
                }
            } catch (Throwable ignore) {
                Log.e("ARouter", "Scan map file in dex files made error.", ignore);
            } finally {
                if (null != dexfile) {
                    try {
                        dexfile.close();
                    } catch (Throwable ignore) {
                    }
                }
            }
            parserCtl.countDown();
        }
    });
}
});
```

```
// 所有 path 处理完成后，继续向下走
parserCtl.await();
Log.d(Consts.TAG, "Filter " + classNames.size() + " classes by packageName
<" + packageName + ">");
return classNames;
}
```

这里的步骤比较简单，主要是如下的步骤：

1. 通过 `getSourcePaths` 方法获取 `dex` 文件的 `path` 集合。
2. 创建了一个 `CountDownLatch` 控制 `dex` 文件的并行处理，以加快速度。
3. 遍历 `path` 列表，通过 `DefaultPoolExecutor` 对 `path` 并行处理。
4. 加载 `path` 对应的 `dex` 文件，并对其中的 `Entry` 进行遍历，若发现了对应 `package` 下的 `ClassName`，将其加入结果集合。
5. 所有 `dex` 处理完成后，返回结果。

关于 `getSourcePaths` 如何获取到的 `dex` 集合这里就不纠结了，因为我们的关注点不在这里。

初始化 Warehouse

`Warehouse` 实际上就是仓库的意思，它存放了 `ARouter` 自动生成的类(`RouteRoot`、`InterceptorGroup`、`ProviderGroup`)的信息。

我们先看看 `Warehouse` 类究竟是怎样的：

```
class Warehouse {
    // 保存 RouteGroup 对应的 class 以及 RouteMeta
    static Map<String, Class<? extends IRouteGroup>> groupsIndex = new HashMap
<>();
    static Map<String, RouteMeta> routes = new HashMap<>();
    // 保存 Provider 以及 RouteMeta
    static Map<Class, IProvider> providers = new HashMap<>();
```

```

    static Map<String, RouteMeta> providersIndex = new HashMap<>();
    // 保存 Interceptor 对应的 class 以及 Interceptor
    static Map<Integer, Class<? extends IInterceptor>> interceptorsIndex = new
    UniqueKeyTreeMap<>("More than one interceptors use same priority [%s]");
    static List<IInterceptor> interceptors = new ArrayList<>();
    static void clear() {
        routes.clear();
        groupsIndex.clear();
        providers.clear();
        providersIndex.clear();
        interceptors.clear();
        interceptorsIndex.clear();
    }
}

```

可以发现 `Warehouse` 就是一个纯粹用来存放信息的仓库类，它的数据的实际上是由上面的几个自动生成的类在 `loadInto` 中对 `Warehouse` 主动填入数据实现的。

例如我们打开一个自动生成的 `IRouteRoot` 的实现类：

```

public class ARouter$$Root$$homework implements IRouteRoot {
    @Override
    public void loadInto(Map<String, Class<? extends IRouteGroup>> routes) {
        routes.put("homework", ARouter$$Group$$homework.class);
    }
}

```

可以看到，它在 `groupsIndex` 中对这个 `RouteRoot` 中的 `IRouteGroup` 进行了注册，也就是向 `groupIndex` 中注册了 `Route Group` 对应的 `IRouteGroup` 类。其他类也是一样，通过自动生成的代码将数据填入 `Map` 或 `List` 中。

可以发现，初始化过程主要完成了对自动生成的路由相关类 `RouteRoot`、`Interceptor`、`ProviderGroup` 的加载，对它们通过反射构造后将信息加载进了 `Warehouse` 类中。

路由跳转

Postcard 的创建

下面我们看看路由的跳转是如何实现的，我们先看到 `ARouter.build` 方法：

```
public Postcard build(String path) {
    return _ARouter.getInstance().build(path);
}
```

它转调到了 `_ARouter` 的 `build` 方法：

```
protected Postcard build(String path) {
    if (TextUtils.isEmpty(path)) {
        throw new HandlerException(Consts.TAG + "Parameter is invalid!");
    } else {
        PathReplaceService pService = ARouter.getInstance().navigation(PathReplaceService.class);
        if (null != pService) {
            path = pService.forString(path);
        }
        return build(path, extractGroup(path));
    }
}
```

它首先通过 `ARouter.navigation` 获取到了 `PathReplaceService`，它需要用户进行实现，若没有实现会返回 `null`，若有实现则调用了它的 `forString` 方法传入了用户的 `Route Path` 进行路径的预处理。

最后转调到了 `build(path, group)`，`group` 通过 `extractGroup` 得到：

```
private String extractGroup(String path) {
    if (TextUtils.isEmpty(path) || !path.startsWith("/")) {
        throw new HandlerException(Consts.TAG + "Extract the default group failed, the path must be start with '/' and contain more than 2 '/'!");
    }
    try {
```

```

        String defaultGroup = path.substring(1, path.indexOf("/", 1));
        if (TextUtils.isEmpty(defaultGroup)) {
            throw new HandlerException(Consts.TAG + "Extract the default group
failed! There's nothing between 2 '/'!");
        } else {
            return defaultGroup;
        }
    } catch (Exception e) {
        logger.warning(Consts.TAG, "Failed to extract default group! " + e.getM
essage());
        return null;
    }
}

```

`extractGroup` 实际上就是对字符串处理，取出 Route Group 的名称部分。

```

protected Postcard build(String path, String group) {
    if (TextUtils.isEmpty(path) || TextUtils.isEmpty(group)) {
        throw new HandlerException(Consts.TAG + "Parameter is invalid!");
    } else {
        PathReplaceService pService = ARouter.getInstance().navigation(PathRep
laceService.class);
        if (null != pService) {
            path = pService.forString(path);
        }
        return new Postcard(path, group);
    }
}

```

`build(path, group)` 方法同样也会尝试获取到 `PathReplaceService` 并对 `path` 进行预处理。之后通过 `path` 与 `group` 构建了一个 `Postcard` 类：

```

public Postcard(String path, String group) {
    this(path, group, null, null);
}
public Postcard(String path, String group, Uri uri, Bundle bundle) {
    setPath(path);
    setGroup(group);
    setUri(uri);
    this.mBundle = (null == bundle ? new Bundle() : bundle);
}

```

这里最终调用到了 PostCard(path, group, uri, bundle)，这里只是进行了一些参数的设置。

之后，如果我们调用 `withInt`、`withDouble` 等方法，就可以进行参数的设置。例如 `withInt` 方法：

```
public Postcard withInt(@Nullable String key, int value) {
    mBundle.putInt(key, value);
    return this;
}
```

它实际上就是在对 `Bundle` 中设置对应的 `key`、`value`。

最后我们通过 `navigation` 即可实现最后的跳转：

```
public Object navigation() {
    return navigation(null);
}
public Object navigation(Context context) {
    return navigation(context, null);
}
public Object navigation(Context context, NavigationCallback callback) {
    return ARouter.getInstance().navigation(context, this, -1, callback);
}
public void navigation(Activity mContext, int requestCode) {
    navigation(mContext, requestCode, null);
}
public void navigation(Activity mContext, int requestCode, NavigationCallback
callback) {
    ARouter.getInstance().navigation(mContext, this, requestCode, callback);
}
```

通过如上的 `navigation` 可以看到，实际上它们都是最终调用到 `ARouter.navigation` 方法，在没有传入 `Context` 时会使用 `Application` 初始化的 `Context`，并且可以通过 `NavigationCallback` 对 `navigation` 的过程进行监听。

```
public Object navigation(Context mContext, Postcard postcard, int requestCode,
    NavigationCallback callback) {
    return _ARouter.getInstance().navigation(mContext, postcard, requestCode,
        callback);
}
```

ARouter 仍然只是将请求转发到了 _ARouter:

```
protected Object navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallback callback) {
    try {
        // 通过 LogisticsCenter.completion 对 postcard 进行补全
        LogisticsCenter.completion(postcard);
    } catch (NoRouteNotFoundException ex) {
        // ...
    }
    if (null != callback) {
        callback.onFound(postcard);
    }
    // 如果设置了 greenChannel, 会跳过所有拦截器的执行
    if (!postcard.isGreenChannel()) {
        // 没有跳过拦截器, 对 postcard 的所有拦截器进行执行
        interceptorService.doInterceptions(postcard, new InterceptorCallback()
    {
        @Override
        public void onContinue(Postcard postcard) {
            _navigation(context, postcard, requestCode, callback);
        }
        @Override
        public void onInterrupt(Throwable exception) {
            if (null != callback) {
                callback.onInterrupt(postcard);
            }
            logger.info(Consts.TAG, "Navigation failed, termination by interceptor : " + exception.getMessage());
        }
    });
} else {
    return _navigation(context, postcard, requestCode, callback);
}
return null;
```

上面的代码主要有以下步骤:

1. 通过 `LogisticsCenter.completion` 对 `postcard` 进行补全。
2. 如果 `postcard` 没有设置 `greenChannel`, 则对 `postcard` 的拦截器进行执行, 执行完成后调用 `_navigation` 方法真正实现跳转。
3. 如果 `postcard` 设置了 `greenChannel`, 则直接跳过所有拦截器, 调用 `_navigation` 方法真正实现跳转。

Postcard 的补全

我们看看 `LogisticsCenter.completion` 是如何实现 `postcard` 的补全的:

```
public synchronized static void completion(Postcard postcard) {  
    if (null == postcard) {  
        throw new NoRouteNotFoundException(TAG + "No postcard!");  
    }  
    // 通过 Warehouse.routes.get 尝试获取 RouteMeta  
    RouteMeta routeMeta = Warehouse.routes.get(postcard.getPath());  
    if (null == routeMeta) {  
        // 若 routeMeta 为 null, 可能是并不存在, 或是还没有加载进来  
        // 尝试获取 postcard 的 RouteGroup  
        Class<? extends IRouteGroup> groupMeta = Warehouse.groupsIndex.get(post  
card.getGroup()); // Load route meta.  
        if (null == groupMeta) {  
            throw new NoRouteNotFoundException(TAG + "There is no route match the  
path [" + postcard.getPath() + "], in group [" + postcard.getGroup() + "]");  
        } else {  
            // ...  
            // 如果找到了对应的 RouteGroup, 则将其加载进来并重新调用 completion 进行  
            // 补全  
            IRouteGroup iGroupInstance = groupMeta.getConstructor().newInstance  
();  
            iGroupInstance.loadInto(Warehouse.routes);  
            Warehouse.groupsIndex.remove(postcard.getGroup());  
            // ...  
            completion(postcard); // Reload  
        }  
    } else {  
        // 如果找到了对应的 routeMeta, 将它的信息设置进 postcard 中  
    }  
}
```

```
postcard.setDestination(routeMeta.getDestination());
postcard.setType(routeMeta.getType());
postcard.setPriority(routeMeta.getPriority());
postcard.setExtra(routeMeta.getExtra());
Uri rawUri = postcard.getUri();
    // 将 uri 中的参数设置进 bundle 中
if (null != rawUri) {
    Map<String, String> resultMap = TextUtils.splitQueryParameters(rawUri);
    Map<String, Integer> paramsType = routeMeta.getParamsType();
    if (MapUtils.isNotEmpty(paramsType)) {
        // Set value by its type, just for params which annotation by @Param
        for (Map.Entry<String, Integer> params : paramsType.entrySet()) {
            setValue(postcard,
                    params.getValue(),
                    params.getKey(),
                    resultMap.get(params.getKey()));
        }
        // Save params name which need auto inject.
        postcard.getExtras().putStringArray(ARouter.AUTO_INJECT, params
Type.keySet().toArray(new String[]{}));
    }
    // Save raw uri
    postcard.withString(ARouter.RAW_URI, rawUri.toString());
}
// 对于 provider 和 fragment, 进行特殊处理
switch (routeMeta.getType()) {
    case PROVIDER:
        // 如果是一个 provider, 尝试从 Warehouse 中查找它的类并构造对象,
        然后将其设置到 provider
        Class<? extends IProvider> providerMeta = (Class<? extends IProvider>) routeMeta.getDestination();
        IProvider instance = Warehouse.providers.get(providerMeta);
        if (null == instance) { // There's no instance of this provider
            IProvider provider;
            try {
                provider = providerMeta.getConstructor().newInstance();
                provider.init(mContext);
                Warehouse.providers.put(providerMeta, provider);
                instance = provider;
            } catch (Exception e) {

```

```
        throw new HandlerException("Init provider failed! " + e);
getMessage());
    }
}

postcard.setProvider(instance);
// provider 和 fragment 都会跳过拦截器
postcard.greenChannel();
break;

case FRAGMENT:
    // provider 和 fragment 都会跳过拦截器
    postcard.greenChannel();

default:
    break;
}

}
}
```

这个方法主要完成了对 `postcard` 的信息与 `Warehouse` 的信息进行结合，以补充 `postcard` 的信息，它的步骤如下：

1. 通过 `Warehouse.routes.get` 根据 `path` 尝试获取 `RouteMeta` 对象。
 2. 若获取不到 `RouteMeta` 对象，可能是不存在或是还没有进行加载（第一次都未加载），尝试获取 `RouteGroup` 调用其 `loadInto` 方法将 `RouteMeta` 加载进 `Warehouse`，最后调用 `completion` 重新尝试补全。
 3. 将 `RouteMeta` 的信息设置到 `postcard` 中，其中会将 `rawUri` 的参数设置进 `Bundle`。
 4. 对于 `Provider` 和 `Fragment` 特殊处理，其中 `Provider` 会从 `Warehouse` 中加载并构造它的对象，然后设置到 `postcard`。`Provider` 和 `Fragment` 都会跳过拦截器。

RouteGroup 的 loadInto 仍然是自动生成的，例如下面就是一些自动生成的代码：

```
public void loadInto(Map<String, RouteMeta> atlas) {  
    atlas.put("/homework/commit", RouteMeta.build(RouteType.ACTIVITY, HomeworkCo  
mmitActivity.class, "/homework/commit", "homework", null, -1, -2147483648));
```

```
// ...
}
```

它包括了我们补全所需要的如 `Destination`、`Class`、`path` 等信息，在生成代码时自动根据注解进行生成。

执行跳转

我们看看 `navigation` 方法是如何实现的跳转：

```
private Object _navigation(final Context context, final Postcard postcard, final int requestCode, final NavigationCallback callback) {
    final Context currentContext = null == context ? mContext : context;
    switch (postcard.getType()) {
        case ACTIVITY:
            // 对 Activity, 构造 Intent, 将参数设置进去
            final Intent intent = new Intent(currentContext, postcard.getDestination());
            intent.putExtras(postcard.getExtras());
            // Set flags.
            int flags = postcard.getFlags();
            if (-1 != flags) {
                intent.setFlags(flags);
            } else if (!(currentContext instanceof Activity)) { // Non activity, need less one flag.
                intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            }
            // 切换到主线程, 根据是否需要 result 调用不同的 startActivity 方法
            new Handler(Looper.getMainLooper()).post(new Runnable() {
                @Override
                public void run() {
                    if (requestCode > 0) { // Need start for result
                        ActivityCompat.startActivityForResult((Activity) currentContext, intent, requestCode, postcard.getOptionsBundle());
                    } else {
                        ActivityCompat.startActivity(currentContext, intent, postcard.getOptionsBundle());
                    }
                    if ((0 != postcard.getEnterAnim() || 0 != postcard.getExitAnim()) && currentContext instanceof Activity) { // Old version.
                }
            });
        }
    }
}
```

```

        ((Activity) currentContext).overridePendingTransition(po
stcard.getEnterAnim(), postcard.getExitAnim());
    }
    if (null != callback) { // Navigation over.
        callback.onArrival(postcard);
    }
}
});

break;
case PROVIDER:
    // provider 直接返回对应的 provider
    return postcard.getProvider();
case BOARDCAST:
case CONTENT_PROVIDER:
case FRAGMENT:
    // 对于 broadcast、contentprovider、fragment，构造对象，设置参数后
    返回
    Class fragmentMeta = postcard.getDestination();
    try {
        Object instance = fragmentMeta.getConstructor().newInstance();
        if (instance instanceof Fragment) {
            ((Fragment) instance).setArguments(postcard.getExtras());
        } else if (instance instanceof android.support.v4.app.Fragment)
{
            ((android.support.v4.app.Fragment) instance).setArguments(p
ostcard.getExtras());
        }
        return instance;
    } catch (Exception ex) {
        logger.error(Consts.TAG, "Fetch fragment instance error, " + Te
xtUtils.formatStackTrace(ex.getStackTrace()));
    }
case METHOD:
case SERVICE:
default:
    return null;
}
return null;
}

```

可以发现，它会根据 postcard 的 type 来分别处理：

- 对于 Activity，会构造一个 Intent 并将之前 postcard 中的参数设置进去，之后会根据是否需要 result 调用不同的 startActivity 方法。
- 对于 Provider，直接返回其对应的 provider 对象。
- 对于 Broadcast、ContentProvider、Fragment，反射构造对象后，将参数设置进去并返回。

可以发现 ARouter 的初始化和路由跳转的整体逻辑还是不难的，实际上就是对 Activity、Fragment 的调转过程进行了包装。

Service 的获取

ARouter 除了可以通过 ARouter.getInstance().build().navigation() 这样的方式实现页面跳转之外，还可以通过 ARouter.getInstance().navigation(XXService.class) 这样的方式实现跨越组件的服务获取，我们看看它是如何实现的：

```
public <T> T navigation(Class<? extends T> service) {
    return _ARouter.getInstance().navigation(service);
}
```

仍然跳转到了 _ARouter 中去实现：

```
protected <T> T navigation(Class<? extends T> service) {
    try {
        Postcard postcard = LogisticsCenter.buildProvider(service.getName());
        // Compatible 1.0.5 compiler sdk.
        // Earlier versions did not use the fully qualified name to get the service
        if (null == postcard) {
            // No service, or this service in old version.
            postcard = LogisticsCenter.buildProvider(service.getSimpleName());
        }
        if (null == postcard) {
```

```
        return null;
    }
    LogisticsCenter.completion(postcard);
    return (T) postcard.getProvider();
} catch (NoRouteNotFoundException ex) {
    logger.warning(Consts.TAG, ex.getMessage());
    return null;
}
}
```

这里首先通过 `LogisticsCenter.buildProvider` 传入 `service.class` 的 name 构建出了一个 `postcard`。

而在 `ARouter` 老版本中，并不是通过这样一个完整的 name 来获取 `Service` 的，而是通过 `simpleName`，下面为了兼容老版本，在获取不到时会尝试用老版本的方式重新构建一次。

之后会通过 `LogisticsCenter.completion` 对 `postcard` 进行补全，最后通过 `postcard.Provider` 获取对应的 `Provider`。

除了 `buildProvider` 之外，其他方法我们已经在前面进行过分析，就不再赘述了：

```
public static Postcard buildProvider(String serviceName) {
    RouteMeta meta = Warehouse.providersIndex.get(serviceName);
    if (null == meta) {
        return null;
    } else {
        return new Postcard(meta.getPath(), meta.getGroup());
    }
}
```

这里实际上非常简单，就是通过 `Warehouse` 中已经初始化的 `providersIndex` 根据 `serviceName` 获取对应的 `RouteMeta`，之后根据 `RouteMeta` 的 `path` 和 `group` 返回对应的 `Postcard`。

拦截器机制

通过前面的分析，可以发现 ARouter 中存在一套拦截器机制，在 `completion` 的过程中对拦截器进行了执行，让我们看看它的拦截器机制的实现。

我们先看到 `IInterceptor` 接口：

```
public interface IInterceptor extends IProvider {
    /**
     * The operation of this interceptor.
     *
     * @param postcard meta
     * @param callback cb
     */
    void process(Postcard postcard, InterceptorCallback callback);
}
```

拦截器中主要通过 `process` 方法完成执行过程，可以在其中对 `postcard` 进行处理。而拦截器的执行我们知道，是通过 `InterceptorServiceImpl.doInterceptions` 实现的：

```
if (null != Warehouse.interceptors && Warehouse.interceptors.size() > 0) {
    checkInterceptorsInitStatus();
    if (!interceptorHasInit) {
        callback.onInterrupt(new HandlerException("Interceptors initialization
takes too much time."));
        return;
    }
    LogisticsCenter.executor.execute(new Runnable() {
        @Override
        public void run() {
            CancelableCountDownLatch interceptorCounter = new CancelableCountDo
wnLatch(Warehouse.interceptors.size());
            try {
                _execute(0, interceptorCounter, postcard);
                interceptorCounter.await(postcard.getTimeout(), TimeUnit.SECOND
S);
                if (interceptorCounter.getCount() > 0) { // Cancel the naviga
tion this time, if it hasn't return anythings.
            }
        }
    });
}
```

```

        callback.onInterrupt(new HandlerException("The interceptor
processing timed out."));
    } else if (null != postcard.getTag()) { // Maybe some excepti
on in the tag.
        callback.onInterrupt(new HandlerException(postcard.getTag()).
toString()));
    } else {
        callback.onContinue(postcard);
    }
} catch (Exception e) {
    callback.onInterrupt(e);
}
}
});
} else {
    callback.onContinue(postcard);
}
}

```

这里的执行通过一个 Executor 执行，它首先构造了一个值为 interceptors 个数的 CountDownLatch，之后通过 _execute 方法进行执行：

```

private static void _execute(final int index, final CancelableCountDownLatch co
unter, final Postcard postcard) {
    if (index < Warehouse.interceptors.size()) {
        IInterceptor iInterceptor = Warehouse.interceptors.get(index);
        iInterceptor.process(postcard, new InterceptorCallback() {
            @Override
            public void onContinue(Postcard postcard) {
                // Last interceptor execute over with no exception.
                counter.countDown();
                _execute(index + 1, counter, postcard); // When counter is down,
it will be execute continue ,but index bigger than interceptors size, then U k
now.
            }
            @Override
            public void onInterrupt(Throwable exception) {
                // Last interceptor execute over with fatal exception.
                postcard.setTag(null == exception ? new HandlerException("No me
ssage.") : exception.getMessage()); // save the exception message for backu
p.
                counter.cancel();
                // Be attention, maybe the thread in callback has been changed,
                // then the catch block(L207) will be invalid.
            }
        });
    }
}

```

```
// The worst is the thread changed to main thread, then the app  
will be crash, if you throw this exception!  
        if (!Looper.getMainLooper().equals(Looper.myLooper())) {    //  
            You shouldn't throw the exception if the thread is main thread.  
            throw new HandlerException(exception.getMessage());  
        }  
    }  
});  
}  
}
```

这里会调用 `interceptor.process`, 并在其调用完成后调用 `_execute` 执行下一个 `interceptor`, 从而对每个 `interceptor` 进行执行。

注解处理

那么 `ARouter` 是如何自动生成 `RouteRoot`、`RouteMeta`、`ProviderGroup`、`Provider`、`Interceptor` 的子类的呢?

实际上 `ARouter` 是通过 `AnnotationProcessor` 配合 `AutoService` 实现的, 而对于类的生成主要是通过 `JavaPoet` 实现了对 `Java` 文件的编写, 关于 `JavaPoet` 的具体使用可以看到其 `GitHub` 主页: <https://github.com/square/javapoet>

由于注解处理部分的代码大部分就是获取注解的属性, 并结合 `JavaPoet` 生成每个 `Element` 对应的 `Java` 代码, 这块的代码比较多且并不复杂, 这里就不带大家去看这部分的源码了, 有兴趣的读者可以看看 `arouter-complier` 包下的具体实现。

总结

ARouter 的核心流程主要分为三部分：

编译期注解处理

通过 `AnnotationProcessor` 配合 `JavaPoet` 实现了编译期根据注解对 `RouteRoot`、`RouteMeta`、`ProviderGroup`、`Provider`、`Interceptor` 等类的代码进行生成，在这些类中完成了对 `Warehouse` 中装载注解相关信息的工作。

初始化

通过 `ARouter.init`，可以对 `ARouter` 进行初始化，它主要分为两个步骤：

1. 遍历 `Apk` 的 `dex` 文件，查找存放自动生成类的包下的类的 `ClassName` 集合。其中为了加快查找速度，通过一个线程池进行了异步查找，并通过 `CountDownLatch` 来等待所有异步查找任务的结束。这个查找过程在非 `debug` 模式下是有缓存的，因为 `release` 的 `Apk` 其自动生成的类的信息必然不会变化
2. 根据 `ClassName` 的类型，分别构建 `RouteRoot`、`InterceptorGroup`、`ProviderGroup` 的对象并调用了其 `loadInto` 方法将这些 `Group` 的信息装载进 `Warehouse`，这个过程并不会将具体的 `RouteMeta` 装载。这些 `Group` 中主要包含了一些其对应的下级的信息（如 `RouteGroup` 的 `Class` 对象等），之后就只需要取出下级的信息并从中装载，不再需要遍历 `dex` 文件。

路由

路由的过程，主要分为以下几步：

1. 通过 ARouter 中的 `build(path)` 方法构建出一个 Postcard，或直接通过其 `navigate(serviceClass)` 方法构建一个 Postcard。
2. 通过对 Postcard 中提供的一系列方法对这次路由进行配置，包括携带的参数，是否跳过拦截器等等。
3. 通过 `navigation` 方法完成路由的跳转，它的步骤如下：
 - a. 通过 `LogisticsCenter.completion` 方法根据 Postcard 的信息结合 Warehouse 中加载的信息对 Postcard 的 Destination、Type 等信息进行补全，这个过程中会实现对 RouteMeta 信息的装载，并且对于未跳过拦截器的类会逐个调用拦截器进行拦截器处理。
 - b. 根据补全后 Postcard 的具体类型，调用对应的方法进行路由的过程（如对于 Activity 调用 `startActivity`，对于 Fragment 构建对象并调用 `setArgument`）。
4. 将 `navigation` 的结果返回（Activity 返回的就是 null）

3. 深入解析 AsyncTask 源码（一款 Android 内置的异步任务执行库）

AsyncTask 是 Android SDK 中提供的一个用于执行异步任务的框架，在 Android 兴起的早期被广泛使用，但如今已经被 RxJava、协程等新兴框架所取代。虽然它存在着一些不足，但我们还是可以尝试了解一下它的实现原理以及存在的不足。

功能概述

首先，让我们来简单地了解一下它的设计初衷：

AsyncTask 的设计初衷是能够帮助用户方便地完成异步任务的线程调度，它对用户提供了如下的几个接口：

```
public abstract class AsyncTask<Params, Progress, Result> {
    @WorkerThread
    protected abstract Result doInBackground(Params... params);
    @MainThread
    protected void onPostExecute() {
    }
    @MainThread
    protected void onPostExecute(Result result) {
    }
    @MainThread
    protected void onProgressUpdate(Progress... values) {
    }
    @MainThread
    protected void onCancelled(Result result) {
        onCancelled();
    }
}
```

首先看到它的三个范型参数：`Params` 代表了传递给它的参数类型，`Progress` 代表了用于反馈进度的数据类型，而 `Result` 则代表了异步执行的结果类型。

用户可以通过实现 `doInBackground` 方法来编写在异步线程所要进行的处理，并且通过 `onPostExecute` 方法的重写实现对异步请求结果的获取。并且，用户可以通过 `onPreExecute` 实现了在 `doInBackground` 之前进行一些预处理，并且可以通过 `onProgressUpdate` 实现对进度的监听。以及通过 `onCancelled` 实现对任务中断的监听。

当我们编写一个 `AsyncTask` 后，只需要调用它的 `execute` 方法即可，而背后的线程调度的过程都会由它替我们完成，看上去是十分美好的。

同时从上面的代码中可以看到，每个方法都有被类似 `@MainThread` 的注解标注，这些注解主要的作用是用来标注这个方法运行所处的线程。可以看出，只有 `doInBackground` 是在异步线程进行执行。

创建

接着，让我们看看它的创建过程，我们来到它的构造函数：

```
public AsyncTask() {
    this((Looper) null);
}
```

它的无参构造函数转调到了它的有参构造函数，这个构造函数需要以 `Looper` 作为一个参数：

```
public AsyncTask(@Nullable Looper callbackLooper) {
    mHandler = callbackLooper == null || callbackLooper == Looper.getMainLooper()
        ? getMainHandler()
        : new Handler(callbackLooper);
    mWorker = new WorkerRunnable<Params, Result>() {
        public Result call() throws Exception {
            mTaskInvoked.set(true);
            Result result = null;
            try {
                Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                //noinspection unchecked
                result = doInBackground(mParams);
                Binder.flushPendingCommands();
            } catch (Throwable tr) {
                mCancelled.set(true);
                throw tr;
            } finally {
                postResult(result);
            }
        }
    };
}
```

```
        }
        return result;
    }
};

mFuture = new FutureTask<Result>(mWorker) {
    @Override
    protected void done() {
        try {
            postResultIfNotInvoked(get());
        } catch (InterruptedException e) {
            android.util.Log.w(LOG_TAG, e);
        } catch (ExecutionException e) {
            throw new RuntimeException("An error occurred while executing d
oInBackground()", e.getCause());
        } catch (CancellationException e) {
            postResultIfNotInvoked(null);
        }
    }
};
}
```

首先它根据传递进来的 Looper 来构建了一个 Handler，若没有指定 Looper 或指定的 Looper 是主线程的 Looper，则指定内置的 InternalHandler 对消息进行处理，否则构造一个对应的 Handler。可以看出， **AsyncTask 不是一定回到主线程的。**

接着它构建了一个 WorkerRunnable，WorkerRunnable 实际上就是对 Callable 的简单包装，区别仅仅在于可以传入参数。当 mWorker 被执行时，它首先将当前 Task 设置为了被执行，然后进行了线程优先级的设置，并调用了 doInBackground 方法并获取了返回值。看来这个 WorkerRunnable 是在异步线程中执行的。不论成功与否，它最后都会调用 postResult 进行结果的交付，并返回结果。

之后它基于前面的 `mWorker` 构建了一个 `FutureTask`, 当执行完成或被取消时, 会调用 `postResultIfNotInvoked` 方法传入 `mWorker` 的执行结果。

这里有两个类似的方法: `postResult` 和 `postResultIfNotInvoked`, 至于为什么要这样设计我们后面会提到。

总的来说, 其构造过程主要是对 `Handler`、`WorkerRunnable` 以及 `FutureTask` 进行了构建。

执行

接着让我们看看当我们调用 `execute` 之后它做了什么:

```
@MainThread
public final AsyncTask<Params, Progress, Result> execute(Params... params) {
    return executeOnExecutor(sDefaultExecutor, params);
}
```

可以看到, 它转调到了 `executeOnExecutor` 方法, 并且传入了一个 `Executor` 对象 `sDefaultExecutor`, 我们先不去关注这个 `Executor` 的设计, 让我们先看看 `executeOnExecutor` 方法:

```
@MainThread
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor ex
ec
    Params... params) {
    if (mStatus != Status.PENDING) {
        switch (mStatus) {
            case RUNNING:
                throw new IllegalStateException("Cannot execute task:"
                    + " the task is already running.");
            case FINISHED:
                throw new IllegalStateException("Cannot execute task:"
                    + " the task has already been executed"
    }
}
```

```
        + "(a task can be executed only once)");
    }

    mStatus = Status.RUNNING;
    onPreExecute();
    mWorker.mParams = params;
    exec.execute(mFuture);
    return this;
}
```

首先，它对当前 Task 的状态进行了检查，`AsyncTask` 共有三种状态：`PENDING`、`RUNNING`、`FINISHED`，分别代表了待执行、正在执行以及已执行。这里只有 `PENDING` 的 Task 才能被 `execute`。

之后它首先改变了当前 Task 的状态并调用了 `onPreExecute` 方法进行了用户实现的预处理，之后将用户的参数交给 `mWorker` 后，将 `mFuture` 交给了传入的线程池进行处理。

首先，`FutureTask` 执行后，会使得 `mWorker` 被执行，它执行后会将 `mTaskInvoked` 置为 `true` 并调用 `postResult` 进行结果的交付：

```
private Result postResult(Result result) {
    @SuppressWarnings("unchecked")
    Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
        new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}
```

这里实际上就是把结果包装成了 `AsyncTaskResult` 类并放入了消息队列，这样就实现了线程的切换，将消息通过 `Handler` 由子线程发送给了主线程（指 `AsyncTask` 被指定的线程），当 `Handler` 收到消息后，就会对消息进行处理。

我们可以看到在构造时没有指定 Looper 的情况下，默认的 InternalHandler 是如何进行处理的：

```
private static class InternalHandler extends Handler {
    public InternalHandler(Looper looper) {
        super(looper);
    }
    @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                // There is only one result
                result.mTask.finish(result mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                result.mTask.onProgressUpdate(result mData);
                break;
        }
    }
}
```

可以看到，在 MESSAGE_POST_RESULT 消息下，它在收到 AsyncTaskResult 后会调用 finish 方法进行整个任务的完成处理：

```
private void finish(Result result) {
    if (isCancelled()) {
        onCancelled(result);
    } else {
        onPostExecute(result);
    }
    mStatus = Status.FINISHED;
}
```

在 finish 方法中，若任务已经取消，它会调用 onCancelled 方法进行回调，若任务正常完成，则会调用 onPostExecute 方法，并最后设置它的状态为 FINISHED。

而当 FutureTask 执行完成后，会调用 postResultIfNotInvoked 方法，并传入 mWorker 的执行结果：

```
private void postResultIfNotInvoked(Result result) {
    final boolean wasTaskInvoked = mTaskInvoked.get();
    if (!wasTaskInvoked) {
        postResult(result);
    }
}
```

可以看到，这里实际上是在 Task 没有被 Invoke 的情况下才会调用到，由于 mWorker 在被执行时首先就会将 wasTaskInvoked 置为 true，因此实际上这里是很少能被调用到的。

我们再看看 onProgressUpdated 何时会被调用到，我们可以找到 MESSAGE_POST_PROGRESS 的消息是何时被发出的：

```
protected final void publishProgress(Progress... values) {
    if (!isCancelled()) {
        getHandler().obtainMessage(MESSAGE_POST_PROGRESS,
            new AsyncTaskResult<Progress>(this, values)).sendToTarget();
    }
}
```

它是在 publishProgress 方法中被调用的，这个方法是提供给用户的，因此进度的更新需要用户自行在 doInBackground 中调用 publishProgress 从而实现。

取消

接着让我们看看任务的取消，我们看到 cancel 方法：

```
public final boolean cancel(boolean mayInterruptIfRunning) {
    mCancelled.set(true);
    return mFuture.cancel(mayInterruptIfRunning);
```

```
}
```

可以看到，`cancel` 方法最后会调用到 `mFuture` 的 `cancel`，并且它需要传递一个参数 `mayInterruptIfRunning`，当该参数为 `true` 时，即使任务在运行也会被打断，而如果这个参数是 `false`，则 `doInBackground` 仍然会一直执行到结束。

这个设计非常奇怪，按道理来说取消的场景只有任务正在执行时需要取消，完全可以提供一个这个参数默认为 `true` 的方法供外部调用。并且这个方法只能尽量让任务尽快结束，如果执行的过程中有一些不可打断的操作，则这个方法调用后仍然不会使得任务停止。

线程池

对于 `AsyncTask`，我们还有最后一个没有研究，那就是它执行任务的

`sDefaultExecutor`:

```
private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;
```

可以看到它的默认值为 `SERIAL_EXECUTOR`，而 `SERIAL_EXECUTOR` 则

是 `SerialExecutor` 的实例：

```
private static class SerialExecutor implements Executor {
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
    Runnable mActive;
    public synchronized void execute(final Runnable r) {
        mTasks.offer(new Runnable() {
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
            }
        });
        if (mActive == null) {
```

```
        scheduleNext();
    }
}

protected synchronized void scheduleNext() {
    if ((mActive = mTasks.poll()) != null) {
        THREAD_POOL_EXECUTOR.execute(mActive);
    }
}
}
```

可以看到，这个 `Executor` 内部维护了一个 `ArrayDeque` 队列，这个队列是一个任务缓存队列，里面存放了对真正要执行的 `Runnable` 进行包装的 `Runnable`。

执行时，当 `mActive` 为 `null`，它会调用下面的 `scheduleNext` 方法，在该方法中会从 `mTasks` 队首取出任务赋值给 `mActive`，之后通过 `THREAD_POOL_EXECUTOR` 这个线程池对该任务进行执行。当任务执行完成后，会继续调用 `scheduleNext` 对队列中的下一个任务进行执行。通过这样的设计，`Runnable` 的执行变成了一种按序执行，只有前一个执行结束，才会进行下一个任务的执行，因此 `AsyncTask` 的执行顺序实际上是一种串行执行。

实际上在 Android 1.6 之前，`AsyncTask` 都是通过一个单独的 `background` 线程对任务进行串行执行，但在 Android 1.6 之后，设计人员认为串行执行效率太低，因此引入了一个线程池从而支持对它进行并行执行。

而由于这个改动使得很多应用出现了并发问题，因此 Android 3.0 之后又把它改回了串行执行（引入了 `SerialExecutor`），同时对并行执行进行了支持。但它默认仍然是以串行的方式对任务进行执行，如果需要并行执行可以调用 `executeOnExecutor` 方法并将 `THREAD_POOL_EXECUTOR` 传入。

那么 `THREAD_POOL_EXECUTOR` 又是一个怎样的线程池呢？让我们看看它的声明：

```

// We want at least 2 threads and at most 4 threads in the core pool,
// preferring to have 1 less than the CPU count to avoid saturating
// the CPU with background work
private static final int CORE_POOL_SIZE = Math.max(2, Math.min(CPU_COUNT - 1,
4));
private static final int MAXIMUM_POOL_SIZE = CPU_COUNT * 2 + 1;
private static final int KEEP_ALIVE_SECONDS = 30;
private static final BlockingQueue<Runnable> sPoolWorkQueue = new LinkedBlockingQueue<Runnable>(128);

ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
    CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE_SECONDS, TimeUnit.
SECONDS,
    sPoolWorkQueue, sThreadFactory);
threadPoolExecutor.allowCoreThreadTimeOut(true);
THREAD_POOL_EXECUTOR = threadPoolExecutor;

```

可以看到，`THREAD_POOL_EXECUTOR` 的配置如下：

- `corePoolSize`: CPU 核心数 - 1，但保持在 2-4 个。
- `maxPoolSize`: $2 * \text{CPU 核心数} + 1$
- `keepAliveSeconds`: 30 秒
- `workQueue`: 容量为 128 的阻塞队列。

设置核心线程数为 CPU 核心数 - 1 的主要原因是除开主线程外这个线程数已经足够了，CPU 最多也就能同时进行 CPU 核心数个线程的真正并行执行，超出了也只是将任务按时间分片执行，带不来太大的效率提高。

不足之处

`AsyncTask` 看似十分美好，但实际上存在着非常多的不足，这些不足使得它逐渐退出了历史舞台：

- 生命周期: `AsyncTask` 没有与 `Activity`、`Fragment` 的生命周期绑定, 即使 `Activity` 被销毁, 它的 `doInBackground` 任务仍然会继续执行。
- 取消任务: `AsyncTask` 的 `cancel` 方法的参数 `mayInterruptIfRunning` 存在的意义不大, 并且它无法保证任务一定能取消, 只能尽快让任务取消 (比如如果正在进行一些无法打断的操作时, 任务就仍然会运行)
- 内存泄漏: 由于它没有与 `Activity` 等生命周期进行绑定, 因此它的生命周期仍然可能比 `Activity` 长, 如果将它作为 `Activity` 的非 `static` 内部类, 则它会持有 `Activity` 的引用, 导致 `Activity` 的内存无法释放。(与 `Handler` 的内存泄漏问题类似)
- 并行/串行: 由于 `AsyncTask` 的串行和并行执行在多个版本上都进行了修改, 所以当多个 `AsyncTask` 依次执行时, 它究竟是串行还是并行执行取决于用户手机的版本。具体修改如下:
 - Android 1.6 之前: 各个 `AsyncTask` 按串行的顺序进行执行。
 - Android 3.0 之前: 由于设计者认为串行执行效率太低, 因此改为了并行执行, 最多五个 `AsyncTask` 同时执行。
 - Android 3.0 之后: 由于之前的改动, 很多应用出现了并发问题, 因此引入 `SerialExecutor` 改回了串行执行, 但对并行执行进行了支持。

总结

到了这里, 我们基本上能对 `AsyncTask` 的实现原理有一个大致的了解了, 它的原理实际上还是挺简单的: 通过 `Executor` 与 `FutureTask` 配合, 从而实现实任务的异步执行, 最后在任务结束后通过 `Handler` 进行线程的切换从而实现了整个线程调

度的功能。并且在 Android 3.0 之后的版本中，默认情况下 **AsyncTask** 的执行顺序是串行进行的。

4. 深入解析 **Volley** 源码（一款 Google 推出的网络请求框架）

Volley 是 Google 开发的一款网络请求框架，目前已停止更新。虽然目前大家的关注焦点都在 Retrofit、OkHttp 等第三方网络请求框架，团队的项目中所用的也是这两个框架，但 Volley 中还是有非常多优秀的设计思想值得我们去学习的。因此今天准备来学习一下 Volley 的源码，了解一下它的核心设计思想。

Volley

我们先看到 Volley 的入口——Volley 类。

创建 RequestQueue

Volley 在使用之前，我们需要一个请求队列对象 RequestQueue，一般整个应用统一用同一个 RequestQueue，让我们看看创建它的方法 `Volley.newRequestQueue(Context)`：

```
/**  
 * Creates a default instance of the worker pool and calls {@link RequestQueue#  
 start()} on it.  
 *  
 * @param context A {@link Context} to use for creating the cache dir.  
 * @return A started {@link RequestQueue} instance.  
 */  
public static RequestQueue newRequestQueue(Context context) {
```

```
    return newRequestQueue(context, (BaseHttpStack) null);
}
```

它转调到了 `Volley.newRequestQueue(Context, BaseHttpStack)` 方法，同时还有一个 `Volley.newRequestQueue(Context, HttpStack)` 方法：

```
/**  
 * Creates a default instance of the worker pool and calls {@link RequestQueue#  
 start()} on it.  
 *  
 * @param context A {@link Context} to use for creating the cache dir.  
 * @param stack An {@link HttpStack} to use for the network, or null for default.  
 * @return A started {@link RequestQueue} instance.  
 * @deprecated Use {@link #newRequestQueue(Context, BaseHttpStack)} instead to  
 avoid depending  
 * on Apache HTTP. This method may be removed in a future release of Volle  
 y.  
 */  
public static RequestQueue newRequestQueue(Context context, HttpStack stack) {  
    if (stack == null) {  
        return newRequestQueue(context, (BaseHttpStack) null);  
    }  
    return newRequestQueue(context, new BasicNetwork(stack));  
}
```

从上面的注释可以看出，`HttpStack` 是一个用于在 `Network` 中使用的对象，如果传入的 `stack` 不为 `null`，则会调用到 `Volley.newRequestQueue(Context, Network)`，否则它同样会转调到 `Volley.newRequestQueue(Context, BaseHttpStack)`：

```
/**  
 * Creates a default instance of the worker pool and calls {@link RequestQueue#  
 start()} on it.  
 *  
 * @param context A {@link Context} to use for creating the cache dir.  
 * @param stack A {@link BaseHttpStack} to use for the network, or null for def  
 ault.  
 * @return A started {@link RequestQueue} instance.  
 */
```

```
public static RequestQueue newRequestQueue(Context context, BaseHttpStack stack) {
    BasicNetwork network;
    // 创建 Network 对象
    if (stack == null) {
        // stack 为空则创建一个 Stack, 然后再创建 Network
        // 高版本下, HttpStack 使用的是 HurlStack, 低版本下使用 HttpClientStack
        if (Build.VERSION.SDK_INT >= 9) {
            network = new BasicNetwork(new HurlStack());
        } else {
            // Prior to Gingerbread, HttpURLConnection was unreliable.
            // See: http://android-developers.blogspot.com/2011/09/androids-htt
            p-clients.html
            // At some point in the future we'll move our minSdkVersion past Fro
            yo and can
            // delete this fallback (along with all Apache HTTP code).
            String userAgent = "volley/0";
            try {
                String packageName = context.getPackageName();
                PackageInfo info =
                    context.getPackageManager().getPackageInfo(packageName,
                    /* flags= */ 0);
                userAgent = packageName + "/" + info.versionCode;
            } catch (NameNotFoundException e) {
            }
            network =
                new BasicNetwork(
                    new HttpClientStack(AndroidHttpClient.newInstance(us
erAgent)));
        }
    } else {
        network = new BasicNetwork(stack);
    }
    return newRequestQueue(context, network);
}
```

这个方法主要做的事情就是根据 `stack` 创建 `network` 对象。

在 `stack` 为空时, 对于高于 9 的 SDK 版本, 使用 `HurlStack`, 而对于低于它的版本则使用 `HttpClientStack`。根据上面的注释可以看出这样做的原因: 因为在

SDK 9 之前的 `HttpURLConnection` 不是很可靠。(我们可以推测在高版本 (SDK > 9) Volley 基于 `HttpURLConnection` 实现, 低版本则基于 `HttpClient` 实现。

另外这里可能会比较疑惑, `Network` 和 `HttpStack` 都是用来做什么的? 这些问题我们后面都会一一解决。

它最后同样调用到了 `Volley.newRequestQueue(Context, Network)`:

```
private static RequestQueue newRequestQueue(Context context, Network network)
{
    final Context applicationContext = context.getApplicationContext();
    // Use a Lazy supplier for the cache directory so that newRequestQueue() can be called on
    // main thread without causing strict mode violation.
    DiskBasedCache.FileSupplier cacheSupplier =
        new DiskBasedCache.FileSupplier() {
            private File cacheDir = null;
            @Override
            public File get() {
                if (cacheDir == null) {
                    cacheDir = new File(applicationContext.getCacheDir(), DEFAULT_CACHE_DIR);
                }
                return cacheDir;
            }
        };
    RequestQueue queue = new RequestQueue(new DiskBasedCache(cacheSupplier), network);
    queue.start();
    return queue;
}
```

首先根据 `Context` 获取到了应用缓存文件夹并创建 `Cache` 文件。这里有个比较小的细节, 为了在 `Application` 调用 `newRequestQueue` 同时又不被 `StrictMode` 有关文件操作相关的规则所影响, `Volley` 中使用了一个 `FileSupplier` 来

对 File 进行包装，它采用了一个懒创建的思路，只有用到的时候才创建对应的 cacheDir。

之后构造了 RequestQueue，调用了其 start 方法并对其返回。可以看出来，Volley 是一个 RequestQueue 的静态工厂。

RequestQueue

RequestQueue 中维护了三个容器：两个 PriorityBlockingQueue：

mCacheQueue 与 mNetQueue，以及一个 Set：mCurrentRequests。

- mCacheQueue：用于存放缓存的待请求的 Request。
- mNetQueue：用于存放等待发起的 Request。
- mCurrentQueue：用于存放当前正在进行请求的 Request。

创建

我们先来看看 RequestQueue 的构造函数：

```
public RequestQueue(Cache cache, Network network) {  
    this(cache, network, DEFAULT_NETWORK_THREAD_POOL_SIZE);  
}
```

它转调到了另一个构造函数，并传递了一个默认线程数 DEFAULT_NETWORK_THREAD_POOL_SIZE，它代表了网络请求分派线程启动时的默认个数，默认值为 4。

```
public RequestQueue(Cache cache, Network network, int threadPoolSize) {  
    this(
```

```
        cache,  
        network,  
        threadPoolSize,  
        new ExecutorDelivery(new Handler(Looper.getMainLooper())));  
    }  
}
```

这个构造函数创建了一个主线程的 Handler 并用它构建了一个 ExecutorDelivery，关于 ExecutorDelivery 我们后面再讨论，它是一个用于交付 Response 和 Error 信息的类。

后面转调的构造函数主要是进行一些赋值。

启动

接着我们看看 RequestQueue.start，看看它是如何启动的：

```
/** Starts the dispatchers in this queue. */  
public void start() {  
    stop(); // Make sure any currently running dispatchers are stopped.  
    // Create the cache dispatcher and start it.  
    mCacheDispatcher = new CacheDispatcher(mCacheQueue, mNetworkQueue, mCache,  
    mDelivery);  
    mCacheDispatcher.start();  
    // Create network dispatchers (and corresponding threads) up to the pool si-  
    ze.  
    for (int i = 0; i < mDispatchers.length; i++) {  
        NetworkDispatcher networkDispatcher =  
            new NetworkDispatcher(mNetworkQueue, mNetwork, mCache, mDeliver-  
            y);  
        mDispatchers[i] = networkDispatcher;  
        networkDispatcher.start();  
    }  
}
```

start 的主要用途是启动该 Queue 中的 Dispatcher。它的主要步骤如下：

1. 调用 stop 方法停止所有正在运行的 Dispatcher

2. 创建 CacheDispatcher 并启动。
3. 创建前面指定个数的 NetworkDispatcher (默认为 4 个) 并启动。

可以看出来，每个 RequestQueue 中共有 5 个 Dispatcher，其中有 4 个 NetworkDispatcher 和 1 个 CacheDispatcher。

入队

我们可以通过 RequestQueue.add 将一个 Request 入队，它会根据当前 Request 是否需要进行缓存将其加入 mNetworkQueue 或 mCacheQueue (这里实际上 GET 请求首先会放入 mCacheQueue，其余请求直接放入 mNetworkQueue)

```
/**  
 * Adds a Request to the dispatch queue.  
 *  
 * @param request The request to service  
 * @return The passed-in request  
 */  
public <T> Request<T> add(Request<T> request) {  
    // Tag the request as belonging to this queue and add it to the set of current requests.  
    request.setRequestQueue(this);  
    synchronized (mCurrentRequests) {  
        mCurrentRequests.add(request);  
    }  
    // Process requests in the order they are added.  
    request.setSequence(getSequenceNumber());  
    request.addMarker("add-to-queue");  
    sendRequestEvent(request, RequestEvent.REQUEST_QUEUED);  
    // If the request is uncachable, skip the cache queue and go straight to the network.  
    if (!request.shouldCache()) {  
        mNetworkQueue.add(request);  
        return request;  
    }  
    mCacheQueue.add(request);
```

```
    return request;
}
```

停止

我们先看看 `stop` 做了什么：

```
public void stop() {
    if (mCacheDispatcher != null) {
        mCacheDispatcher.quit();
    }
    for (final NetworkDispatcher mDispatcher : mDispatchers) {
        if (mDispatcher != null) {
            mDispatcher.quit();
        }
    }
}
```

这里仅仅是对每个 `Dispatcher` 调用了其 `quit` 方法。

结束

`RequestQueue` 还有个 `finish` 方法，对应了 `Request.finish`：

```
/**
 * Called from {@link Request#finish(String)}, indicating that processing of the given request
 * has finished.
 */
@SuppressWarnings("unchecked") // see above note on RequestFinishedListener
<T> void finish(Request<T> request) {
    // Remove from the set of requests currently being processed.
    synchronized (mCurrentRequests) {
        mCurrentRequests.remove(request);
    }
    synchronized (mFinishedListeners) {
        for (RequestFinishedListener<T> listener : mFinishedListeners) {
            listener.onRequestFinished(request);
        }
    }
}
```

```
    }
    sendRequestEvent(request, RequestEvent.REQUEST_FINISHED);
}
```

主要是将结束的 Request 从 `mCurrentRequests` 中移除，并调用外部注册的回调以及发送 `REQUEST_FINISHED` 事件。

ExecutorDelivery

接下来我们看看 `ExecutorDelivery` 究竟是做什么的

```
/** Delivers responses and errors. */
public class ExecutorDelivery implements ResponseDelivery {
    private final Executor mResponsePoster;

    /**
     * Creates a new response delivery interface.
     *
     * @param handler {@link Handler} to post responses on
     */
    public ExecutorDelivery(final Handler handler) {
        // Make an Executor that just wraps the handler.
        mResponsePoster =
            new Executor() {
                @Override
                public void execute(Runnable command) {
                    handler.post(command);
                }
            };
    }
    //...
}
```

根据上面的注释可以看出 `ExecutorDelivery` 的主要作用是交付 Response 和 Error 信息。

它内部持有了一个名为 ResponsePoster 的 Executor，每个调用这个 Poster 的 execute 方法的 Runnable 都会通过 Handler.post 发送到主线程的 MessageQueue 中。

接着我们看到它内部的方法：

```
@Override  
public void postResponse(Request<?> request, Response<?> response) {  
    postResponse(request, response, null);  
}  
  
@Override  
public void postResponse(Request<?> request, Response<?> response, Runnable ru  
nnable) {  
    request.markDelivered();  
    request.addMarker("post-response");  
    mResponsePoster.execute(new ResponseDeliveryRunnable(request, response, ru  
nnable));  
}  
  
@Override  
public void postError(Request<?> request, VolleyError error) {  
    request.addMarker("post-error");  
    Response<?> response = Response.error(error);  
    mResponsePoster.execute(new ResponseDeliveryRunnable(request, response, nu  
ll));  
}
```

可以看出来，它内部的方法主要是将 Response 信息和 Error 信息 post 到 MessageQueue 中，其中 request 和 response 会被包装为一个 ResponseDeliveryRunnable。

ResponseDeliveryRunnable

ResponseDeliveryRunnable 是 ExecutorDelivery 的一个内部类，可以看到它的 run 方法：

```

@Override
public void run() {
    // NOTE: If cancel() is called off the thread that we're currently running
    // in (by default, the main thread), we cannot guarantee that deliverResponse()/de-
    // liverError()
    // won't be called, since it may be canceled after we check isCanceled() but before we
    // deliver the response. Apps concerned about this guarantee must either call cancel()
    // from the same thread or implement their own guarantee about not invoking their
    // listener after cancel() has been called.
    // If this request has canceled, finish it and don't deliver.
    if (mRequest.isCanceled()) {
        mRequest.finish("canceled-at-delivery");
        return;
    }
    // Deliver a normal response or error, depending.
    if (mResponse.isSuccess()) {
        mRequest.deliverResponse(mResponse.result);
    } else {
        mRequest.deliverError(mResponse.error);
    }
    // If this is an intermediate response, add a marker, otherwise we're done
    // and the request can be finished.
    if (mResponse.intermediate) {
        mRequest.addMarker("intermediate-response");
    } else {
        mRequest.finish("done");
    }
    // If we have been provided a post-delivery runnable, run it.
    if (mRunnable != null) {
        mRunnable.run();
    }
}

```

这里会根据 Request、Response 的不同状态调用 Request 中的不同方法以对 Response 和 Error 的结果进行交付：

- 如果 Request 被取消，调用 Request.finish 方法直接结束。

- 如果请求成功，调用 `Request.deliverResponse` 方法反馈 `Response`
- 如果请求失败，调用 `Request.deliverError` 方法反馈 `Error`
- 如果该请求只是一个中间请求，则不结束该请求，而是给它加上一个 "intermediate-response" 标记，否则结束该请求。
- 如果 `postResponse` 方法传递了一个 `Runnable` 进来，则执行该 `Runnable`。

NetworkDispatcher

我们接着来看到 `NetworkDispatcher`，它继承自 `Thread`，我们先看到其 `run` 方法：

```
@Override
public void run() {
    Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
    while (true) {
        try {
            processRequest();
        } catch (InterruptedException e) {
            // We may have been interrupted because it was time to quit.
            if (mQuit) {
                Thread.currentThread().interrupt();
                return;
            }
            VolleyLog.e(
                "Ignoring spurious interrupt of NetworkDispatcher thread; "
                + "use quit() to terminate it");
        }
    }
}
```

它在不断循环调用 `processRequest` 方法：

```
// Extracted to its own method to ensure Locals have a constrained Liveness scope by the GC.
// This is needed to avoid keeping previous request references alive for an indeterminate amount
// of time. Update consumer-proguard-rules.pro when modifying this. See also
// https://github.com/google/volley/issues/114
```

```
private void processRequest() throws InterruptedException {
    // Take a request from the queue.
    Request<?> request = mQueue.take();
    processRequest(request);
}
```

这里首先从 `mQuque` (`RequestQueue` 中的 `NetQuque`) 中取出了 `Request`, 虽然 `NetworkDispatcher` 是多个同时执行, 但由于使用了 `BlockingQueue` 因此不用考虑线程安全问题。

可以发现这是一种典型的**生产者消费者模型**, 多个 `NetworkDispatcher` 不断地从 `NetQueue` 中取出 `Request` 并进行网络请求, 用户就是网络请求的生产者, 而 `NetworkDispatcher` 就是网络请求的消费者。

`processRequest` 方法继续调用了 `processRequest(Request)` 方法:

```
void processRequest(Request<?> request) {
    long startTimeMs = SystemClock.elapsedRealtime();
    // 发送 REQUEST_NETWORK_DISPATCH_STARTED 事件
    request.sendEvent(RequestQueue.RequestEvent.REQUEST_NETWORK_DISPATCH_START
ED);
    try {
        request.addMarker("network-queue-take");
        // 如果这个 request 已经被取消, 则 finish 它并通知 Listener Response 没有用
        if (request.isCanceled()) {
            request.finish("network-discard-cancelled");
            request.notifyListenerResponseNotUsable();
            return;
        }
        addTrafficStatsTag(request);
        // 调用 mNetwork.performRequest 发起同步请求拿到 Response
        NetworkResponse networkResponse = mNetwork.performRequest(request);
        request.addMarker("network-http-complete");
        // 如果返回了 304 并且我们已经交付了 Reponse, 则 finish 它并通知 Listener R
esponse 没有用
        if (networkResponse.notModified && request.hasHadResponseDelivered())
        {
            request.finish("not-modified");
        }
    }
```

```

        request.notifyListenerResponseNotUsable();
        return;
    }
    // 对 response 进行转换
    Response<?> response = request.parseNetworkResponse(networkResponse);
    request.addMarker("network-parse-complete");
    // 如果适用，写入缓存
    if (request.shouldCache() && response.cacheEntry != null) {
        mCache.put(request.getCacheKey(), response.cacheEntry);
        request.addMarker("network-cache-written");
    }
    // 通过 ExecutorDelivery 对 Response 进行交付并将 request 标记为已交付
    request.markDelivered();
    mDelivery.postResponse(request, response);
    // 通知 Listener 已获得 Response
    request.notifyListenerResponseReceived(response);
} catch (VolleyError volleyError) {
    volleyError.setNetworkTimeMs(SystemClock.elapsedRealtime() - startTime
Ms);
    // 交付 Error 信息并通知 Listener Response 没有用
    parseAndDeliverNetworkError(request, volleyError);
    request.notifyListenerResponseNotUsable();
} catch (Exception e) {
    VolleyLog.e(e, "Unhandled exception %s", e.toString());
    VolleyError volleyError = new VolleyError(e);
    volleyError.setNetworkTimeMs(SystemClock.elapsedRealtime() - startTime
Ms);
    // 交付 Error 信息并通知 Listener Response 没有用
    mDelivery.postError(request, volleyError);
    request.notifyListenerResponseNotUsable();
} finally {
    // 发送 REQUEST_NETWORK_DISPATCH_FINISHED 事件
    request.sendEvent(RequestQueue.RequestEvent.REQUEST_NETWORK_DISPATCH_F
INISHED);
}
}

```

上面的代码比较长，我们可以将其分为如下的步骤：

1. 发送 REQUEST_NETWORK_DISPATCH_STARTED 事件，表示请求已开始
2. 如果 request 已经被取消，则 finish 它并通知 Listener Response 没有用，不再继续进行

3. 调用 `mNetwork.performRequest` 发起同步请求拿到 `Response`
4. 如果返回了 `304` 并且我们已经交付了 `Reponse`, 则 `finish` 它并通知 `Listener Response` 没有用, 不再继续进行
5. 对 `Response` 进行转换
6. 如果该 `Request` 可以进行缓存, 以 `Request` 为 `key`, `Response` 为 `value` 进行缓存
7. 通过 `ExecutorDelivery` 对 `Response` 进行交付并将 `Request` 标记为已交付
8. 通知 `Listener` 已获得 `Response`
9. 如果出现了异常, 交付 `Error` 信息并通知 `Listener Response` 没有用, 其中所有 `Error` 都需要转换成 `VolleyError` 类。
10. 发送 `REQUEST_NETWORK_DISPATCH_FINISHED` 事件, 表示请求已结束

从上面的步骤我们可以得到一些信息:

1. 有一套 `Event` 机制用于传送各种事件
2. `Network` 类是网络请求真正的实现类。
3. `Response` 有个转换的问题
4. 存在一套 `Response` 缓存机制。

CacheDispatcher

`CacheDispatcher` 同样继承自 `Thread`, 我们先看到其 `run` 方法:

```
@Override  
public void run() {  
    if (DEBUG) VolleyLog.v("start new dispatcher");
```

```

Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
// Make a blocking call to initialize the cache.
mCache.initialize();
while (true) {
    try {
        processRequest();
    } catch (InterruptedException e) {
        // We may have been interrupted because it was time to quit.
        if (mQuit) {
            Thread.currentThread().interrupt();
            return;
        }
        VolleyLog.e(
            "Ignoring spurious interrupt of CacheDispatcher thread; "
            + "use quit() to terminate it");
    }
}
}

```

它也是在不断地调用 `processRequest` 方法：

```

private void processRequest() throws InterruptedException {
    // Get a request from the cache triage queue, blocking until
    // at least one is available.
    final Request<?> request = mCacheQueue.take();
    processRequest(request);

}

```

这里与 `NetworkDispatcher` 中一样，从 `mCacheQueue` 中取出待请求的 `Request`，

之后看到 `processRequest(Request)`：

```

void processRequest(final Request<?> request) throws InterruptedException {
    request.addMarker("cache-queue-take");
    // 发送 Event
    request.sendEvent(RequestQueue.RequestEvent.REQUEST_CACHE_LOOKUP_STARTED);
    try {
        // 若 request 已取消，直接 finish 它
        if (request.isCanceled()) {
            request.finish("cache-discard-canceled");
            return;
        }
        // 通过 request 尝试获取 Entry
    }
}

```

```
Cache.Entry entry = mCache.get(request.getCacheKey());
if (entry == null) {
    // 若缓存未命中，则将其放入 mNetworkQueue 等待进行网络请求
    request.addMarker("cache-miss");
    // Cache miss; send off to the network dispatcher.
    if (!mWaitingRequestManager.maybeAddToWaitingRequests(request)) {
        mNetworkQueue.put(request);
    }
    return;
}
// 如果缓存过期，放入 mNetworkQueue 等待进行网络请求
if (entry.isExpired()) {
    request.addMarker("cache-hit-expired");
    request.setCacheEntry(entry);
    if (!mWaitingRequestManager.maybeAddToWaitingRequests(request)) {
        mNetworkQueue.put(request);
    }
    return;
}
// 缓存命中，转换并获取 Response
request.addMarker("cache-hit");
Response<?> response =
    request.parseNetworkResponse(
        new NetworkResponse(entry.data, entry.responseHeaders));
request.addMarker("cache-hit-parsed");
if (!entry.refreshNeeded()) {
    // 如果得到的 response 不需要刷新，直接交付 Response
    mDelivery.postResponse(request, response);
} else {
    // 如果缓存到期，则将当前 response 设为中间 Response，进行 Response 交付
    // 当 Response 成功交付，则会将其放入 mNetworkQueue 等待网络请求
    request.addMarker("cache-hit-refresh-needed");
    request.setCacheEntry(entry);
    // Mark the response as intermediate.
    response.intermediate = true;
    if (!mWaitingRequestManager.maybeAddToWaitingRequests(request)) {
        // Post the intermediate response back to the user and have
        // the delivery then forward the request along to the network.
        mDelivery.postResponse(
            request,
            response,
            new Runnable() {
                @Override
                public void run() {
```

```

        try {
            mNetworkQueue.put(request);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
});

} else {
    // request has been added to List of waiting requests
    // to receive the network response from the first request once it returns.
    mDelivery.postResponse(request, response);
}
}

} finally {
    request.sendEvent(RequestQueue.RequestEvent.REQUEST_CACHE_LOOKUP_FINISHED);
}
}
}

```

代码比较长，它主要有以下的步骤：

1. 若 Request 已取消，直接 finish 它。
2. 通过 Request 尝试获取缓存的 Response 对应的 Entry。
3. 若缓存未命中，则将其放入 mNetworkQueue 等待进行网络请求。
4. 若缓存过期，则将其放入 mNetworkQueue 等待进行网络请求。
5. 若缓存命中，转换并获取 Response。
6. 若得到的 Response 不需要刷新，直接交付 Response。
7. 若检测 Response 发现缓存到期，则将当前 Response 设为中间 Response，进行 Response 交付。此时传入了一个 Runnable，当 Response 成功交付时，会将 Request 放入 mNetworkQueue 等待网络请求。

简单点说就是，如果缓存命中且 Response 未过期，则直接将其交付，否则将其放入 mNetworkQueue 等待网络请求。

Request

前面提到的很多机制都与 `Request` 有关，为了更好的理解，我们暂时先不关心 `Network` 是如何发起网络请求的，让我们先看看 `Request` 的组成。

`Request` 只是一个抽象类，它的几个实现如下：

- `StringRequest`: 返回 `String` 的 `Request`。
- `JsonRequest`: `JsonObjectRequest` 和 `JsonArrayRequest` 的基类。
- `JsonObjectRequest`: 返回 `JsonObject` 的 `Request`。
- `JsonArrayRequest`: 返回 `JsonArray` 的 `Request`。
- `ImageRequest`: 返回图片的 `Request`。
- `ClearCacheRequest`: 一个用来清空缓存的 `Request`，并不是用来进行网络请求的 `Request`。

构建

要构建一个 `Request`，我们可以通过其构造函数，其中不同的子类有不同的构造函数，主要的不同是体现在其传入的 `Listener` 的不同，例如 `StringRequest` 的构造函数如下：

```
public StringRequest(int method, String url, Listener<String> listener, @Nulla  
ble ErrorListener errorListener) {  
    super(method, url, errorListener);  
    mListener = listener;  
}
```

它们都会调用 `Request(method, url, errorListener)` 的构造函数：

```
public Request(int method, String url, @Nullable Response.ErrorListener listener) {
    mMethod = method;
    mUrl = url;
    mErrorListener = listener;
    setRetryPolicy(new DefaultRetryPolicy());
    mDefaultTrafficStatsTag = findDefaultTrafficStatsTag(url);
}
```

如果想要创建一个 POST 请求并传入参数，需要重写它的 `getParams` 方法：

```
protected Map<String, String> getParams() throws AuthFailureError {
    return null;
}
```

取消

取消一个 `Request` 十分简单，只需要将 `mCanceled` 置为 `true` 即可：

```
public void cancel() {
    synchronized (mLock) {
        mCanceled = true;
        mErrorListener = null;
    }
}
```

Response 转换

`Request` 的 `parseNetworkResponse` 方法是一个抽象方法，需要不同子类去具体实现。

例如 `StringRequest` 的实现如下：

```
@Override
@SuppressLint("DefaultCharset")
protected Response<String> parseNetworkResponse(NetworkResponse response) {
    String parsed;
    try {
```

```
        parsed = new String(response.data, HttpHeaderParser.parseCharset(respo
nse.headers));
    } catch (UnsupportedEncodingException e) {
        // Since minSdkVersion = 8, we can't call
        // new String(response.data, Charset.defaultCharset())
        // So suppress the warning instead.
        parsed = new String(response.data);
    }
    return Response.success(parsed, HttpHeaderParser.parseCacheHeaders(respons
e));
}
```

非常简单，根据 `Response.data` 转化为了对应的 `String`。

Response 的交付

`Response` 的交付方法 `deliverResponse` 是一个抽象方法，需要不同子类去具体实现，它主要做的事就是调用 `Listener` 的 `onResponse` 方法，并传入转换后的对象。

例如 `StringRequest` 的实现如下：

```
@Override
protected void deliverResponse(String response) {
    Response.Listener<String> listener;
    synchronized (mLock) {
        listener = mListener;
    }
    if (listener != null) {
        listener.onResponse(response);
    }
}
```

Error 的交付

`Error` 的交付在 `Request` 中进行了实现，也非常简单，实际上就是调用 `ErrorListener` 的 `onErrorResponse` 方法：

```
public void deliverError(VolleyError error) {
    Response.ErrorListener listener;
    synchronized (mLock) {
        listener = mErrorListener;
    }
    if (listener != null) {
        listener.onErrorResponse(error);
    }
}
```

结束

```
/**
 * Notifies the request queue that this request has finished (successfully or with error).
 *
 * <p>Also dumps all events from this request's event log; for debugging.
 */
void finish(final String tag) {
    if (mRequestQueue != null) {
        mRequestQueue.finish(this);
    }
    if (MarkerLog.ENABLED) {
        final long threadId = Thread.currentThread().getId();
        if (Looper.myLooper() != Looper.getMainLooper()) {
            // If we finish marking off of the main thread, we need to
            // actually do it on the main thread to ensure correct ordering.
            Handler mainThread = new Handler(Looper.getMainLooper());
            mainThread.post(
                new Runnable() {
                    @Override
                    public void run() {
                        mEventLog.add(tag, threadId);
                        mEventLog.finish(Request.this.toString());
                    }
                });
        }
        return;
    }
    mEventLog.add(tag, threadId);
    mEventLog.finish(this.toString());
}
```

主要是调用了 `RequestQueue.finish` 方法，它会将当前 `Request` 从正在执行的 `Request` 队列移除。

缓存

在 `Response` 被缓存之前会调用 `Request.shouldCache` 判断是否需要缓存，

```
/** Whether or not responses to this request should be cached. */
// TODO(#190): Turn this off by default for anything other than GET requests.
private boolean mShouldCache = true;
```

可以看到，除了 `GET` 请求，其余请求方式都不允许缓存。

我们再看看存在缓存中的 `key` 是如何通过 `Request` 转换而得：

```
public String getCacheKey() {
    String url = getUrl();
    // If this is a GET request, just use the URL as the key.
    // For callers using DEPRECATED_GET_OR_POST, we assume the method is GET, which matches
    // legacy behavior where all methods had the same cache key. We can't determine which method
    // will be used because doing so requires calling getPostBody() which is expensive and may
    // throw AuthFailureError.
    // TODO(#190): Remove support for non-GET methods.
    int method = getMethod();
    if (method == Method.GET || method == Method.DEPRECATED_GET_OR_POST) {
        return url;
    }
    return Integer.toString(method) + '-' + url;
}
```

由于新版 `Volley` 中只支持 `GET` 请求进行缓存，因此 `Request` 是以 `url` 作为缓存的 `key`。

Event 机制

我们暂时先不关心 Network 的具体实现，让我们先看看 Event 发出后做了什么。

调用 sendEvent 后，转调到了 RequestQueue.sendRequestEvent 中：

```
void sendRequestEvent(Request<?> request, @RequestEvent int event) {
    synchronized (mEventListeners) {
        for (RequestEventListener listener : mEventListeners) {
            listener.onRequestEvent(request, event);
        }
    }
}
```

可以看出，我们的用户可以向 RequestQueue 中注册一

个 RequestEventListener 来监听 Request 相关的 Event。

Response

相比 Request，Response 就简单了很多（感觉很多 Response 的功能放在了 Request），里面主要是携带了一些请求结束后的相关信息。

```
public class Response<T> {
    /** Callback interface for delivering parsed responses. */
    public interface Listener<T> {
        /** Called when a response is received. */
        void onResponse(T response);
    }
    /** Callback interface for delivering error responses. */
    public interface ErrorListener {
        /**
         * Callback method that an error has been occurred with the provided error code and optional
         * user-readable message.
         */
        void onErrorResponse(VolleyError error);
    }
}
```

```

    }

    /** Returns a successful response containing the parsed result. */
    public static <T> Response<T> success(T result, Cache.Entry cacheEntry) {
        return new Response<>(result, cacheEntry);
    }

    /**
     * Returns a failed response containing the given error code and an optional
     * localized message
     * displayed to the user.
     */

    public static <T> Response<T> error(VolleyError error) {
        return new Response<>(error);
    }

    /** Parsed response, or null in the case of error. */
    public final T result;
    /** Cache metadata for this response, or null in the case of error. */
    public final Cache.Entry cacheEntry;
    /** Detailed error information if <code>errorCode != OK</code>. */
    public final VolleyError error;
    /** True if this response was a soft-expired one and a second one MAY be coming. */
    public boolean intermediate = false;
    /** Returns whether this response is considered successful. */
    public boolean isSuccess() {
        return error == null;
    }

    private Response(T result, Cache.Entry cacheEntry) {
        this.result = result;
        this.cacheEntry = cacheEntry;
        this.error = null;
    }

    private Response(VolleyError error) {
        this.result = null;
        this.cacheEntry = null;
        this.error = error;
    }
}

```

Network

从前面的分析中，我们可以知道，真正的网络请求是通过 Network 类实现的，它是一个抽象类，只有一个子类 BaseNetwork。

我们主要关注它的 performRequest 方法：

```
@Override
public NetworkResponse performRequest(Request<?> request) throws VolleyError {
    long requestStart = SystemClock.elapsedRealtime();
    while (true) {
        // 不断循环
        HttpResponse httpResponse = null;
        byte[] responseContents = null;
        List<Header> responseHeaders = Collections.emptyList();
        try {
            // 获取 Header
            Map<String, String> additionalRequestHeaders =
                getCacheHeaders(request.getCacheEntry());
            // 通过 httpStack.executeRequest 来同步执行网络请求
            httpResponse = mBaseHttpStack.executeRequest(request, additionalRequestHeaders);
            int statusCode = httpResponse.getStatusCode();
            responseHeaders = httpResponse.getHeaders();
            // 若没有 Modified，则构建一个 NetworkResponse 并返回
            if (statusCode == HttpURLConnection.HTTP_NOT_MODIFIED) {
                Entry entry = request.getCacheEntry();
                if (entry == null) {
                    return new NetworkResponse(
                        HttpURLConnection.HTTP_NOT_MODIFIED,
                        /* data= */ null,
                        /* notModified= */ true,
                        SystemClock.elapsedRealtime() - requestStart,
                        responseHeaders);
                }
                // 将 response 的 header 和缓存的 entry 结合，变成新的 Headers
                List<Header> combinedHeaders = combineHeaders(responseHeaders,
                    entry);
                return new NetworkResponse(
                    HttpURLConnection.HTTP_NOT_MODIFIED,
                    entry.data,
                    /* notModified= */ true,
                    SystemClock.elapsedRealtime() - requestStart,
```

```
        combinedHeaders);
    }
    // 有的 Response 没有 content, 因此需要对 content 进行判断
    InputStream inputStream = httpResponse.getContent();
    if (inputStream != null) {
        responseContents =
            inputStreamToBytes(inputStream, httpResponse.getContentLength());
    } else {
        // Add 0 byte response as a way of honestly representing a
        // no-content request.
        responseContents = new byte[0];
    }
    // if the request is slow, log it.
    long requestLifetime = SystemClock.elapsedRealtime() - requestStart;
    logSlowRequests(requestLifetime, request, responseContents, statusCode);
    if (statusCode < 200 || statusCode > 299) {
        throw new IOException();
    }
    // 返回请求结果构建的 Response
    return new NetworkResponse(
        statusCode,
        responseContents,
        /* notModified= */ false,
        SystemClock.elapsedRealtime() - requestStart,
        responseHeaders);
} catch (SocketTimeoutException e) {
    attemptRetryOnException("socket", request, new TimeoutError());
} catch (MalformedURLException e) {
    throw new RuntimeException("Bad URL " + request.getUrl(), e);
} catch (IOException e) {
    int statusCode;
    if (httpResponse != null) {
        statusCode = httpResponse.getStatusCode();
    } else {
        throw new NoConnectionError(e);
    }
    VolleyLog.e("Unexpected response code %d for %s", statusCode, request.getUrl());
    NetworkResponse networkResponse;
    if (responseContents != null) {
        networkResponse =
```

```
        new NetworkResponse(
            statusCode,
            responseContents,
            /* notModified= */ false,
            SystemClock.elapsedRealtime() - requestStart,
            responseHeaders);
    if (statusCode == HttpURLConnection.HTTP_UNAUTHORIZED
        || statusCode == HttpURLConnection.HTTP_FORBIDDEN) {
        attemptRetryOnException(
            "auth", request, new AuthFailureError(networkResponse));
    } else if (statusCode >= 400 && statusCode <= 499) {
        // Don't retry other client errors.
        throw new ClientError(networkResponse);
    } else if (statusCode >= 500 && statusCode <= 599) {
        if (request.shouldRetryServerErrors()) {
            attemptRetryOnException(
                "server", request, new ServerError(networkResponse));
        } else {
            throw new ServerError(networkResponse);
        }
    } else {
        // 3xx? No reason to retry.
        throw new ServerError(networkResponse);
    }
} else {
    attemptRetryOnException("network", request, new NetworkError());
}
}
```

这里的代码非常长，主要是下面的步骤：

1. 不断循环进行请求，直到出现错误或请求成功
 2. 获取 Headers
 3. 之后通过 `mBaseHttpStack` 进行网络请求
 4. 如果请求结果没有 `Modified`，则将返回的 Headers 与缓存的信息结合构建 `NetworkResponse` 并返回。

5. 最后如果请求成功，构建 NetworkResponse 并返回。
6. 若出现错误，如果是可以进行重试的（如 3xx 状态码或超时），会尝试进行重试。
7. 否则会根据具体错误构建对应的 Error 或 Response 并返回。

HttpStack

HttpStack 是我们真正执行网络请求的类，它有两个实现：HurlStack 以及 HttpClientStack，前者基于 HttpURLConnection 实现，后者基于 HttpClient 实现。

由于 HttpClient 已经被彻底抛弃，并且目前几乎已经不存在 SDK 9 以下的机器，因此我们只需要分析 HurlStack 即可，我们看到其 executeRequest 方法：

```
@Override
public HttpResponse executeRequest(Request<?> request, Map<String, String> additionalHeaders)
    throws IOException, AuthFailureError {
    String url = request.getUrl();
    HashMap<String, String> map = new HashMap<>();
    map.putAll(additionalHeaders);
    // Request.getHeaders() takes precedence over the given additional (cache) headers).
    map.putAll(request.getHeaders());
    if (mUrlRewriter != null) {
        String rewritten = mUrlRewriter.rewriteUrl(url);
        if (rewritten == null) {
            throw new IOException("URL blocked by rewriter: " + url);
        }
        url = rewritten;
    }
    URL parsedUrl = new URL(url);
    HttpURLConnection connection = openConnection(parsedUrl, request);
    boolean keepConnectionOpen = false;
```

```

try {
    for (String headerName : map.keySet()) {
        connection.setRequestProperty(headerName, map.get(headerName));
    }
    setConnectionParametersForRequest(connection, request);
    // Initialize HttpResponse with data from the HttpURLConnection.
    int responseCode = connection.getResponseCode();
    if (responseCode == -1) {
        // -1 is returned by getResponseCode() if the response code could not be retrieved.
        // Signal to the caller that something was wrong with the connection.
        throw new IOException("Could not retrieve response code from HttpURLConnection.");
    }
    if (!hasResponseBody(request.getMethod(), responseCode)) {
        return new HttpResponse(responseCode, convertHeaders(connection.getHeaderFields()));
    }
    // Need to keep the connection open until the stream is consumed by the caller. Wrap the
    // stream such that close() will disconnect the connection.
    keepConnectionOpen = true;
    return new HttpResponse(
        responseCode,
        convertHeaders(connection.getHeaderFields()),
        connection.getContentLength(),
        new UrlConnectionInputStream(connection));
} finally {
    if (!keepConnectionOpen) {
        connection.disconnect();
    }
}
}

```

这里没什么难度，就不再详细介绍，主要是调用了 `HttpURLConnection` 的 API 进行网络请求。

缓存机制

Volley 的缓存机制基于 `Cache` 这个接口实现，它对外暴露了 `put`、`get` 等常见的缓存操作接口。默认情况下采用基于磁盘的缓存 `DiskBasedCache`。

这一块主要是一些对文件的读取与写入，暂时就不研究了，有兴趣的读者可以自行阅读。

总结

为了在 `Application` 调用 `newRequestQueue` 同时又不被 `StrictMode` 有关文件操作相关的规则所影响，`Volley` 中使用了一个 `FileSupplier` 来对 `File` 进行包装，它采用了一个懒创建的思路，只有用到的时候才创建对应的 `cacheDir` 文件

`RequestQuque` 维护了三个队列，分别是待请求缓存队列、待网络请求队列以及正在请求队列。

`RequestQueue` 默认情况下维护了 4 个 `NetworkDispatcher` 以及 1 个 `CacheDispatcher`，它们都是继承自 `Thread`，通过异步进行网络请求的方式从而提高请求效率。

请求的成功或失败都会通过 `ExecutorDelivery` 进行交付，然后通过 `Request` 通知到各个 `Listener`。

存在一套缓存机制，以 `Request` 为 `Key`，以 `Response` 为 `Value`。只有 `GET` 请求需要进行缓存，所有 `GET` 请求会首先被放入缓存队列 `mCacheQueue`，当缓存未命中或缓存过期时，才会被放入 `mNetQueue` 进行网络请求。

`Network` 是一个对网络请求的执行进行包装的类，它主要负责了 `Response` 的转换以及对重试的支持。

`HttpStack` 是真正执行网络请求的类，它在高于 SDK 9 的版本下会基于 `HttpURLConnection` 进行网络请求，否则会基于 `HttpClient` 进行网络请求。

`Cache` 实现了 `Volley` 的缓存机制，默认情况下采用基于磁盘的缓存 `DiskBasedCache`。

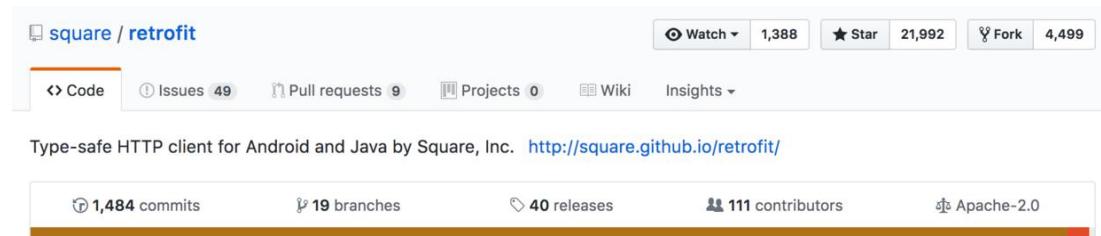
`Volley` 虽没有像 `OkHttp` 那样灵活的拦截器机制，但提供了很多 `Listener` 供外界对请求过程进行监听。

5. 深入解析 Retrofit 源码

前言

在 `Android` 开发中，网络请求十分常用

而在 `Android` 网络请求库中，`Retrofit` 是当下最热的一个网络请求库



- 今天，我将手把手带你深入剖析 `Retrofit v2.0` 的源码，希望你们会喜欢

在阅读本文前，建议先阅读文章：[这是一份很详细的 Retrofit 2.0 使用教](#)

[程（含实例讲解）](#)

目录



1. 简介

Retrofit简介	
介绍	一个 RESTful 的 HTTP 网络请求框架（基于OkHttp）
作者	Square
功能	<ul style="list-style-type: none">• 基于OkHttp & 遵循Restful API设计风格• 通过注解配置网络请求参数• 支持同步 & 异步网络请求• 支持多种数据的解析 & 序列化格式（Gson、Json、XML、Protobuf）• 提供对 RxJava支持
优点	<ul style="list-style-type: none">• 功能强大：支持同步 & 异步、支持多种数据的解析 & 序列化格式、支持RxJava• 简洁易用：通过注解配置网络请求参数、采用大量设计模式简化使用• 可拓展性好：功能模块高度封装、解耦彻底，如自定义Converters等等
应用场景	任何网络请求的需求场景都应优先选择 (特别是后台API遵循Restful API设计风格 & 项目中使用到RxJava)

特别注意：

- 准确来说，Retrofit 是一个 RESTful 的 HTTP 网络请求框架的封装。
- 原因：网络请求的工作本质上是 OkHttp 完成，而 Retrofit 仅负责网络请求接口的封装



- App 应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数、Header、Url 等信息，之后由 OkHttp 完成后续的请求操作
- 在服务端返回数据之后，OkHttp 将原始的结果交给 Retrofit，Retrofit 根据用户的需求对结果进行解析

2. 与其他网络请求开源库对比

除了 Retrofit，如今 Android 中主流的网络请求框架有：

- Android-Async-Http
- Volley
- OkHttp

下面是简单介绍：

网络请求开源库 - 介绍

网络请求库 /基础介绍	android-async-http	Volley	OkHttp	Retrofit
作者	Loopj	Google	Square	Square
面世时间	android-async-http > Volley > OkHttp > Retrofit			
人们使用情况 (GitHub Star数)	Volley > android-async-http > OkHttp > Retrofit			

一图让你了解全部的网络请求库和他们之间的区别！

网络请求库- 对比

网络请求库 / 对比	android-async-http	Volley	OkHttp	Retrofit
功能	<ul style="list-style-type: none"> • 基于HttpClient • 在 UI 线程外、异步进行Http请求 • 在匿名回调中处理请求结果 callback使用了Android的Handler发送消息机制在创建它的线程中执行 • 自动智能请求重试 • 持久化cookie存储 保存cookie到你的应用程序的 SharedPreferences 	<ul style="list-style-type: none"> • 基于HttpURLConnection • 封装了UI图片加载框架，支持图片加载 • 网络请求的排序、优先级处理 • 缓存 • 多级别取消请求 • Activity和生命周期的联动（Activity 结束时同时取消所有网络请求） 	<ul style="list-style-type: none"> • 高性能Http请求库 可把它理解成是一个封装之后的类似 HttpURLConnection 的一个东西，属于同级并不是基于上述二者； • 支持 SPDY，共享同一个Socket来处理同一个服务器的所有请求； • 支持http 2.0, websocket • 支持同步、异步 • 封装了线程池、数据转换、参数使用、错误处理等 • 无缝的支持GZIP来减少数据流量 • 缓存响应数据来减少重复的网络请求 • 能从很多常用的连接问题中自动恢复 • 解决了代理服务器问题和SSL握手失败问题 	<ul style="list-style-type: none"> • 基于OkHttp • RESTful API设计风格 • 支持同步、异步； • 通过注解配置请求 包括请求方法，请求参数，请求头，返回值等 • 可以搭配多种Converter将获得的数据解析&序列化 支持Gson（默认）、 Jackson、 Protobuf等 • 提供对 RxJava 的支持
性能		<ul style="list-style-type: none"> • 可拓展性好：可支持HttpClient、HttpURLConnection和OkHttp 	<ul style="list-style-type: none"> • 基于 NIO 和 Okio，所以性能更好：请求、处理速度快 (IO：阻塞式；NIO：非阻塞式；Okio 是 Square 公司基于 IO 和 NIO 基础上做的一个更简单、高效处理数据流的一个库) 	<ul style="list-style-type: none"> • 性能最好，处理最快； • 扩展性差 高度封装所带来的必然后果：解析数据都是使用的统一的converter，如果服务器不能给出统一的API的形式，将很难进行处理。
开发者使用	<p>1. 作者已经停止对该项目维护； 2. Android5.0后不推荐用 HttpClient； 所以不推荐在项目中使用了。</p>	<ul style="list-style-type: none"> • 封装性好：简单易用 	<ul style="list-style-type: none"> • Api调用更加简单、方便； • 使用时需要进行多一层封装 	<ul style="list-style-type: none"> • 简洁易用（Restful API设计风格） • 代码简化（更加高度的封装性和注解用法） • 解耦的更彻底、职责更细分 • 易与其他框架联合使用（RxJava） • 使用方法较多，原理复杂，存在一定门槛
应用场景		<ul style="list-style-type: none"> • 适合轻量级网络交互：网络请求频繁、传输数据量小； • 不能进行大数据量的网络操作（比如下载视频、音频），所以不适合用来上传文件。 	<p>重量级网络交互场景：网络请求频繁、传输数据量大 (其实会更推荐Retrofit，反正 Retrofit是基于Okhttp的)</p>	<p>任何场景下优先选择，特别是： 后台Api遵循RESTful的风格&项目中有使用RxJava；</p>
备注		<p>Volley的request和response都是把数据放到byte数组里，不支持输入输出流，把数据放到数组中，如果大文件多了，数组就会非常的大且多，消耗内存，所以不如直接返回Stream那样具备可操作性，比如下载一个大文件，不可能把整个文件都缓存到内存之后再写到文件里。</p>	<p>Android4.4的源码中可以看到 HttpURLConnection已经替换成 OkHttp实现了。所以我们更有理由相信OkHttp的强大。</p>	

附：各个主流网络请求库的 Github 地址

- [Android-Async-Http](#)
- [Volley](#)
- [OkHttp](#)

- Retrofit
-
-

3. Retrofit 的具体使用

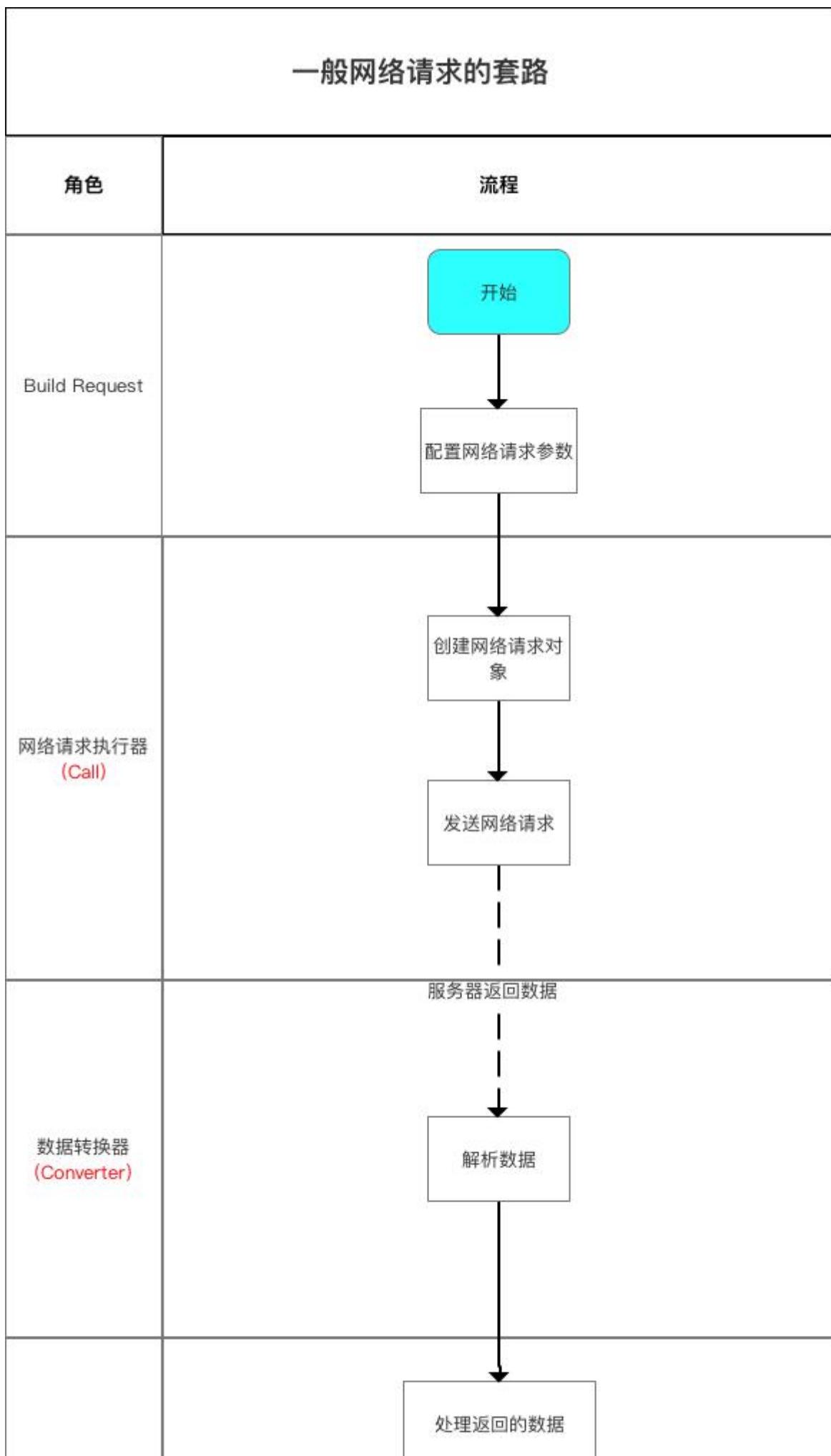
具体请看我写的文章：[这是一份很详细的 Retrofit 2.0 使用教程（含实例讲解）](#)

4. 源码分析

4.1 Retrofit 的本质流程

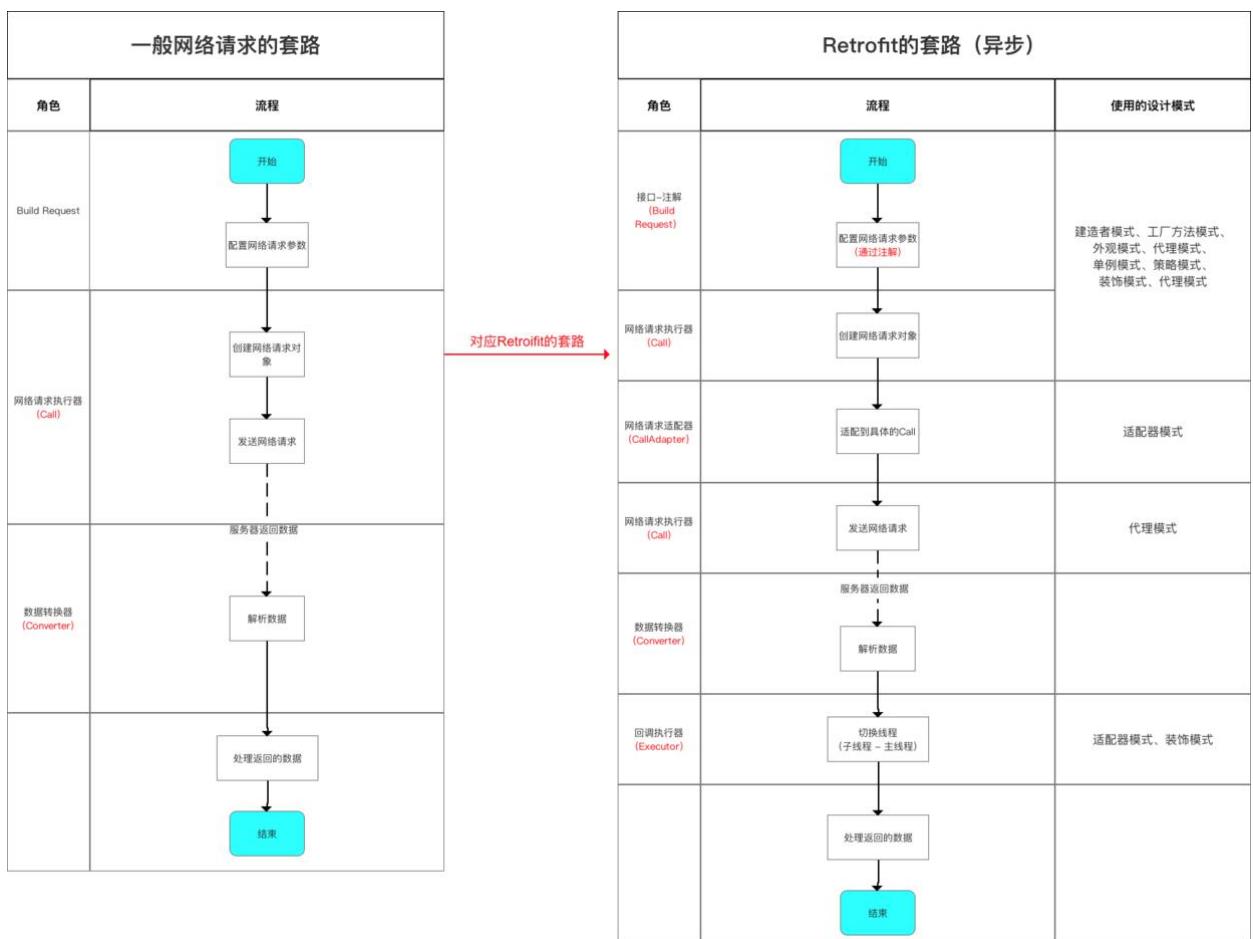
一般从网络通信过程如下图：

一般网络请求的套路



- 其实 Retrofit 的本质和上面是一样的套路
- 只是 Retrofit 通过使用**大量的设计模式进行功能模块的解耦**，使得上面的过程进行得更加简单 & 流畅

如下图：



具体过程解释如下：

1. 通过解析 网络请求接口的注解 配置 网络请求参数
2. 通过 动态代理 生成 网络请求对象
3. 通过 网络请求适配器 将 网络请求对象 进行平台适配

平台包括：Android、Rxjava、Guava 和 java8

1. 通过 网络请求执行器 发送网络请求
2. 通过 数据转换器 解析服务器返回的数据
3. 通过 回调执行器 切换线程 (子线程 ->>主线程)
4. 用户在主线程处理返回结果

下面介绍上面提到的几个角色

角色	作用	备注
网络请求执行器 <i>(Call)</i>	创建 HTTP网络请求	Retrofit默认为 okhttp3.Call
网络请求适配器 <i>(CallAdapter)</i>	网络请求执行器 (Call) 的适配器，将默认的网络请求执行器 (OkHttpCall) 转换成适合被不同平台来调用的网络请求执行器形式	<ul style="list-style-type: none"> • Retrofit支持Android、RxJava、java8 和 Guava 四个平台； • 所以在Retrofit中提供了四种CallAdapterFactory：AndroidCallAdapterFactory (Android默认) 、GuavaCallAdapterFactory、Java8CallAdapterFactory、RxJavaCallAdapterFactory； • 网络适配器作用的理解：如：一开始Retrofit只打算利用OkHttpCall通过ExecutorCallbackCall切换线程；但后来发现使用Rxjava更加方便（不需要Handler来切换线程）。想要实现Rxjava的情况，那就得使用RxJavaCallAdapterFactory将OkHttpCall转换成Rxjava(Scheduler)； • 好处：用最小代价兼容更多平台，即能适配更多的使用场景
数据转换器 <i>(Converter)</i>	将返回数据解析成我们需要的数据类型	Retrofit支持如XML、Gson、JSON、protobuf等等
回调执行器 <i>(CallBackExecutor)</i>	线程切换 (子线程 -> 主线程)	作用解释：将最后Okhttp的请求结果通过 callbackExecutor 使用Handler异步回调传回到主线程

特别注意：因下面的 源码分析 是根据 使用步骤 逐步带你 debug 进去的，所以必须先看文章[这是一份很详细的 Retrofit 2.0 使用教程\(含实例讲解 \)](#)

4. 2 源码分析

先来回忆 Retrofit 的使用步骤：

1. 创建 Retrofit 实例

2. 创建 网络请求接口实例 并 配置网络请求参数

3. 发送网络请求

封装了 数据转换、线程切换的操作

1. 处理服务器返回的数据

4.2.1 创建 Retrofit 实例

a. 使用步骤

```
Retrofit retrofit = new Retrofit.Builder()  
  
        .baseUrl("http://fanyi.youdao.com/")  
  
        .addConverterFactory(GsonConverterFactory.crea  
te())  
  
        .build();
```

b. 源码分析

Retrofit 实例是使用建造者模式通过 Builder 类进行创建的

建造者模式 : 将一个复杂对象的构建与表示分离 , 使得用户在不知道对象的

创建细节情况下就可以直接创建复杂的对象。具体请看文章 : [建造者模式](#)

(Builder Pattern) - 最易懂的设计模式解析

接下来 , 我将分五个步骤对创建 Retrofit 实例进行逐步分析

步骤1

步骤2

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://fanyi.youdao.com/") 步骤3

.addConverterFactory(GsonConverterFactory.create()) 步骤4

.build(); 步骤5

步骤 1

步骤1 步骤2

```
Retrofit retrofit = new Retrofit.Builder()  
    .baseUrl("http://fanyi.youdao.com/")      步骤3  
    .addConverterFactory(GsonConverterFactory.create())      步骤4  
    .build();      步骤5
```

<-- Retrofit 类 -->

```
public final class Retrofit {  
  
    private final Map<Method, ServiceMethod> serviceMethodCache = new  
    LinkedHashMap<>();  
  
    // 网络请求配置对象（对网络请求接口中方法注解进行解析后得到的对象）  
  
    // 作用：存储网络请求相关的配置，如网络请求的方法、数据转换器、网络请求适配器、  
    // 网络请求工厂、基地址等  
  
    private final HttpUrl baseUrl;  
  
    // 网络请求的 url 地址  
  
    private final okhttp3.Call.Factory callFactory;  
  
    // 网络请求器的工厂  
  
    // 作用：生产网络请求器（Call）  
  
    // Retrofit 是默认使用 okhttp  
  
    private final List<CallAdapter.Factory> adapterFactories;
```

```
// 网络请求适配器工厂的集合

// 作用：放置网络请求适配器工厂

// 网络请求适配器工厂作用：生产网络请求适配器（CallAdapter）

// 下面会详细说明

private final List<Converter.Factory> converterFactories;

// 数据转换器工厂的集合

// 作用：放置数据转换器工厂

// 数据转换器工厂作用：生产数据转换器（converter）

private final Executor callbackExecutor;

// 回调方法执行器

private final boolean validateEagerly;

// 标志位

// 作用：是否提前对业务接口中的注解进行验证转换的标志位

<-> Retrofit 类的构造函数 -->
```

```
Retrofit(okhttp3.Call.Factory callFactory, HttpUrl baseUrl,  
        List<Converter.Factory> converterFactories, List<CallAdapter.Factory>  
        adapterFactories,  
  
        Executor callbackExecutor, boolean validateEagerly) {  
  
    this.callFactory = callFactory;  
  
    this.baseUrl = baseUrl;  
  
    this.converterFactories = unmodifiableList(converterFactories);  
  
    this.adapterFactories = unmodifiableList(adapterFactories);  
  
    // unmodifiableList(list)类似于 UnmodifiableList<E>(list)  
  
    // 作用：创建的新对象能够对 list 数据进行访问，但不可通过该对象对 list 集合中的  
    // 元素进行修改  
  
    this.callbackExecutor = callbackExecutor;  
  
    this.validateEagerly = validateEagerly;  
  
    ...  
  
    // 仅贴出关键代码  
  
}
```

成功建立一个 Retrofit 对象的标准 配置好 Retrofit 类里的成员变量，即配置好：

- `serviceMethod`：包含所有网络请求信息的对象
- `baseUrl`：网络请求的 url 地址
- `callFactory`：网络请求工厂

- `adapterFactories` : 网络请求适配器工厂的集合
- `converterFactories` : 数据转换器工厂的集合
- `callbackExecutor` : 回调方法执行器

所谓 `xxxFactory`、 “`xxx` 工厂” 其实是设计模式中**工厂模式**的体现 : 将 “类实例化的操作” 与 “使用对象的操作” 分开 , 使得使用者不用知道具体参数就可以实例化出所需要的 “产品” 类。

具体请看我写的文章

[简单工厂模式 \(SimpleFactoryPattern \) - 最易懂的设计模式解析](#)

[工厂方法模式 \(Factory Method \) - 最易懂的设计模式解析](#)

[抽象工厂模式 \(Abstract Factory \) - 最易懂的设计模式解析](#)

这里详细介绍一下 : `CallAdapterFactory` : 该 `Factory` 生产的是 `CallAdapter` , 那么 `CallAdapter` 又是什么呢 ?

`CallAdapter` 详细介绍

- 定义 : 网络请求执行器 (Call) 的适配器
 1. `Call` 在 Retrofit 里默认是 `OkHttpCall`
 2. 在 Retrofit 中 提供 了 四 种 `CallAdapterFactory` :
`ExecutorCallAdapterFactory`(默认)、`GuavaCallAdapterFactory`、
`Java8CallAdapterFactory`、`RxJavaCallAdapterFactory`
- 作用 : 将默认的网络请求执行器 (`OkHttpCall`) 转换成适合被不同平台来调用的网络请求执行器形式

1. 如：一开始 `Retrofit` 只打算利用 `OkHttpCall` 通过 `ExecutorCallbackCall` 切换线程；但后来发现使用 `Rxjava` 更加方便（不需要 `Handler` 来切换线程）。想要实现 `Rxjava` 的情况，那就得使用 `RxJavaCallAdapterFactory` 将 `OkHttpCall` 转换成 `Rxjava(Scheduler)`：

```
// 把 response 封装成 rxjava 的 observable，然后进行流式操作
Retrofit.Builder.addCallAdapterFactory(new RxJavaCallAdapterFactory().create());
// 关于 RxJava 的使用这里不作更多的展开
```

1. `Retrofit` 还支持 `java8、Guava` 平台。
 - 好处：用最小代价兼容更多平台，即能适配更多的使用场景所以，接下来需要分析的步骤 2、步骤 3、步骤 4、步骤 4 的目的是配置好上述所有成员变量

步骤 2

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://fanyi.youdao.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build();
```

我们先来看 `Builder` 类

请按下面提示的步骤进行查看

```
<-- Builder 类-->

public static final class Builder {
```

```
private Platform platform;

private okhttp3.Call.Factory callFactory;

private HttpUrl baseUrl;

private List<Converter.Factory> converterFactories = new ArrayList<>();

private List<CallAdapter.Factory> adapterFactories = new ArrayList<>();

private Executor callbackExecutor;

private boolean validateEagerly;

// 从上面可以发现， Builder 类的成员变量与 Retrofit 类的成员变量是对应的

// 所以 Retrofit 类的成员变量基本上是通过 Builder 类进行配置

// 开始看步骤 1

<-- 步骤 1 -->

// Builder 的构造方法（无参）

public Builder() {

    this(Platform.get());

    // 用 this 调用自己的有参构造方法 public Builder(Platform platform) ->> 步骤 5 （看完步骤 2、3、4 再看）

    // 并通过调用 Platform.get() 传入了 Platform 对象

    // 继续看 Platform.get() 方法 ->> 步骤 2

    // 记得最后继续看步骤 5 的 Builder 有参构造方法
```

```
    }

    ...

}

<-- 步骤 2 -->

class Platform {

    private static final Platform PLATFORM = findPlatform();

    // 将 findPlatform()赋给静态变量

    static Platform get() {

        return PLATFORM;

        // 返回静态变量 PLATFORM, 即 findPlatform() ->>步骤 3

    }

}

<-- 步骤 3 -->

private static Platform findPlatform() {

    try {

        Class.forName("android.os.Build");

    }
```

```
// Class.forName("xxx.xx.xx")的作用：要求 JVM 查找并加载指定的类（即 JVM 会执行该类的静态代码段）

if (Build.VERSION.SDK_INT != 0) {

    return new Android();

    // 此处表示：如果是 Android 平台，就创建并返回一个 Android 对象 ->>步骤 4

}

} catch (ClassNotFoundException ignored) {

}

try {

    // 支持 Java 平台

    Class.forName("java.util.Optional");

    return new Java8();

} catch (ClassNotFoundException ignored) {

}

try {

    // 支持 iOS 平台

    Class.forName("org.robovm.apple.foundation.NSObject");

    return new IOS();

} catch (ClassNotFoundException ignored) {
```

```
    }

// 从上面看出：Retrofit2.0 支持 3 个平台：Android 平台、Java 平台、IOS 平台

// 最后返回一个 Platform 对象(指定了 Android 平台)给 Builder 的有参构造方法 public
Builder(Platform platform) --> 步骤 5

// 说明 Builder 指定了运行平台为 Android

    return new Platform();

}

...

}

<-- 步骤 4 -->

// 用于接收服务器返回数据后进行线程切换在主线程显示结果

static class Android extends Platform {

    @Override

    CallAdapter.Factory defaultCallAdapterFactory(Executor callbackExecutor) {

        return new ExecutorCallAdapterFactory(callbackExecutor);

    }

    // 创建默认的网络请求适配器工厂
```

```
// 该默认工厂生产的 adapter 会使得 Call 在异步调用时在指定的 Executor 上执行
回调

// 在 Retrofit 中提供了四种 CallAdapterFactory: ExecutorCallAdapterFactory(默
认)、GuavaCallAdapterFactory、Java8CallAdapterFactory、RxJavaCallAdapterFactory

// 采用了策略模式

}

@Override

public Executor defaultCallbackExecutor() {

    // 返回一个默认的回调方法执行器

    // 该执行器作用: 切换线程 (子->>主线程), 并在主线程 (UI 线程) 中执行回调
方法

    return new MainThreadExecutor();
}

static class MainThreadExecutor implements Executor {

    private final Handler handler = new Handler(Looper.getMainLooper());

    // 获取与 Android 主线程绑定的 Handler

    @Override
```

```
public void execute(Runnable r) {  
    handler.post(r);  
  
    // 该 Handler 是上面获取的与 Android 主线程绑定的 Handler  
  
    // 在 UI 线程进行对网络请求返回数据处理等操作。  
  
}  
  
}  
  
  
// 切换线程的流程:  
  
// 1. 回调 ExecutorCallAdapterFactory 生成了一个 ExecutorCallbackCall 对象  
  
//2. 通过调用 ExecutorCallbackCall.enqueue(CallBack)从而调用 MainThreadExecutor  
的 execute() 通过 handler 切换到主线程  
  
}  
  
  
  
  
// 下面继续看步骤 5 的 Builder 有参构造方法  
  
<-- 步骤 5 -->  
  
// Builder 类的构造函数 2 (有参)  
  
public Builder(Platform platform) {  
  
    // 接收 Platform 对象 (Android 平台)  
}
```

```
this.platform = platform;

// 通过传入 BuiltInConverters() 对象配置数据转换器工厂 (converterFactories)  
  
// converterFactories 是一个存放数据转换器 Converter.Factory 的数组  
  
// 配置 converterFactories 即配置里面的数据转换器  
  
    converterFactories.add(new BuiltInConverters());  
  
// BuiltInConverters 是一个内置的数据转换器工厂 (继承 Converter.Factory 类)  
  
// new BuiltInConverters() 是为了初始化数据转换器  
  
}
```

对 Builder 类分析完毕，总结：Builder 设置了默认的

- 平台类型对象：Android
- 网络请求适配器工厂：CallAdapterFactory

CallAdapter 用于对原始 Call 进行再次封装，如 Call 到 Observable

- 数据转换器工厂： converterFactory
- 回调执行器：callbackExecutor

**特别注意，这里只是设置了默认值，但未真正配置到具体的 Retrofit
类的成员变量当中**

步骤 3

步骤1 步骤2

```
Retrofit retrofit = new Retrofit.Builder()  
    .baseUrl("http://fanyi.youdao.com/") 步骤3  
    .addConverterFactory(GsonConverterFactory.create()) 步骤4  
    .build(); 步骤5
```

还是按部就班按步骤来观看

```
<-- 步骤 1 -->  
  
public Builder baseUrl(String baseUrl) {  
  
    // 把 String 类型的 url 参数转化为适合 OkHttpClient 的 HttpUrl 类型  
  
    HttpUrl httpUrl = HttpUrl.parse(baseUrl);  
  
    // 最终返回带 httpUrl 类型参数的 baseUrl ()  
  
    // 下面继续看 baseUrl(httpUrl) ->> 步骤 2  
  
    return baseUrl(httpUrl);  
  
}  
  
<-- 步骤 2 -->  
  
public Builder baseUrl(HttpUrl baseUrl) {
```

```
//把 URL 参数分割成几个路径碎片

List<String> pathSegments = baseUrl.pathSegments();

// 检测最后一个碎片来检查 URL 参数是不是以"/"结尾

// 不是就抛出异常

if (!"".equals(pathSegments.get(pathSegments.size() - 1))) {

    throw new IllegalArgumentException("baseUrl must end in /: " + baseUrl);

}

this.baseUrl = baseUrl;

return this;

}
```

- 至此，步骤 3 分析完毕
- 总结：**baseUrl() 用于配置 Retrofit 类的网络请求 url 地址**

将传入的 String 类型 url 转化为适合 OkHttpClient 的 HttpUrl 类型的 url

步骤 4

步骤1	步骤2
<pre>Retrofit retrofit = new Retrofit.Builder()</pre>	
	<pre>.baseUrl("http://fanyi.youdao.com/") 步骤3</pre>
	<pre>.addConverterFactory(GsonConverterFactory.create()) 步骤4</pre>
<pre>.build();</pre>	步骤5

我们从里往外看，即先看 `GsonConverterFactory.create()`

```
public final class GsonConverterFactory extends Converter.Factory {
```

```
<-- 步骤 1 -->

public static GsonConverterFactory create() {

    // 创建一个 Gson 对象

    return create(new Gson()); ->>步骤 2

}

<-- 步骤 2 -->

public static GsonConverterFactory create(Gson gson) {

    // 创建了一个含有 Gson 对象实例的 GsonConverterFactory

    return new GsonConverterFactory(gson); ->>步骤 3

}

private final Gson gson;

<-- 步骤 3 -->

private GsonConverterFactory(Gson gson) {

    if (gson == null) throw new NullPointerException("gson == null");

    this.gson = gson;

}
```

- 所以，`GsonConverterFactory.create()`是创建了一个含有 Gson 对象实例的 `GsonConverterFactory`，并返回给 `addConverterFactory(())`
- 接下来继续看：`addConverterFactory(())`

```
// 将上面创建的 GsonConverterFactory 放入到 converterFactories 数组

// 在第二步放入一个内置的数据转换器工厂 BuiltInConverters() 后又放入了一个
GsonConverterFactory

public Builder addConverterFactory(Converter.Factory factory) {

    converterFactories.add(checkNotNull(factory, "factory == null"));

    return this;

}
```

- 至此，分析完毕
- 总结：步骤 4 用于创建一个含有 Gson 对象实例的 `GsonConverterFactory` 并放入到数据转换器工厂 `converterFactories` 里
 1. 即 Retrofit 默认使用 Gson 进行解析

2. 若使用其他解析方式（如 Json、XML 或 Protobuf），也可通过自
定义数据解析器来实现（必须继承 Converter.Factory）

步骤 5

```
        Retrofit retrofit = new Retrofit.Builder()  
            .baseUrl("http://fanyi.youdao.com/")    步骤1  
            .addConverterFactory(GsonConverterFactory.create()) 步骤2  
        .build();    步骤3  
    步骤4  
    步骤5
```

终于到了最后一个步骤了。

```
public Retrofit build() {  
  
    <-- 配置网络请求执行器 (callFactory) -->  
  
    okhttp3.Call.Factory callFactory = this.callFactory;  
  
    // 如果没指定，则默认使用 okhttp  
  
    // 所以 Retrofit 默认使用 okhttp 进行网络请求  
  
    if (callFactory == null) {  
  
        callFactory = new OkHttpClient();  
  
    }  
  
    <-- 配置回调方法执行器 (callbackExecutor) -->  
  
    Executor callbackExecutor = this.callbackExecutor;  
  
    // 如果没指定，则默认使用 Platform 检测环境时的默认 callbackExecutor
```

```
// 即 Android 默认的 callbackExecutor

if (callbackExecutor == null) {

    callbackExecutor = platform.defaultCallbackExecutor();

}

<-- 配置网络请求适配器工厂 (CallAdapterFactory) -->

List<CallAdapter.Factory> adapterFactories = new
ArrayList<>(this.adapterFactories);

// 向该集合中添加了步骤 2 中创建的 CallAdapter.Factory 请求适配器 (添加在集
合器末尾)

adapterFactories.add(platform.defaultCallAdapterFactory(callbackExecutor));

// 请求适配器工厂集合存储顺序: 自定义 1 适配器工厂、自定义 2 适配器工厂...默认
适配器工厂 (ExecutorCallAdapterFactory)

<-- 配置数据转换器工厂: converterFactory -->

// 在步骤 2 中已经添加了内置的数据转换器 BuiltInConverters() (添加到集合器
的首位)

// 在步骤 4 中又插入了一个 Gson 的转换器 - GsonConverterFactory (添加到集合
器的首二位)

List<Converter.Factory> converterFactories = new
ArrayList<>(this.converterFactories);

// 数据转换器工厂集合存储的是: 默认数据转换器工厂 (BuiltInConverters) 、
自定义 1 数据转换器工厂 (GsonConverterFactory) 、自定义 2 数据转换器工厂....
```

```
// 注：

//1. 获取合适的网络请求适配器和数据转换器都是从 adapterFactories 和
converterFactories 集合的首位-末位开始遍历

// 因此集合中的工厂位置越靠前就拥有越高的使用权限

// 最终返回一个 Retrofit 的对象，并传入上述已经配置好的成员变量

    return new Retrofit(callFactory, baseUrl, converterFactories,
adapterFactories,

        callbackExecutor, validateEagerly);

}
```

- 至此，步骤 5 分析完毕
- 总结：在最后一步中，通过前面步骤设置的变量，将 Retrofit 类的所有成员变量都配置完毕。
- 所以，成功创建了 Retrofit 的实例

总结

Retrofit 使用建造者模式通过 **Builder** 类建立了一个 Retrofit 实例，具体创建细节是配置了：

- 平台类型对象 (Platform - Android)
- 网络请求的 url 地址 (baseUrl)

- 网络请求工厂 (callFactory)

默认使用 OkHttpCall

- 网络请求适配器工厂的集合 (adapterFactories)

本质是配置了网络请求适配器工厂 - 默认是 ExecutorCallAdapterFactory

- 数据转换器工厂的集合 (converterFactories)

本质是配置了数据转换器工厂

- 回调方法执行器 (callbackExecutor)

默认回调方法执行器作用是：切换线程（子线程 - 主线程）

由于使用了建造者模式，所以开发者并不需要关心配置细节就可以创建好 Retrofit 实例，建造者模式 get。

在创建 Retrofit 对象时，你可以通过更多更灵活的方式去处理你的需求，如使用不同的 Converter、使用不同的 CallAdapter，这也就提供了你使用 RxJava 来调用 Retrofit 的可能

2. 创建网络请求接口的实例

2.1 使用步骤

```
<-- 步骤 1: 定义接收网络数据的类 -->
<-- JavaBean.java -->
public class JavaBean {
    .. // 这里就不介绍了
```

```
}

<-- 步骤 2: 定义网络请求的接口类 -->

<-- AccessApi.java -->

public interface AccessApi {

    // 注解 GET: 采用 Get 方法发送网络请求

    // Retrofit 把网络请求的 URL 分成了 2 部分: 1 部分 baseUrl 放在创建 Retrofit 对象时设置; 另一部分在网络请求接口设置(即这里)

    // 如果接口里的 URL 是一个完整的网址, 那么放在创建 Retrofit 对象时设置的部分可以不设置

    @GET("openapi.do?keyfrom=Yanzhikai&key=2032414398&type=data&doctype=json&version=1.1&q=car")

    // 接受网络请求数据的方法

    Call<JavaBean> getCall();

    // 返回类型为 Call<*>, *是解析得到的数据类型, 即 JavaBean

}

<-- 步骤 3: 在 MainActivity 创建接口类实例 -->

AccessApi NetService = retrofit.create(AccessApi.class);
```

```
<-- 步骤 4: 对发送请求的 url 进行封装, 即生成最终的网络请求对象 -->
```

```
Call<JavaBean> call = NetService.getCall();
```

2.2 源码分析

- 结论 : Retrofit 是通过外观模式 & 代理模式 使用 create() 方
法创建网络请求接口的实例 (同时 , 通过网络请求接口里设置的注
解进行了网络请求参数的配置)

1. 外观模式 : 定义一个统一接口 , 外部与通过该统一的接口对子系统里

的其他接口进行访问。具体请看 : [外观模式 \(Facade Pattern \) - 最
易懂的设计模式解析](#)

2. 代理模式 : 通过访问代理对象的方式来间接访问目标对象。具体请看 :

[代理模式 \(Proxy Pattern \) - 最易懂的设计模式解析](#)

- 下面主要分析步骤 3 和步骤 4 :

```
<-- 步骤 3: 在 MainActivity 创建接口类实例 -->
```

```
AccessApi NetService = retrofit.create(NetService.class);
```

```
<-- 步骤 4: 对发送请求的 url 进行封装, 即生成最终的网络请求对象 -->
```

```
Call<JavaBean> call = NetService.getCall();
```

**步骤 3 讲解 : AccessApi NetService =
retrofit.create(NetService.class);**

```
public <T> T create(final Class<T> service) {
```

```
if (validateEagerly) {

    // 判断是否需要提前验证

    eagerlyValidateMethods(service);

    // 具体方法作用:

    // 1. 给接口中每个方法的注解进行解析并得到一个 ServiceMethod 对象

    // 2. 以 Method 为键将该对象存入 LinkedHashMap 集合中

    // 特别注意: 如果不是提前验证则进行动态解析对应方法(下面会详细说明), 得到
    // 一个 ServiceMethod 对象, 最后存入到 LinkedHashMap 集合中, 类似延迟加载(默认)

}

// 创建了网络请求接口的动态代理对象, 即通过动态代理创建网络请求接口
// 的实例 (并最终返回)

// 该动态代理是为了拿到网络请求接口实例上所有注解

return (T) Proxy.newProxyInstance(
    service.getClassLoader(),           // 动态生成接口的实现类
    new Class<?>[] { service },       // 动态创建实例
    new InvocationHandler() {          // 将代理类的实现交给 InvocationHandler
        // 作为具体的实现(下面会解释)
        private final Platform platform = Platform.get();
    }
);
```

```
// 在 InvocationHandler 类的 invoke() 实现中，除了执行真正的逻辑（如再次转发给真正的实现类对象），还可以进行一些有用的操作

// 如统计执行时间、进行初始化和清理、对接口调用进行检查等。

@Override

public Object invoke(Object proxy, Method method, Object... args)

throws Throwable {

    // 下面会详细介绍 invoke() 的实现

    // 即下面三行代码

    ServiceMethod serviceMethod = loadServiceMethod(method);

    OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);

    return serviceMethod.callAdapter.adapt(okHttpCall);

}

});

}

// 特别注意

// return (T) roxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces,
InvocationHandler invocationHandler)

// 可以解读为：getProxyClass(loader,
interfaces) .getConstructor(InvocationHandler.class).newInstance(invocationHandler);
```

```
// 即通过动态生成的代理类，调用 interfaces 接口的方法实际上是通过调用  
InvocationHandler 对象的 invoke() 来完成指定的功能

// 先记住结论，在讲解步骤 4 的时候会再次详细说明

<-- 关注点 1: eagerlyValidateMethods() -->

private void eagerlyValidateMethods(Class<?> service) {

    Platform platform = Platform.get();

    for (Method method : service.getDeclaredMethods()) {

        if (!platform.isDefaultMethod(method)) { loadServiceMethod(method); }

        // 将传入的 ServiceMethod 对象加入 LinkedHashMap<Method, ServiceMethod>集合

        // 使用 LinkedHashMap 集合的好处：lruEntries.values().iterator().next() 获取
        // 到的是集合最不经常用到的元素，提供了一种 LRU 算法的实现
    }
}
```

创建网络接口实例用了外观模式 & 代理模式：

使用外观模式进行访问，里面用了代理模式

1. 外观模式

外观模式：定义一个统一接口，外部与通过该统一的接口对子系统里的其他接口进行访问。具体请看：[外观模式（Facade Pattern） - 最易懂的设计模式解析](#)

Retrofit 对象的外观（门店） = `retrofit.create()`

通过这一外观方法就可以在内部调用各个方法创建网络请求接口的实例和配置网络请求参数

大大降低了系统的耦合度

2. 代理模式

- 代理模式：通过访问代理对象的方式来间接访问目标对象

分为静态代理 & 动态代理：

1. 静态代理：代理类在程序运行前已经存在的代理方式

2. 动态代理：代理类在程序运行前不存在、运行时由程序动态生成的代理方式

具体请看文章[代理模式（Proxy Pattern） - 最易懂的设计模式解析](#)

- `return (T) proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler invocationHandler)` 通过代理模式中的动态代理模式，动态生成网络请求接口的代理类，并将代理类的实例创建交给 `InvocationHandler` 类 作为具体的实现，并最终返回一个动态代理对象。

生成实例过程中含有生成实现类的缓存机制（单例模式），下面会详细分析

使用动态代理的好处：

- 当 `NetService` 对象调用 `getCall()` 接口中方法时会进行拦截，调用都会集中转发到 `InvocationHandler#invoke()`，可集中进行处理

- 获得网络请求接口实例上的所有注解
- 更方便封装 ServiceMethod

下面看源码分析

下面将详细分析 `InvocationHandler` 类 `# invoke()` 里的具体实现

```
new InvocationHandler() {

    private final Platform platform = Platform.get();

    @Override
    public Object invoke(Object proxy, Method method, Object... args)
            throws Throwable {
        // 将详细介绍下面代码
        // 关注点 1
        // 作用：读取网络请求接口里的方法，并根据前面配置好的属性配置
        serviceMethod 对象
        ServiceMethod serviceMethod = loadServiceMethod(method);

        // 关注点 2
        // 作用：根据配置好的 serviceMethod 对象创建 okHttpCall 对象
        OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
    }
}
```

```
// 关注点 3

        // 作用：调用 OkHttp，并根据 okHttpCall 返回 rejava 的 Observe 对象或者
        // 返回 Call

        return serviceMethod.callAdapter.adapt(okHttpCall);

    }
```

下面将详细介绍 3 个关注点的代码。

关注点 1：**ServiceMethod serviceMethod = loadServiceMethod(method);**

```
<-- loadServiceMethod(method)方法讲解 -->

// 一个 ServiceMethod 对象对应于网络请求接口里的一个方法

// loadServiceMethod (method) 负责加载 ServiceMethod:

ServiceMethod loadServiceMethod(Method method) {

    ServiceMethod result;

    // 设置线程同步锁

    synchronized (serviceMethodCache) {

        result = serviceMethodCache.get(method);

        // ServiceMethod 类对象采用了单例模式进行创建
    }
}
```

```
// 即创建 ServiceMethod 对象前，先看 serviceMethodCache 有没有缓存之前创建过的网络请求实例

// 若没缓存，则通过建造者模式创建 serviceMethod 对象

if (result == null) {

    // 下面会详细介绍 ServiceMethod 生成实例的过程

    result = new ServiceMethod.Builder(this, method).build();

    serviceMethodCache.put(method, result);

}

return result;

}

// 这里就是上面说的创建实例的缓存机制：采用单例模式从而实现一个 ServiceMethod 对象对于网络请求接口里的一个方法

// 注：由于每次获取接口实例都是传入 class 对象

// 而 class 对象在进程内单例的，所以获取到它的同一个方法 Method 实例也是单例的，所以这里的缓存是有效的。
```

下面，我将分 3 个步骤详细分析 `serviceMethod` 实例的创建过程：

步骤1	步骤2	步骤3
<code>result = new ServiceMethod.Builder(this, method).build();</code>		

步骤 1：`ServiceMethod` 类 构造函数

步骤1 步骤2 步骤3

```
result = new ServiceMethod.Builder(this, method).build();
```

```
<-- ServiceMethod 类 -->

public final class ServiceMethod {

    final okhttp3.Call.Factory callFactory; // 网络请求工厂

    final CallAdapter<?> callAdapter;

    // 网络请求适配器工厂

    // 具体创建是在 new ServiceMethod.Builder(this, method).build()最后的 build()中

    // 下面会详细说明

    private final Converter<ResponseBody, T> responseConverter;

    // Response 内容转换器

    // 作用：负责把服务器返回的数据（JSON 或者其他格式，由 ResponseBody 封装）转化为
    // T 类型的对象；

    private final HttpUrl baseUrl; // 网络请求地址

    private final String relativeUrl; // 网络请求的相对地址

    private final String httpMethod; // 网络请求的 Http 方法

    private final Headers headers; // 网络请求的 http 请求头 键值对

    private final MediaType contentType; // 网络请求的 http 报文 body 的类型
```

```
private final ParameterHandler<?>[] parameterHandlers;

// 方法参数处理器

// 作用：负责解析 API 定义时每个方法的参数，并在构造 HTTP 请求时设置参数；

// 下面会详细说明

// 说明：从上面的成员变量可以看出，ServiceMethod 对象包含了访问网络的所有基本信息

<-> ServiceMethod 类的构造函数 -->

// 作用：传入各种网络请求参数

ServiceMethod(Builder<T> builder) {

    this.callFactory = builder.retrofit.callFactory();

    this.setAdapter = builder.setAdapter;

    this.responseConverter = builder.responseConverter;

    this.baseUrl = builder.retrofit.baseUrl();

    this.relativeUrl = builder.relativeUrl;

    this.httpMethod = builder.httpMethod;

    this.headers = builder.headers;
```

```
this.contentType = builder.contentType; .  
  
this.hasBody = builder.hasBody; y  
  
this.isFormEncoded = builder.isFormEncoded;  
  
this.isMultipart = builder.isMultipart;  
  
this.parameterHandlers = builder.parameterHandlers;  
  
}  
  
}
```

步骤 2 : ServiceMethod 的 Builder ()

```
result = new ServiceMethod.Builder(this, method).build();
```

```
public Builder(Retrofit retrofit, Method method) {  
  
    this.retrofit = retrofit;  
  
    this.method = method;  
  
    // 获取网络请求接口方法里的注释  
  
    this.methodAnnotations = method.getAnnotations();  
  
    // 获取网络请求接口方法里的参数类型  
  
    this.parameterTypes = method.getGenericParameterTypes();  
}
```

```
//获取网络请求接口方法里的注解内容

this.parameterAnnotationsArray = method.getParameterAnnotations();

}
```

步骤 3 : ServiceMethod 的 build()

步骤1 步骤2 步骤3

```
result = new ServiceMethod.Builder(this, method).build();
```

// 作用：控制 ServiceMethod 对象的生成流程

```
public ServiceMethod build() {
```

```
    callAdapter = createCallAdapter();
```

// 根据网络请求接口方法的返回值和注解类型，从 Retrofit 对象中获取对应的网络请求适配器 -->关注点 1

```
    responseType = callAdapter.responseType();
```

// 根据网络请求接口方法的返回值和注解类型，从 Retrofit 对象中获取该网络适配器返回的数据类型

```
    responseConverter = createResponseConverter();
```

// 根据网络请求接口方法的返回值和注解类型，从 Retrofit 对象中获取对应的数据转换器 -->关注点 3

```
// 构造 HTTP 请求时，我们传递的参数都是 String

// Retrofit 类提供 converter 把传递的参数都转化为 String

// 其余类型的参数都利用 Converter.Factory 的 stringConverter 进行转换

// @Body 和 @Part 类型的参数利用 Converter.Factory 提供的
requestBodyConverter 进行转换

// 这三种 converter 都是通过“询问”工厂列表进行提供，而工厂列表我们在构造
// Retrofit 对象时进行添加。

for (Annotation annotation : methodAnnotations) {

    parseMethodAnnotation(annotation);

}

// 解析网络请求接口中方法的注解

// 主要是解析获取 Http 请求的方法

// 注解包括：DELETE、GET、POST、HEAD、PATCH、PUT、OPTIONS、HTTP、
retrofit2.http.Headers、Multipart、FormUrlEncoded

// 处理主要是调用方法 parseHttpMethodAndPath(String httpMethod, String
value, boolean hasBody) ServiceMethod 中的 httpMethod、hasBody、relativeUrl、
relativeUrlParamNames 域进行赋值

int parameterCount = parameterAnnotationsArray.length;

// 获取当前方法的参数数量
```

```
parameterHandlers = new ParameterHandler<?>[parameterCount];

for (int p = 0; p < parameterCount; p++) {

    Type parameterType = parameterTypes[p];

    Annotation[] parameterAnnotations = parameterAnnotationsArray[p];

    // 为方法中的每个参数创建一个 ParameterHandler<?>对象并解析每个参数
    // 使用的注解类型

    // 该对象的创建过程就是对方法参数中注解进行解析

    // 这里的注解包括: Body、PartMap、Part、FieldMap、Field、Header、QueryMap、
    // Query、Path、Url

    parameterHandlers[p] = parseParameter(p, parameterType,
    parameterAnnotations);

}

return new ServiceMethod<>(this);

<-- 总结 -->

// 1. 根据返回值类型和方法标注从 Retrofit 对象的的网络请求适配器工厂集合和内容转
// 换器工厂集合中分别获取到该方法对应的网络请求适配器和 Response 内容转换器;

// 2. 根据方法的标注对 ServiceMethod 的域进行赋值

// 3. 最后为每个方法的参数的标注进行解析, 获得一个 ParameterHandler<?>对象

// 该对象保存有一个 Request 内容转换器—根据参数的类型从 Retrofit 的内容转换器工厂
// 集合中获取一个 Request 内容转换器或者一个 String 内容转换器。

}
```

```
<-- 关注点 1: createCallAdapter() -->

private CallAdapter<?> createCallAdapter() {

    // 获取网络请求接口里方法的返回值类型

    Type returnType = method.getGenericReturnType();

    // 获取网络请求接口接口里的注解

    // 此处使用的是@Get

    Annotation[] annotations = method.getAnnotations();

    try {

        return retrofit.callAdapter(returnType, annotations);

        // 根据网络请求接口方法的返回值和注解类型, 从 Retrofit 对象中获取对应的网络
        // 请求适配器

        // 下面会详细说明 retrofit.callAdapter () -->关注点 2

    }

    ...

<-- 关注点 2: retrofit.callAdapter() -->
```

```
public CallAdapter<?> callAdapter(Type returnType, Annotation[] annotations) {  
    return nextCallAdapter(null, returnType, annotations);  
}  
  
public CallAdapter<?> nextCallAdapter(CallAdapter.Factory skipPast, Type  
returnType,  
    Annotation[] annotations) {  
  
    // 创建 CallAdapter 如下  
  
    // 遍历 CallAdapter.Factory 集合寻找合适的工厂（该工厂集合在第一步构造  
Retrofit 对象时进行添加（第一步时已经说明））  
  
    // 如果最终没有工厂提供需要的 CallAdapter，将抛出异常  
  
    for (int i = start, count = adapterFactories.size(); i < count; i++) {  
  
        CallAdapter<?> adapter = adapterFactories.get(i).get(returnType,  
annotations, this);  
  
        if (adapter != null) {  
  
            return adapter;  
        }  
    }  
}
```

```
private Converter<ResponseBody, T> createResponseConverter() {  
  
    Annotation[] annotations = method.getAnnotations();  
  
    try {  
  
        // responseConverter 还是由 Retrofit 类提供 ->关注点 4  
  
        return retrofit.responseBodyConverter(responseType, annotations);  
  
    } catch (RuntimeException e) {  
  
        throw methodError(e, "Unable to create converter for %s", responseType);  
  
    }  
  
}  
  
<-- 关注点 4: responseBodyConverter () -->  
  
public <T> Converter<ResponseBody, T> responseBodyConverter(Type type,  
Annotation[] annotations) {  
  
    return nextResponseBodyConverter(null, type, annotations);  
  
}  
  
public <T> Converter<ResponseBody, T>  
nextResponseBodyConverter(Converter.Factory skipPast,
```

```
int start = converterFactories.indexOf(skipPast) + 1;

for (int i = start, count = converterFactories.size(); i < count; i++) {

    // 获取 Converter 过程: (和获取 callAdapter 基本一致)

    Converter<ResponseBody, ?> converter =
        converterFactories.get(i).responseBodyConverter(type, annotations,
this);

    // 遍历 Converter.Factory 集合并寻找合适的工厂(该工厂集合在构造 Retrofit
对象时进行添加(第一步时已经说明))

    // 由于构造 Retrofit 采用的是 Gson 解析方式, 所以取出的是
GsonResponseBodyConverter

    // Retrofit - Converters 还提供了 JSON, XML, ProtoBuf 等类型数据的转换功
能。

    // 继续看 responseBodyConverter () -->关注点 5

}

<--  关注点 5: responseBodyConverter () -->

@Override

public Converter<ResponseBody, ?> responseBodyConverter(Type type,
Annotation[] annotations, Retrofit retrofit) {
```

```
TypeAdapter<?> adapter = gson.getAdapter(TypeToken.get(type));

// 根据目标类型，利用 Gson#getAdapter 获取相应的 adapter

return new GsonResponseBodyConverter<>(gson, adapter);

}

// 做数据转换时调用 Gson 的 API 即可。

final class GsonResponseBodyConverter<T> implements Converter<ResponseBody, T> {

    private final Gson gson;

    private final TypeAdapter<T> adapter;

    GsonResponseBodyConverter(Gson gson, TypeAdapter<T> adapter) {

        this.gson = gson;

        this.adapter = adapter;

    }

    @Override

    public T convert(ResponseBody value) throws IOException {

        JsonReader jsonReader = gson.newJsonReader(value.charStream());

        try {

            return adapter.read(jsonReader, gson.fromJsonValue(value));
        } finally {
            jsonReader.close();
        }
    }
}
```

```
        return adapter.read(jsonReader);

    } finally {
        value.close();
    }
}
```

- 当选择了 RxjavaCallAdapterFactory 后，Rxjava 通过策略模式选择对应的 adapter

关于策略模式的讲解，请看文章[策略模式 \(Strategy Pattern \) - 最易懂的](#)

[设计模式解析](#)

- 具体过程是：根据网络接口方法的返回值类型来选择具体要用哪种 CallAdapterFactory，然后创建具体的 CallAdapter 实例

采用工厂模式使得各功能模块高度解耦

- 上面提到了两种工厂：CallAdapter.Factory & Converter.Factory 分别负责提供不同的功能模块
- 工厂负责如何提供、提供何种功能模块
- Retrofit 只负责提供选择何种工厂的决策信息（如网络接口方法的参数、返回值类型、注解等）

这正是所谓的高内聚低耦合，工厂模式 get。

关于工厂模式请看我写的文章：

[简单工厂模式 \(SimpleFactoryPattern \) - 最易懂的设计模式解析](#)

[工厂方法模式 \(Factory Method \) - 最易懂的设计模式解析](#)

[抽象工厂模式 \(Abstract Factory \) - 最易懂的设计模式解析](#)

终于配置完网络请求参数（即配置好 `ServiceMethod` 对象）。接下来将讲

解第二行代码：`okHttpCall` 对象的创建

第二行：`OkHttpCall okHttpCall = new`

`OkHttpCall<>(serviceMethod, args);`

根据第一步配置好的 `ServiceMethod` 对象和输入的请求参数创建 `okHttpCall`

对象

```
<--OkHttpCall 类 -->

public class OkHttpCall {

    private final ServiceMethod<T> serviceMethod; // 含有所有网络请求参数信息的对象

    private final Object[] args; // 网络请求接口的参数

    private okhttp3.Call rawCall; //实际进行网络访问的类

    private Throwable creationFailure; //几个状态标志位

    private boolean executed;

    private volatile boolean canceled;

    <--OkHttpCall 构造函数 -->
```

```
public OkHttpCall<T> serviceMethod, Object[] args) {  
  
    // 传入了配置好的 ServiceMethod 对象和输入的请求参数  
  
    this.serviceMethod = serviceMethod;  
  
    this.args = args;  
  
}  
}
```

第三行：return

```
serviceMethod.callAdapter.adapt(okHttpCall);
```

将第二步创建的 `OkHttpCall` 对象传给第一步创建的 `serviceMethod` 对象中对应的网络请求适配器工厂的 `adapt()`

返回对象类型：Android 默认的是 Call<>；若设置了

RxJavaCallAdapterFactory，返回的则是 Observable<>

```
<-- adapt () 详解-->

public <R> Call<R> adapt(Call<R> call) {
    return new ExecutorCallbackCall<>(callbackExecutor, call);
}

}

ExecutorCallbackCall(Executor callbackExecutor, Call<T> delegate) {
    this.delegate = delegate;
    // 把上面创建并配置好参数的 OkhttpCall 对象交给静态代理 delegate
}

// 静态代理和动态代理都属于代理模式
```

```
// 静态代理作用：代理执行被代理者的方法，且可在要执行的方法前后加入自己的动作，进行对系统功能的拓展
```

```
this.callbackExecutor = callbackExecutor;  
  
// 传入上面定义的回调方法执行器  
  
// 用于进行线程切换  
  
}
```

- 采用了**装饰模式**：ExecutorCallbackCall = 装饰者，而里面真正去执行网络请求的还是 OkHttpCall
- 使用装饰模式的原因：希望在 OkHttpCall 发送请求时做一些额外操作。这里的额外操作是线程转换，即将子线程切换到主线程
 1. OkHttpCall 的 enqueue() 是进行网络异步请求的：当你调用 OkHttpCall.enqueue() 时，回调的 callback 是在子线程中，需要通过 Handler 转换到主线程进行回调。ExecutorCallbackCall 就是用于线程回调；
 2. 当然以上是原生 Retrofit 使用的切换线程方式。如果你用 Rxjava，那就不会用到这个 ExecutorCallbackCall 而是 RxJava 的 Call，此处不过多展开

步骤 4 讲解：Call<JavaBean> call =

NetService.getCall();

- `NetService` 对象实际上是动态代理对象 `Proxy.newProxyInstance()` (步骤 3 中已说明) , 并不是真正的网络请求接口创建的对象
- 当 `NetService` 对象调用 `getCall()` 时会被动态代理对象 `Proxy.newProxyInstance()` 拦截 , 然后调用自身的 `InvocationHandler#invoke()`
- `invoke(Object proxy, Method method, Object... args)` 会传入 3 个参数 :
 - `Object proxy` : (代理对象) 、
 - `Method method` (调用的 `getCall()`)
 - `Object... args` (方法的参数 , 即 `getCall(*)` 中的 *)
- 接下来利用 Java 反射获取到 `getCall()` 的注解信息 , 配合 `args` 参数创建 `ServiceMethod` 对象。

如上面步骤 3 描述 , 此处不再次讲解

最终创建并返回一个 `OkHttpCall` 类型的 Call 对象

1. `OkHttpCall` 类是 `OkHttp` 的包装类
2. 创建了 `OkHttpCall` 类型的 Call 对象还不能发送网络请求 , 需要创建 `Request` 对象才能发送网络请求

总结

Retrofit 采用了 外观模式 统一调用创建网络请求接口实例和网络请求参数配置的方法 , 具体细节是 :

- 动态创建网络请求接口的实例** (代理模式 - 动态代理) **

- 创建 `serviceMethod` 对象** (建造者模式 & 单例模式 (缓存机制))
**
 - 对 `serviceMethod` 对象进行网络请求参数配置：通过解析网络请求接口方法的参数、返回值和注解类型，从 Retrofit 对象中获取对应的网络请求的 url 地址、网络请求执行器、网络请求适配器 & 数据转换器。 (策略模式)
 - 对 `serviceMethod` 对象加入线程切换的操作，便于接收数据后通过 Handler 从子线程切换到主线程从而对返回数据结果进行处理** (装饰模式) **
 - 最终创建并返回一个 `OkHttpCall` 类型的网络请求对象
-
-

3. 执行网络请求

- Retrofit 默认使用 `OkHttp`，即 `OkHttpCall` 类 (实现了 `retrofit2.Call<T>` 接口)
但可以自定义选择自己需要的 Call 类
- `OkHttpCall` 提供了两种网络请求方式：
 - 同步请求 : `OkHttpCall.execute()`
 - 异步请求 : `OkHttpCall.enqueue()`

下面将详细介绍这两种网络请求方式。

对于 OkHttpCall 的 enqueue()、execute() 此处不往下分析，有兴趣

的读者可以看 OkHttpClient 的源码

3.1 同步请求 OkHttpClient.execute()

3.1.1 发送请求过程

- **步骤 1**：对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 ServiceMethod 对象创建一个 OkHttpClient 的 Request 对象

- **步骤 2**：使用 OkHttpClient 的 Request 发送网络请求；
- **步骤 3**：对返回的数据使用之前设置的数据转换器（ GsonConverterFactory ）解析返回的数据，最终得到一个 Response<T> 对象

3.1.2 具体使用

```
Response<JavaBean> response = call.execute();
```

上面简单的一行代码，其实包含了整个发送网络同步请求的三个步骤。

3.1.3 源码分析

```
@Override  
  
public Response<T> execute() throws IOException {  
  
    okhttp3.Call call;
```

```
// 设置同步锁

synchronized (this) {

    call = rawCall;

    if (call == null) {

        try {

            call = rawCall = createRawCall();

            // 步骤 1：创建一个 OkHttpClient 的 Request 对象请求 ->关注 1

        } catch (IOException | RuntimeException e) {

            creationFailure = e;

            throw e;

        }

    }

}

return parseResponse(call.execute());

// 步骤 2：调用 OkHttpClient 的 execute() 发送网络请求（同步）

// 步骤 3：解析网络请求返回的数据 parseResponse () ->关注 2

}
```

```
<-- 关注 1: createRawCall() -->

private okhttp3.Call createRawCall() throws IOException {

    Request request = serviceMethod.toRequest(args);

    // 从 ServiceMethod 的 toRequest () 返回一个 Request 对象

    okhttp3.Call call = serviceMethod.callFactory.newCall(request);

    // 根据 serviceMethod 和 request 对象创建 一个 okhttp3.Request

    if (call == null) {

        throw new NullPointerException("Call.Factory returned null.");

    }

    return call;

}

<-- 关注 2: parseResponse () -->

Response<T> parseResponse(okhttp3.Response rawResponse) throws IOException {

    ResponseBody rawBody = rawResponse.body();

    rawResponse = rawResponse.newBuilder()

        .body(new NoContentResponseBody(rawBody.contentType(),
rawBody.contentLength()))
}
```

```
.build();

// 收到返回数据后进行状态码检查

// 具体关于状态码说明下面会详细介绍

int code = rawResponse.code();

if (code < 200 || code >= 300) {

}

if (code == 204 || code == 205) {

    return Response.success(null, rawResponse);
}

ExceptionCatchingRequestBody catchingBody = new
ExceptionCatchingRequestBody(rawBody);

try {

    T body = serviceMethod.toResponse(catchingBody);

    // 等 Http 请求返回后 & 通过状态码检查后，将 response body 传入 ServiceMethod
    // 中，ServiceMethod 通过调用 Converter 接口（之前设置的 GsonConverterFactory）将
    // response body 转成一个 Java 对象，即解析返回的数据

// 生成 Response 类

    return Response.success(body, rawResponse);
}
```

```
        } catch (RuntimeException e) {  
  
            ... // 异常处理  
  
        }  
  
    }  
  
}
```

特别注意：

- `ServiceMethod` 几乎保存了一个网络请求所需要的数据
- 发送网络请求时，`OkHttpCall` 需要从 `ServiceMethod` 中获得一个 `Request` 对象
- 解析数据时，还需要通过 `ServiceMethod` 使用 `Converter`（数据转换器）转换成 Java 对象进行数据解析

为了提高效率，Retrofit 还会对解析过的请求 `ServiceMethod` 进行缓存，存

放在 `Map<Method, ServiceMethod> serviceMethodCache = new`

`LinkedHashMap<>();` 对象中，即第二步提到的单例模式

- 关于状态码检查时的状态码说明：

类型		含义
成功 (2开头)	200	OK: 请求成功、其后是对GET和POST请求的应答文档
	202	Accepted: 供处理的请求已被接受，但是处理未完成。
	204	No Content: 没有新文档。浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而Servlet可以确定用户文档足够新，这个状态代码是很有用的。
	205	Reset Content: 没有新文档。但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容。
	206	Partial Content: 断点续传，客户发送了一个带有Range头的GET请求，服务器完成了它。
重定向 (3开头)	300	Multiple Choices: 多重选择。链接列表。用户可以选择某链接到达目的地。最多允许五个地址。
	301	Moved Permanently: 所请求的页面已经转移至新的url。
	302	Move temporarily: 所请求的页面临时转移至新的url。如果这不是一个 GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认
	304	Not Modified: 客户端有缓冲的文档并发出一个条件性的请求（一般是提供If-Modified-Since头表示客户只想比指定日期更新的文档）。服务器告诉客户，原来缓冲的文档还可以继续使用。
请求错误 (4开头)	400	Bad Request: 语义有误、请求参数有误。
	403	Forbidden: 服务器已经理解请求，但是拒绝执行它。
	404	Not Found: 请求失败，请求所希望得到的资源未被在服务器上发现
服务器错误 (5、6开头)	500	Internal Server Error: 请求未完成。服务器遇到不可预知的情况。

以上便是整个以**同步的方式**发送网络请求的过程。

3.2 异步请求 `OkHttpCall.enqueue()`

3.2.1 发送请求过程

- **步骤 1**：对网络请求接口的方法中的每个参数利用对应
的 `ParameterHandler` 进行解析，再根据 `ServiceMethod` 对象创建一个 `OkHttp`
的 `Request` 对象
- **步骤 2**：使用 `OkHttp` 的 `Request` 发送网络请求；
- **步骤 3**：对返回的数据使用之前设置的数据转换器
(`GsonConverterFactory`) 解析返回的数据，最终得到一个
`Response<T>` 对象
- **步骤 4**：进行线程切换从而在主线程处理返回的数据结果
若使用了 RxJava，则直接回调到主线程

异步请求的过程跟同步请求类似，唯一不同之处在于：异步请求会将回调方法交给回调执行器在指定的线程中执行。

指定的线程此处是指主线程 (UI 线程)

3.2.2 具体使用

```
call.enqueue(new Callback<JavaBean>() {  
  
    @Override  
  
    public void onResponse(Call<JavaBean> call, Response<JavaBean>  
response) {  
  
        System.out.println(response.isSuccessful());  
  
        if (response.isSuccessful()) {  
  
            response.body().show();  
  
        }  
    }  
})
```

```
        else {

            try {

                System.out.println(response.errorBody().string());

            } catch (IOException e) {

                e.printStackTrace();

            } ;

        }

    }
```

- 从上面分析有：`call` 是一个静态代理
- 使用静态代理的作用是：在 `okhttpCall` 发送网络请求的前后进行
额外操作

这里的额外操作是：线程切换，即将子线程切换到主线程，从而在主线程对

返回的数据结果进行处理

3.2.3 源码分析

```
<-- call.enqueue () 解析 -->

@Override

public void enqueue(final Callback<T> callback) {

    ...

    delegate.enqueue(new Callback<T>() {

        // 使用静态代理 delegate 进行异步请求 ->>分析 1

    });

}
```

```
// 等下记得回来

@Override

    public void onResponse(Call<T> call, final Response<T> response)
{

    // 步骤 4：线程切换，从而在主线程显示结果

    callbackExecutor.execute(new Runnable() {

        // 最后 Okhttp 的异步请求结果返回到 callbackExecutor

        // callbackExecutor.execute() 通过 Handler 异步回调将结果传回到主线程
        // 进行处理（如显示在 Activity 等等），即进行了线程切换

        // 具体是如何做线程切换 ->>分析 2

        @Override

            public void run() {

                if (delegate.isCanceled()) {

                    callback.onFailure(ExecutorCallbackCall.this, new
IOException("Canceled"));

                } else {

                    callback.onResponse(ExecutorCallbackCall.this, response);

                }

            }

        });

    }

}
```

```
    @Override

        public void onFailure(Call<T> call, final Throwable t) {

            callbackExecutor.execute(new Runnable() {

                @Override public void run() {

                    callback.onFailure(ExecutorCallbackCall.this, t);

                }
            });

        });
    }

}

}

<-- 分析 1: delegate.enqueue () 解析 -->

@Override

public void enqueue(final Callback<T> callback) {

    okhttp3.Call call;

    Throwable failure;

    // 步骤 1: 创建 OkHttp 的 Request 对象, 再封装成 OkHttp.call
```

```
// delegate 代理在网络请求前的动作：创建 OkHttp 的 Request 对象，再封装成
OkHttp.call

    synchronized (this) {

        if (executed) throw new IllegalStateException("Already executed.");

        executed = true;

        call = rawCall;

        failure = creationFailure;

        if (call == null && failure == null) {

            try {

                call = rawCall = createRawCall();

                // 创建 OkHttp 的 Request 对象，再封装成 OkHttp.call

                // 方法同发送同步请求，此处不作过多描述

            } catch (Throwable t) {

                failure = creationFailure = t;

            }

        }

    }

// 步骤 2：发送网络请求

    // delegate 是 OkHttpCall 的静态代理
```

```
// delegate 静态代理最终还是调用 Okhttp.enqueue 进行网络请求

call.enqueue(new okhttp3.Callback() {

    @Override

    public void onResponse(okhttp3.Call call, okhttp3.Response rawResponse)
        throws IOException {

        Response<T> response;

        try {

            // 步骤 3：解析返回数据

            response = parseResponse(rawResponse);

        } catch (Throwable e) {

            callFailure(e);

            return;

        }

        callSuccess(response);

    }

    @Override

    public void onFailure(okhttp3.Call call, IOException e) {

        try {
```

```
        callback.onFailure(OkHttpCall.this, e);

    } catch (Throwable t) {

        t.printStackTrace();

    }

}

private void callFailure(Throwable e) {

    try {

        callback.onFailure(OkHttpCall.this, e);

    } catch (Throwable t) {

        t.printStackTrace();

    }

}

private void callSuccess(Response<T> response) {

    try {

        callback.onResponse(OkHttpCall.this, response);

    } catch (Throwable t) {

        t.printStackTrace();

    }

}
```

```
    }

    });

}

// 请回去上面分析 1 的起点

<-- 分析 2: 异步请求后的线程切换-->

// 线程切换是通过一开始创建 Retrofit 对象时 Platform 在检测到运行环境是 Android 时
进行创建的: (之前已分析过)

// 采用适配器模式

static class Android extends Platform {

    // 创建默认的回调执行器工厂

    // 如果不将 RxJava 和 Retrofit 一起使用, 一般都是使用该默认的
    CallAdapter.Factory

    // 后面会对 RxJava 和 Retrofit 一起使用的情况进行分析

    @Override

    CallAdapter.Factory defaultCallAdapterFactory(Executor callbackExecutor) {

        return new ExecutorCallAdapterFactory(callbackExecutor);

    }

}
```

```
@Override

public Executor defaultCallbackExecutor() {

    // 返回一个默认的回调方法执行器

    // 该执行器负责在主线程（UI 线程）中执行回调方法

    return new MainThreadExecutor();

}

// 获取主线程 Handler

static class MainThreadExecutor implements Executor {

    private final Handler handler = new Handler(Looper.getMainLooper());

    @Override

    public void execute(Runnable r) {

        // Retrofit 获取了主线程的 handler

        // 然后在 UI 线程执行网络请求回调后的数据显示等操作。

        handler.post(r);

    }

}
```

```
// 切换线程的流程：  
  
// 1. 回调 ExecutorCallAdapterFactory 生成了一个 ExecutorCallbackCall 对象  
  
// 2. 通过调用 ExecutorCallbackCall.enqueue(CallBack)从而调用 MainThreadExecutor  
的 execute()通过 handler 切换到主线程处理返回结果（如显示在 Activity 等等）  
  
}
```

以上便是整个以 **异步方式**发送网络请求的过程。

5. 总结

`Retrofit` 本质上是一个 `RESTful` 的 `HTTP` 网络请求框架的封装，即通过 大量的设计模式 封装了 `OkHttp`，使得简洁易用。具体过程如下：

1. `Retrofit` 将 `Http` 请求 抽象 成 `Java` 接口
2. 在接口里用 注解 描述和配置 网络请求参数
3. 用动态代理 的方式，动态将网络请求接口的注解 解析 成 `HTTP` 请求
4. 最后执行 `HTTP` 请求

最后贴一张非常详细的 `Retrofit` 源码分析图：

步骤		具体过程	使用的模式				
步骤1	创建Retrofit实例	<p>通过内部类Builder类建立一个Retrofit实例（建造者模式），具体创建过程是配置了：</p> <ul style="list-style-type: none"> 平台类型对象（Platform – Android） 网络请求的url地址（baseUrl） 网络请求工厂（callFactory） – 默认使用OkHttpCall（工厂方法模式） 网络请求适配器工厂的集合（adapterFactories） – 默认是ExecutorCallAdapterFactory* 数据转换器工厂的集合（converterFactories） – 本质上是配置了数据转换器工厂 回调方法执行器（callbackExecutor） – 默认回调方法执行器作用是：切换线程（子线程 – 主线程） 	建造者模式、工厂方法模式				
步骤2	创建网络请求接口的实例 (通过解析注解配置网络请求参数)	<p>Retrofit采用了外观模式统一调用创建网络请求接口实例和网络请求参数配置的方法，具体过程：</p> <ol style="list-style-type: none"> 1. 动态创建网络请求接口的实例（代理模式 – 动态代理# InnovationHandler对象# invoke ()) 2. 创建 serviceMethod 对象（建造者模式 & 单例模式（缓存机制）） 3. 对 serviceMethod 对象进行网络请求参数配置：通过解析网络请求接口方法的参数、返回值和注解类型，从Retrofit对象中获取对应的网络请求的url地址、网络请求执行器、网络请求适配器 & 数据转换器。（策略模式） 4. 对 serviceMethod 对象加入线程切换的操作，便于接收数据后通过Handler从子线程切换到主线程从而对返回数据结果进行处理（装饰模式） 5. 最终创建并返回一个OkHttpCall类型的网络请求对象 	外观模式、代理模式、建造者模式、单例模式、策略模式、装饰模式				
步骤3	发送网络请求	<table border="1"> <thead> <tr> <th>异步方式</th> <th>同步方式</th> </tr> </thead> <tbody> <tr> <td> 1. 通过静态 delegate 代理对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 ServiceMethod 对象创建一个 OkHttp 的 Request 对象，并封装成 OkHttp.call 对象（代理模式） 2. 通过静态 delegate 通过 OkHttp.call 对象发送网络请求（代理模式） </td> <td> 1. 对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 ServiceMethod 对象创建一个 OkHttp 的 Request 对象，并封装成 OkHttp.call 对象 2. 通过 OkHttp.call 对象发送网络请求 </td> </tr> </tbody> </table>	异步方式	同步方式	1. 通过静态 delegate 代理对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 ServiceMethod 对象创建一个 OkHttp 的 Request 对象，并封装成 OkHttp.call 对象（代理模式） 2. 通过静态 delegate 通过 OkHttp.call 对象发送网络请求（代理模式）	1. 对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 ServiceMethod 对象创建一个 OkHttp 的 Request 对象，并封装成 OkHttp.call 对象 2. 通过 OkHttp.call 对象发送网络请求	代理模式、适配器模式、装饰模式
异步方式	同步方式						
1. 通过静态 delegate 代理对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 ServiceMethod 对象创建一个 OkHttp 的 Request 对象，并封装成 OkHttp.call 对象（代理模式） 2. 通过静态 delegate 通过 OkHttp.call 对象发送网络请求（代理模式）	1. 对网络请求接口的方法中的每个参数利用对应 ParameterHandler 进行解析，再根据 ServiceMethod 对象创建一个 OkHttp 的 Request 对象，并封装成 OkHttp.call 对象 2. 通过 OkHttp.call 对象发送网络请求						
步骤4	解析数据	对返回的数据使用之前设置的数据转换器（GsonConverterFactory）解析返回的数据，最终得到一个 Response<T> 对象					
步骤5	切换线程	使用回调执行器进行线程切换（子线程 – 主线程）（适配器模式、装饰模式）					
步骤6	处理结果	在主线程处理返回的数据结果	通过 Response<T> 对象处理返回的数据结果				

6. 深入解析 OkHttp 源码

6.1 OkHttp 3.7 源码分析（一）——整体架构

简介： OkHttp 是一个处理网络请求的开源项目，是 Android 端最火热的轻量级框架，由移动支付 Square 公司贡献用于替代 HttpURLConnection 和 Apache HttpClient。随着 OkHttp 的不断成熟，越来越多的 Android 开发者使用 OkHttp 作为网络框架。

OkHttp3.7 源码分析文章列表如下：

- OkHttp 源码分析——整体架构
- OkHttp 源码分析——拦截器
- OkHttp 源码分析——任务队列
- OkHttp 源码分析——缓存策略
- OkHttp 源码分析——多路复用

OkHttp 是一个处理网络请求的开源项目,是 Android 端最火热的轻量级框架,由移动支付 Square 公司贡献用于替代 HttpURLConnection 和 Apache HttpClient。随着 OkHttp 的不断成熟, 越来越多的 Android 开发者使用 OkHttp 作为网络框架。

之所以可以赢得如此多开发者的喜爱, 主要得益于如下特点:

- 支持 HTTPS/HTTP2/WebSocket (在 OkHttp3.7 中已经剥离对 Spdy 的支持, 转而大力支持 HTTP2)
- 内部维护任务队列线程池, 友好支持并发访问
- 内部维护连接池, 支持多路复用, 减少连接创建开销
- socket 创建支持最佳路由
- 提供拦截器链 (InterceptorChain) , 实现 request 与 response 的分层处理(如透明 GZIP 压缩, logging 等)

为了一探 OkHttp 是如何具备以下特点的, 笔者反复研究 OkHttp 框架源码, 力求通过源码分析向各位解释一二。所以特意准备了几篇博客跟大家一起探讨下 OkHttp 的方方面面, 今天首先从整体架构上分析下 OkHttp。

简单使用

首先来看下 OkHttp 的简单使用, OkHttp 提供了两种调用方式:

- 同步调用
- 异步调用

同步调用

```
@Override public Response execute() throws IOException {  
    synchronized (this) {  
        if (executed) throw new IllegalStateException("Already Executed");  
        executed = true;  
    }  
    try {  
        client.dispatcher().executed(this);  
        Response result = getResponseWithInterceptorChain(false);  
        if (result != null) {  
            return result;  
        }  
        return client.dispatcher().queue.take();  
    } catch (IOException e) {  
        if (executed) {  
            throw e;  
        }  
        synchronized (this) {  
            if (!executed) {  
                throw e;  
            }  
        }  
    }  
}
```

```
    if (result == null) throw new IOException("Canceled");

    return result;

} finally {

    client.dispatcher().finished(this);

}

}
```

首先加锁置标志位，接着使用分配器的 executed 方法将 call 加入到同步队列中，然后调用 getResponseWithInterceptorChain 方法（稍后分析）执行 http 请求，最后调用 finished 方法将 call 从同步队列中删除

异步请求

```
void enqueue(Callback responseCallback, boolean forWebSocket) {

    synchronized (this) {

        if (executed) throw new IllegalStateException("Already Executed");

        executed = true;

    }

    client.dispatcher().enqueue(new AsyncCall(responseCallback, forWebSocket));
}

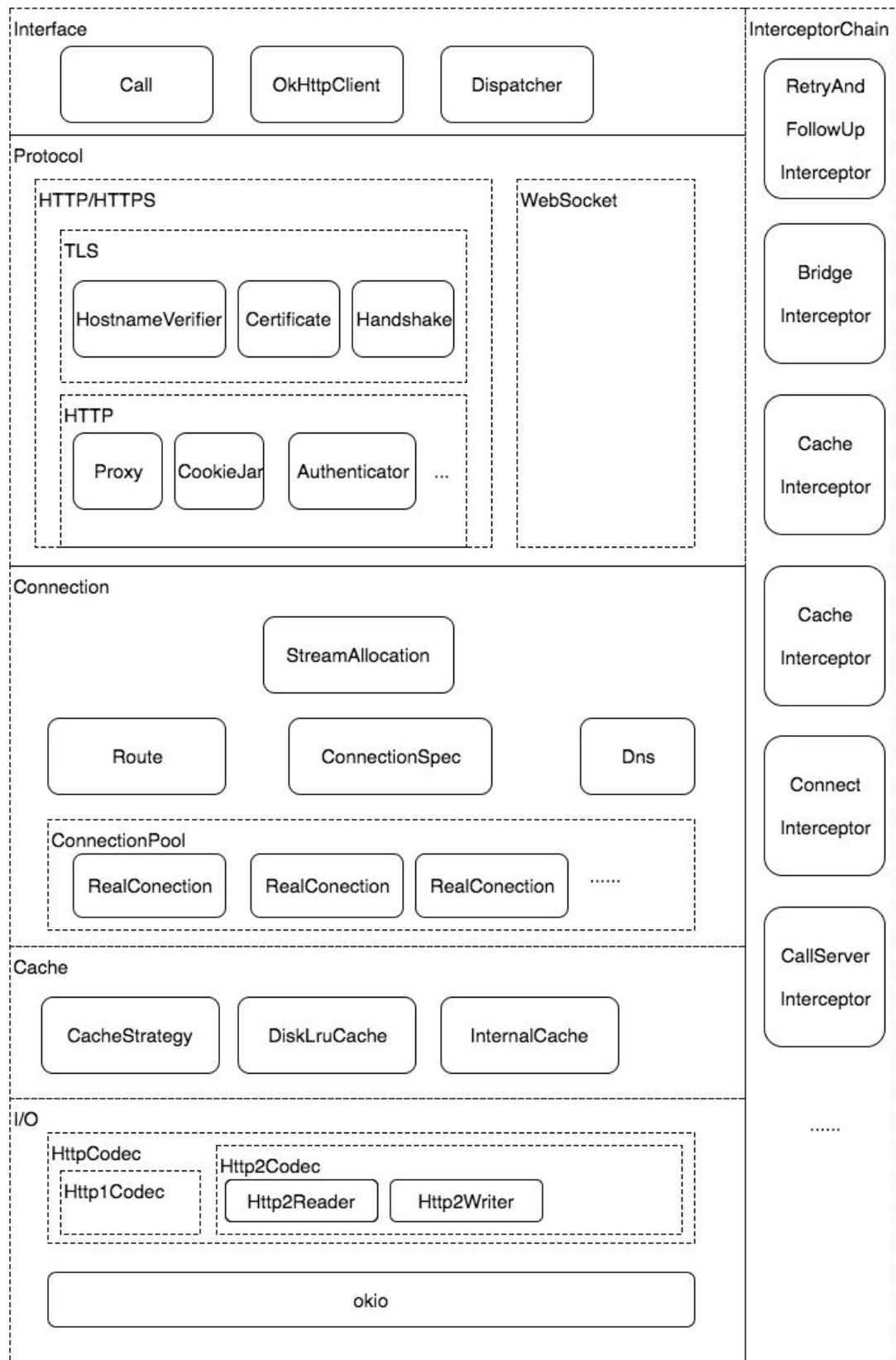
}
```

同样先置标志位，然后将封装的一个执行体放到异步执行队列中。这里面引入了一个新的类 AsyncCall，这个类继承于 NamedRunnable，实现了 Runnable 接口。NamedRunnable 可以给当前的线程设置名字，并且用模板方法将线程的执行体放到了 execute 方法中

拦截器

除了同步调用和异步调用，OkHttp 还提供了一个拦截器的概念。拦截器提供了拦截请求和拦截服务器应答的接口。OkHttp 提供了一个拦截器链的概念，通过将一个个拦截器组合成一个拦截器链，可以达到在不同层面做不同拦截操作的效果，有点 AOP 的意思。具体拦截器的使用可以参考：[Okhttp-wiki 之 Interceptors 拦截器](#)

总体架构



上图是 OkHttp 的总体架构，大致可以分为以下几层：

- Interface——接口层：接受网络访问请求
- Protocol——协议层：处理协议逻辑
- Connection——连接层：管理网络连接，发送新的请求，接收服务器访问
- Cache——缓存层：管理本地缓存
- I/O——I/O 层：实际数据读写实现
- Interceptor——拦截器层：拦截网络访问，插入拦截逻辑

Interface——接口层：

接口层接收用户的网络访问请求（同步请求/异步请求），发起实际的网络访问。OkHttpClient 是 OkHttp 框架的客户端，更确切的说是一个用户面板。用户使用 OkHttp 进行各种设置，发起各种网络请求都是通过 OkHttpClient 完成的。每个 OkHttpClient 内部都维护了属于自己的任务队列，连接池，Cache，拦截器等，所以在使用 OkHttp 作为网络框架时应该全局共享一个 OkHttpClient 实例。

Call 描述一个实际的访问请求，用户的每一个网络请求都是一个 Call 实例。Call 本身只是一个接口，定义了 Call 的接口方法，实际执行过程中，OkHttp 会为每一个请求创建一个 RealCall，每一个 RealCall 内部有一个 AsyncCall：

```
final class AsyncCall extends NamedRunnable {

    private final Callback responseCallback;

    AsyncCall(Callback responseCallback) {
        super("OkHttp %s", redactedUrl());
        this.responseCallback = responseCallback;
    }

    String host() {
        return originalRequest.url().host();
    }
}
```

```
Request request() {  
    return originalRequest;  
}  
  
RealCall get() {  
    return RealCall.this;  
}  
  
@Override protected void execute() {  
    ...  
}  
...  
}
```

AsyncCall 继承的 NamedRunnable 继承自 Runnable 接口：

```
public abstract class NamedRunnable implements Runnable {  
  
    protected final String name;  
  
    public NamedRunnable(String format, Object... args) {  
        this.name = Util.format(format, args);  
    }  
  
    @Override public final void run() {  
        String oldName = Thread.currentThread().getName();  
        Thread.currentThread().setName(name);  
    }  
}
```

```
try {

    execute();

} finally {

    Thread.currentThread().setName(oldName);

}

}

protected abstract void execute();

}
```

所以每一个 Call 就是一个线程，而执行 Call 的过程就是执行其 execute 方法的过程。

Dispatcher 是 OkHttp 的任务队列，其内部维护了一个线程池，当有接收到一个 call 时，Dispatcher 负责在线程池中找到空闲的线程并执行其 execute 方法。这部分将会单独拿一篇博客进行介绍，详细内容可参考本系列接下来的文章。

Protocol——协议层：处理协议逻辑

Protocol 层负责处理协议逻辑，OkHttp 支持 Http1/Http2/WebSocket 协议，并在 3.7 版本中放弃了对 Spdy 协议，鼓励开发者使用 Http/2。

Connection——连接层：管理网络连接，发送新的请求，接收服务器访问

连接层顾名思义就是负责网络连接。在连接层中有一个连接池，统一管理所有的 Socket 连接，当用户新发起一个网络请求时，OkHttp 会首先从连接池中查找是否有符合要求的连接，如果有则直接通过该连接发送网络请求；否则新创建一个网络连接。

RealConnection 描述一个物理 Socket 连接，连接池中维护多个 RealConnection 实例。由于 Http/2 支持多路复用，一个 RealConnection 可以支持多个网络访问请求，所以 OkHttp 又引入了 StreamAllocation 来描述一个实际的网络请求开销（从逻辑上一个 Stream 对应一个 call，但在实际网络请求过程中一个 call 常常涉及到多次请求。如重定向，Authenticate 等场景。所以准确地说，一个 Stream 对应一次请求，而一个 call 对应一组有逻辑关联的 Stream），一个 RealConnection 对应一个或多个 StreamAllocation，所以 StreamAllocation 可以看做是 RealConnection 的计数器，当 RealConnection 的引用计数变为 0，且长时间没有被其他请求重新占用就将被释放。

连接层是 OkHttp 的核心部分，这部分当然也会单独拿一篇博客详细讲解，详细内容可参考本专题相关文章。

Cache——缓存层：管理本地缓存

Cache 层负责维护请求缓存，当用户的网络请求在本地已有符合要求的缓存时，OkHttp 会直接从缓存中返回结果，从而节省网络开销。这部分内容也会单独拿一篇博客进行介绍，详细内容可参考本专题相关文章。

I/O——I/O 层：实际数据读写实现

I/O 层负责实际的数据读写。OkHttp 的另一大有点就是其高效的 I/O 操作，这归因于其高效的 I/O 库 [okio](#)

这部分内容笔者也打算另开一个专题进行介绍。详细内容可以参考本博客相关内容。

Interceptor——拦截器层：拦截网络访问，插入拦截逻辑

拦截器层提供了一个类 AOP 接口，方便用户可以切入到各个层面对网络访问进行拦截并执行相关逻辑。在下一篇博客中，笔者将通过介绍一个实际的网络访问请求实例来介绍拦截器的原理。

6.2 OkHttp 3.7 源码分析（二）——拦截器 &一个实际网络请求的实现

简介：前一篇博客中我们介绍了 OkHttp 的总体架构，接下来我们以一个具体的网络请求来讲述 OkHttp 进行网络访问的具体过程。由于该部分与 OkHttp 的拦截器概念紧密联系在一起，所以将这两部分放在一起进行讲解。

OkHttp3.7 源码分析文章列表如下：

- [OkHttp 源码分析——整体架构](#)
- [OkHttp 源码分析——拦截器](#)
- [OkHttp 源码分析——任务队列](#)
- [OkHttp 源码分析——缓存策略](#)
- [OkHttp 源码分析——多路复用](#)

前一篇博客中我们介绍了 OkHttp 的总体架构，接下来我们以一个具体的网络请求来讲述 OkHttp 进行网络访问的具体过程。由于该部分与 OkHttp 的拦截器概念紧密联系在一起，所以将这两部分放在一起进行讲解。

1. 构造 Demo

首先构造一个简单的异步网络访问 Demo:

```
OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()

    .url("http://publicobject.com/helloworld.txt")

    .build();

client.newCall(request).enqueue(new Callback() {

    @Override

    public void onFailure(Call call, IOException e) {

        Log.d("OkHttp", "Call Failed:" + e.getMessage());

    }

    @Override

    public void onResponse(Call call, Response response) throws IOException {

        Log.d("OkHttp", "Call succeeded:" + response.message());

    }

});
```

2. 发起请求

OkHttpClient.newCall 实际是创建一个 RealCall 实例:

```
/**  
 * Prepares the {@code request} to be executed at some point in the future.  
 */  
  
@Override public Call newCall(Request request) {  
  
    return new RealCall(this, request, false /* for web socket */);  
  
}
```

RealCall.enqueue 实际就是讲一个 RealCall 放入到任务队列中，等待合适的机会执行：

```
@Override public void enqueue(Callback responseCallback) {  
  
    synchronized (this) {  
  
        if (executed) throw new IllegalStateException("Already Executed");  
  
        executed = true;  
  
    }  
  
    captureCallStackTrace();  
  
    client.dispatcher().enqueue(new AsyncCall(responseCallback));  
  
}
```

从代码中可以看到最终 RealCall 被转化成一个 AsyncCall 并被放入到任务队列中，任务队列中的分发逻辑这里先不说，相关实现会放在[OkHttp 源码分析——任务队列]()疑问进行介绍。这里只需要知道 AsyncCall 的 execute 方法最终将会被执行：

```
[RealCall.java]  @Override protected void execute() {  
  
    boolean signalledCallback = false;  
  
    try {  
  
        Response response = getResponseWithInterceptorChain();  
  
        if (retryAndFollowUpInterceptor.isCanceled()) {  
  
            signalledCallback = true;  
        }  
    } catch (IOException e) {  
        onFailure(e);  
    }  
}
```

```

        responseCallback.onFailure(RealCall.this, new IOException("Cancelled"));

    } else {

        signalledCallback = true;

        responseCallback.onResponse(RealCall.this, response);

    }

} catch (IOException e) {

    if (signalledCallback) {

        // Do not signal the callback twice!

        Platform.get().log(INFO, "Callback failure for " + toLoggableString(), e);

    } else {

        responseCallback.onFailure(RealCall.this, e);

    }

} finally {

    client.dispatcher().finished(this);

}

}

```

`execute` 方法的逻辑并不复杂，简单的说就是：

- 调用 `getResponseBodyWithInterceptorChain` 获取服务器返回
 - 通知任务分发器(`client.dispatcher()`)该任务已结束
- `getResponseBodyWithInterceptorChain` 构建了一个拦截器链，通过依次执行该拦截器链中的每一个拦截器最终得到服务器返回。

3. 构建拦截器链

首先来看下 `getResponseBodyWithInterceptorChain` 的实现：

```

[RealCall.java] Response getResponseWithInterceptorChain() throws IOException {
    // Build a full stack of interceptors.

    List<Interceptor> interceptors = new ArrayList<>();
    interceptors.addAll(client.interceptors());
    interceptors.add(retryAndFollowUpInterceptor);
    interceptors.add(new BridgeInterceptor(client.cookieJar()));
    interceptors.add(new CacheInterceptor(client.internalCache()));
    interceptors.add(new ConnectInterceptor(client));
    if (!forWebSocket) {
        interceptors.addAll(client.networkInterceptors());
    }
    interceptors.add(new CallServerInterceptor(forWebSocket));
}

Interceptor.Chain chain = new RealInterceptorChain(
    interceptors, null, null, null, 0, originalRequest);
return chain.proceed(originalRequest);
}

```

其逻辑大致分为两部分：

- 创建一系列拦截器，并将其放入一个拦截器数组中。这部分拦截器即包括用户自定义的拦截器也包括框架内部拦截器
- 创建一个拦截器链 `RealInterceptorChain`,并执行拦截器链的 `proceed` 方法

接下来看下 `RealInterceptorChain` 的实现逻辑：

```

[RealInterceptorChain.java]public final class RealInterceptorChain implements Interceptor.Chain {
    private final List<Interceptor> interceptors;
    private final StreamAllocation streamAllocation;
}

```

```
private final HttpCodec httpCodec;
private final RealConnection connection;
private final int index;
private final Request request;
private int calls;

public RealInterceptorChain(List<Interceptor> interceptors, StreamAllocation streamAllocation,
                            HttpCodec httpCodec, RealConnection connection,
                            int index, Request request) {
    this.interceptors = interceptors;
    this.connection = connection;
    this.streamAllocation = streamAllocation;
    this.httpCodec = httpCodec;
    this.index = index;
    this.request = request;
}

@Override public Connection connection() {
    return connection;
}

public StreamAllocation streamAllocation() {
    return streamAllocation;
}
```

```
public HttpCodec httpStream() {  
    return httpCodec;  
}  
  
@Override public Request request() {  
    return request;  
}  
  
@Override public Response proceed(Request request) throws IOException {  
    return proceed(request, streamAllocation, httpCodec, connection);  
}  
  
public Response proceed(Request request, StreamAllocation streamAllocation,  
HttpCodec httpCodec,  
RealConnection connection) throws IOException {  
  
    .....  
  
    // Call the next interceptor in the chain.  
  
    RealInterceptorChain next = new RealInterceptorChain(  
        interceptors, streamAllocation, httpCodec, connection, index + 1, re  
quest);  
  
    Interceptor interceptor = interceptors.get(index);  
  
    Response response = interceptor.intercept(next);  
  
    .....  
}
```

```
        return response;  
    }  
}
```

在 `proceed` 方法中的核心代码可以看到，`proceed` 实际上也做了两件事：

- 创建下一个拦截链。传入 `index + 1` 使得下一个拦截器链只能从下一个拦截器开始访问
 - 执行索引为 `index` 的 `intercept` 方法，并将下一个拦截器链传入该方法
- 接下来再看下第一个拦截器 `RetryAndFollowUpInterceptor` 的 `intercept` 方法：

[`RetryAndFollowUpInterceptor.java`]

```
public final class RetryAndFollowUpInterceptor implements Interceptor {  
  
    @Override public Response intercept(Chain chain) throws IOException {  
  
        Request request = chain.request();  
  
        StreamAllocation streamAllocation = new StreamAllocation(  
            client.connectionPool(), createAddress(request.url()), callStackTrace);  
  
        int followUpCount = 0;  
  
        Response priorResponse = null;  
  
        while (true) {  
  
            if (canceled) {  
  
                streamAllocation.release();  
  
                throw new IOException("Canceled");  
            }  
  
            Response response = null;
```

```
boolean releaseConnection = true;

try {

    // 执行下一个拦截器链的 proceed 方法

    response = ((RealInterceptorChain) chain).proceed(request, streamAllocation, null, null);

    releaseConnection = false;

} catch (RouteException e) {

    // The attempt to connect via a route failed. The request will not have been sent.

    if (!recover(e.getLastConnectException(), false, request)) {

        throw e.getLastConnectException();

    }

    releaseConnection = false;

    continue;

} catch (IOException e) {

    // An attempt to communicate with a server failed. The request may have been sent.

    boolean requestSendStarted = !(e instanceof ConnectionShutdownException);

    if (!recover(e, requestSendStarted, request)) throw e;

    releaseConnection = false;

    continue;

} finally {

    // We're throwing an unchecked exception. Release any resources.

    if (releaseConnection) {

        streamAllocation.streamFailed(null);

        streamAllocation.release();

    }

}
```

```
// Attach the prior response if it exists. Such responses never have a
body.

....  
  
Request followUp = followUpRequest(response);

closeQuietly(response.body());  
  
...  
  
}  
  
}
```

这段代码最关键的代码是：

```
response = ((RealInterceptorChain) chain).proceed(request, streamAllocation,  
null, null);
```

这行代码就是执行下一个拦截器链的 `proceed` 方法。而我们知道在下一个拦截器链中又会执行下一个拦截器的 `intercept` 方法。所以整个执行链就在拦截器与拦截器链中交替执行，最终完成所有拦截器的操作。这也是 OkHttp 拦截器的链式执行逻辑。而一个拦截器的 `intercept` 方法所执行的逻辑大致分为三部分：

- 在发起请求前对 request 进行处理
 - 调用下一个拦截器，获取 response
 - 对 response 进行处理，返回给上一个拦截器

这就是 OkHttp 拦截器机制的核心逻辑。所以一个网络请求实际上就是一个个拦截器执行其 `intercept` 方法的过程。而这其中除了用户自定义的拦截器外还有几个核心拦截器完成了网络访问的核心逻辑，按照先后顺序依次是：

- RetryAndFollowUpInterceptor
- BridgeInterceptor
- CacheInterceptor
- ConnectIntercetot
- CallServerInterceptor

4 RetryAndFollowUpInterceptor

如上文代码所示，RetryAndFollowUpInterceptor 负责两部分逻辑：

- 在网络请求失败后进行重试
- 当服务器返回当前请求需要进行重定向时直接发起新的请求，并在条件允许情况下复用当前连接

5 BridgeInterceptor

```
public final class BridgeInterceptor implements Interceptor {

    private final CookieJar cookieJar;

    public BridgeInterceptor(CookieJar cookieJar) {
        this.cookieJar = cookieJar;
    }

    @Override public Response intercept(Chain chain) throws IOException {
        Log.e("haha", "BridgeInterceptor.intercept");
        Request userRequest = chain.request();
        Request.Builder requestBuilder = userRequest.newBuilder();

        RequestBody body = userRequest.body();
        if (body != null) {
```

```
MediaType contentType = body.contentType();

if (contentType != null) {

    requestBuilder.header("Content-Type", contentType.toString());

}

long contentLength = body.contentLength();

if (contentLength != -1) {

    requestBuilder.header("Content-Length", Long.toString(contentLength));

    requestBuilder.removeHeader("Transfer-Encoding");

} else {

    requestBuilder.header("Transfer-Encoding", "chunked");

    requestBuilder.removeHeader("Content-Length");

}

}

if (userRequest.header("Host") == null) {

    requestBuilder.header("Host", hostHeader(userRequest.url(), false));

}

if (userRequest.header("Connection") == null) {

    requestBuilder.header("Connection", "Keep-Alive");

}

// If we add an "Accept-Encoding: gzip" header field we're responsible for
// or also decompressing
```

```
// the transfer stream.

boolean transparentGzip = false;

if (userRequest.header("Accept-Encoding") == null && userRequest.header("Range") == null) {

    transparentGzip = true;

    requestBuilder.header("Accept-Encoding", "gzip");

}

List<Cookie> cookies = cookieJar.loadForRequest(userRequest.url());

if (!cookies.isEmpty()) {

    requestBuilder.header("Cookie", cookieHeader(cookies));

}

if (userRequest.header("User-Agent") == null) {

    requestBuilder.header("User-Agent", Version.userAgent());

}

Response networkResponse = chain.proceed(requestBuilder.build());

HttpHeaders.receiveHeaders(cookieJar, userRequest.url(), networkResponse.headers());

Response.Builder responseBuilder = networkResponse.newBuilder()

    .request(userRequest);

if (transparentGzip
```

```

    && "gzip".equalsIgnoreCase(networkResponse.header("Content-Encoding"))
))

    && HttpHeaders.hasBody(networkResponse)) {

        GzipSource responseBody = new GzipSource(networkResponse.body().source
()));

        Headers strippedHeaders = networkResponse.headers().newBuilder()

            .removeAll("Content-Encoding")

            .removeAll("Content-Length")

            .build();

        responseBuilder.headers(strippedHeaders);

        responseBuilder.body(new RealResponseBody(strippedHeaders, Okio.buffer
(responseBody)));

    }

    return responseBuilder.build();
}

}

```

BridgeInterceptor 主要负责以下几部分内容：

- 设置内容长度，内容编码
- 设置 gzip 压缩，并在接收到内容后进行解压。省去了应用层处理数据解压的麻烦
- 添加 cookie
- 设置其他报头，如 User-Agent,Host,Keep-alive 等。其中 Keep-Alive 是实现多路复用的必要步骤

6. CacheInterceptor

[CacheInterceptor.intercept]

```
@Override public Response intercept(Chain chain) throws IOException {  
  
    Log.e("haha", "CacheInterceptor.intercept");  
  
    Response cacheCandidate = cache != null  
  
        ? cache.get(chain.request())  
  
        : null;  
  
  
  
    long now = System.currentTimeMillis();  
  
  
  
    CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(),  
cacheCandidate).get();  
  
    Request networkRequest = strategy.networkRequest;  
  
    Response cacheResponse = strategy.cacheResponse;  
  
  
  
    if (cache != null) {  
  
        cache.trackResponse(strategy);  
  
    }  
  
  
  
    if (cacheCandidate != null && cacheResponse == null) {  
  
        closeQuietly(cacheCandidate.body()); // The cache candidate wasn't applicable. Close it.  
  
    }  
  
  
  
    // If we're forbidden from using the network and the cache is insufficient, fail.  
  
    if (networkRequest == null && cacheResponse == null) {  
  
        return new Response.Builder()  
  
            .request(chain.request())
```

```
.protocol(Protocol.HTTP_1_1)

.code(504)

.message("Unsatisfiable Request (only-if-cached)")

.body(Util.EMPTY_RESPONSE)

.sentRequestAtMillis(-1L)

.receivedResponseAtMillis(System.currentTimeMillis())

.build();

}
```

// If we don't need the network, we're done.

```
if (networkRequest == null) {

    return cacheResponse.newBuilder()

        .cacheResponse(stripBody(cacheResponse))

        .build();

}
```

```
Response networkResponse = null;

try {

    networkResponse = chain.proceed(networkRequest);

} finally {

    // If we're crashing on I/O or otherwise, don't leak the cache body.

    if (networkResponse == null && cacheCandidate != null) {

        closeQuietly(cacheCandidate.body());

    }

}
```

```
// If we have a cache response too, then we're doing a conditional get.

if (cacheResponse != null) {

    if (networkResponse.code() == HTTP_NOT_MODIFIED) {

        Response response = cacheResponse.newBuilder()

            .headers(combine(cacheResponse.headers(), networkResponse.headers()))

            .sentRequestAtMillis(networkResponse.sentRequestAtMillis())

            .receivedResponseAtMillis(networkResponse.receivedResponseAtMillis())

            .cacheResponse(stripBody(cacheResponse))

            .networkResponse(stripBody(networkResponse))

            .build();

        networkResponse.body().close();

    }

    // Update the cache after combining headers but before stripping the

    // Content-Encoding header (as performed by initContentStream()).

    cache.trackConditionalCacheHit();

    cache.update(cacheResponse, response);

    return response;

} else {

    closeQuietly(cacheResponse.body());

}

}

Response response = networkResponse.newBuilder()
```

```

.cacheResponse(stripBody(cacheResponse))

.networkResponse(stripBody(networkResponse))

.build();
```

// If the response has a body, offer it to the cache.

```

if (cache != null) {

    if (HttpHeaders.hasBody(response) && CacheStrategy.isCacheable(response, networkRequest)) {

        // Offer this request to the cache.

        CacheRequest cacheRequest = cache.put(response);

        return cacheWritingResponse(cacheRequest, response);

    }

}

if (HttpMethod.invalidateCache(networkRequest.method())) {

    try {

        cache.remove(networkRequest);

    } catch (IOException ignored) {

        // The cache cannot be written.

    }

}

return response;
}
```

CacheInterceptor 的职责很明确，就是负责 Cache 的管理

- 当网络请求有符合要求的 Cache 时直接返回 Cache
- 当服务器返回内容有改变时更新当前 cache

- 如果当前 cache 失效，删除

7 ConnectInterceptor

```
[ConnectInterceptor.java]public final class ConnectInterceptor implements I
nterceptor {

    public final OkHttpClient client;

    public ConnectInterceptor(OkHttpClient client) {

        this.client = client;

    }

    @Override public Response intercept(Chain chain) throws IOException {

        Log.e("haha", "ConnectInterceptor.intercept");

        RealInterceptorChain realChain = (RealInterceptorChain) chain;

        Request request = realChain.request();

        StreamAllocation streamAllocation = realChain.streamAllocation();

        // We need the network to satisfy this request. Possibly for validating
a conditional GET.

        boolean doExtensiveHealthChecks = !request.method().equals("GET");

        HttpCodec httpCodec = streamAllocation.newStream(client, doExtensiveHeal
thChecks);

        RealConnection connection = streamAllocation.connection();

        return realChain.proceed(request, streamAllocation, httpCodec, connectio
n);
    }
```

```
}
```

ConnectInterceptor 的 intercept 方法只有一行关键代码:

```
RealConnection connection = streamAllocation.connection();
```

即为当前请求找到合适的连接，可能复用已有连接也可能是重新创建的连接，返回的连接由连接池负责决定。

8. CallServerInterceptor

```
[CallServerInterceptor.java]@Override public Response intercept(Chain chain)
throws IOException {

    RealInterceptorChain realChain = (RealInterceptorChain) chain;

    HttpCodec httpCodec = realChain.httpStream();

    StreamAllocation streamAllocation = realChain.streamAllocation();

    RealConnection connection = (RealConnection) realChain.connection();

    Request request = realChain.request();

    long sentRequestMillis = System.currentTimeMillis();

    httpCodec.writeRequestHeaders(request);

    Response.Builder responseBuilder = null;

    .....

    httpCodec.finishRequest();

    if (responseBuilder == null) {

        responseBuilder = httpCodec.readResponseHeaders(false);

    }

}
```

```
Response response = responseBuilder

    .request(request)

    .handshake(streamAllocation.connection().handshake())

    .sentRequestAtMillis(sentRequestMillis)

    .receivedResponseAtMillis(System.currentTimeMillis())

    .build();

int code = response.code();

if (forWebSocket && code == 101) {

    // Connection is upgrading, but we need to ensure interceptors see a no
    n-null response body.

    response = response.newBuilder()

        .body(Util.EMPTY_RESPONSE)

        .build();

} else {

    response = response.newBuilder()

        .body(codec.openResponseBody(response))

        .build();

}

if ("close".equalsIgnoreCase(response.request().header("Connection"))

    || "close".equalsIgnoreCase(response.header("Connection")))) {

    streamAllocation.noNewStreams();

}
```

```
    if ((code == 204 || code == 205) && response.body().contentLength() > 0)
    {

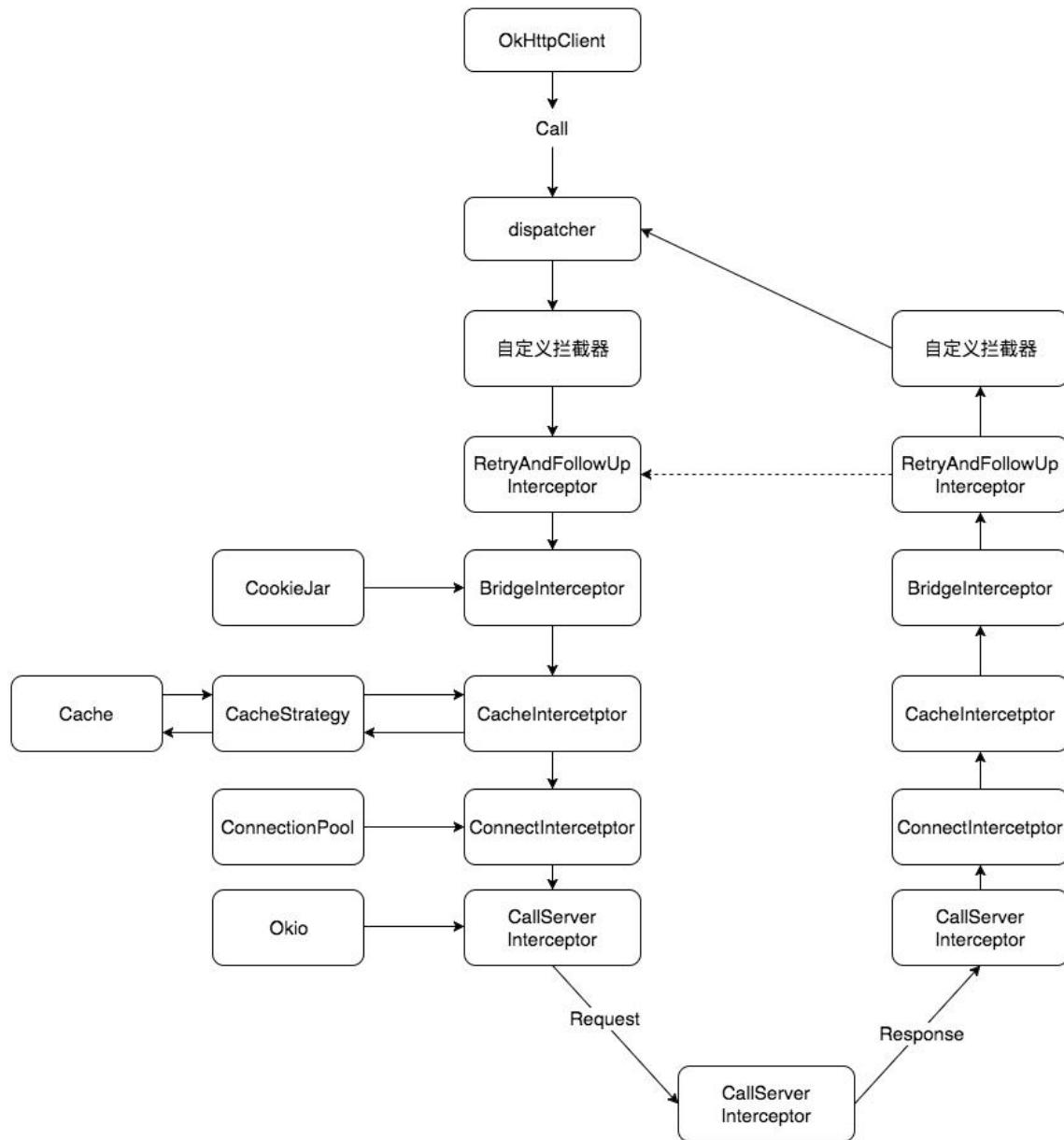
        throw new ProtocolException(
            "HTTP " + code + " had non-zero Content-Length: " + response.body().
contentLength());
    }

    return response;
}
```

CallServerInterceptor 负责向服务器发起真正的访问请求，并在接收到服务器返回后读取响应返回。

8. 整体流程

以上就是整个网络访问的核心步骤，总结起来如下图所示：



6.3 OkHttpClient 3.7 源码分析（三）——任务队

列

简介：OkHttp 的任务队列在内部维护了一个线程池用于执行具体的网络请求。而线程池最大的好处在于通过线程复用减少非核心任务的损耗。

OkHttp3.7 源码分析文章列表如下：

- OkHttp 源码分析——整体架构
- OkHttp 源码分析——拦截器
- OkHttp 源码分析——任务队列
- OkHttp 源码分析——缓存策略
- OkHttp 源码分析——多路复用

前面的博客已经提到过，OkHttp 的一个高效之处在于在内部维护了一个线程池，方便高效地执行异步请求。本篇博客将详细介绍 OkHttp 的任务队列机制。

1. 线程池的优点

OkHttp 的任务队列在内部维护了一个线程池用于执行具体的网络请求。而线程池最大的好处在于通过线程复用减少非核心任务的损耗。

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。但如果对多线程应用不当，会增加对单个任务的处理时间。可以举一个简单的例子：

假设在一台服务器完成一项任务的时间为 T

T1 创建线程的时间 T2 在线程中执行任务的时间，包括线程间同步所需时间 T
3 线程销毁的时间

显然 $T = T_1 + T_2 + T_3$ 。注意这是一个极度简化的假设。

可以看出 T_1, T_3 是多线程本身的带来的开销（在 Java 中，通过映射 pThread，并进一步通过 SystemCall 实现 native 线程），我们渴望减少 T_1, T_3 所用的时间，从而减少 T 的时间。但一些线程的使用者并没有注意到这一点，

所以在程序中频繁的创建或销毁线程，这导致 T1 和 T3 在 T 中占有相当比例。显然这是突出了线程的弱点（T1, T3），而不是优点（并发性）。

线程池技术正是关注如何缩短或调整 T1, T3 时间的技术，从而提高服务器程序性能的。

1. 通过对线程进行缓存，减少了创建销毁的时间损失
2. 通过控制线程数量阀值，减少了当线程过少时带来的 CPU 闲置（比如说长时间卡在 I/O 上了）与线程过多时对 JVM 的内存与线程切换时系统调用的压力
类似的还有 Socket 连接池、DB 连接池、CommonPool(比如 Jedis)等技术。

2. OkHttp 的任务队列

OkHttp 的任务队列主要由两部分组成：

- 任务分发器 dispatcher：负责为任务找到合适的执行线程
- 网络请求任务线程池

```
public final class Dispatcher {  
  
    private int maxRequests = 64;  
  
    private int maxRequestsPerHost = 5;  
  
    private Runnable idleCallback;  
  
    /** Executes calls. Created lazily. */  
  
    private ExecutorService executorService;  
  
    /** Ready async calls in the order they'll be run. */  
  
    private final Deque<AsyncCall> readyAsyncCalls = new ArrayDeque<>();  
  
    /** Running asynchronous calls. Includes canceled calls that haven't finished yet. */  
  
    private final Deque<AsyncCall> runningAsyncCalls = new ArrayDeque<>();
```

```
/** Running synchronous calls. Includes canceled calls that haven't finished yet. */

private final Deque<RealCall> runningSyncCalls = new ArrayDeque<>();

public Dispatcher(ExecutorService executorService) {

    this.executorService = executorService;

}

public Dispatcher() {

}

public synchronized ExecutorService executorService() {

    if (executorService == null) {

        executorService = new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60, TimeUnit.SECONDS,

            new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));

    }

    return executorService;

}

...

}
```

参数说明如下：

- **readyAsyncCalls:** 待执行异步任务队列
- **runningAsyncCalls:** 运行中异步任务队列
- **runningSyncCalls:** 运行中同步任务队列

- executorService: 任务队列线程池:

```
public ThreadPoolExecutor(int corePoolSize,
                         int maximumPoolSize,
                         long keepAliveTime,
                         TimeUnit unit,
                         BlockingQueue<Runnable> workQueue,
                         ThreadFactory threadFactory) {

    this(corePoolSize, maximumPoolSize, keepAliveTime, unit,
         workQueue,
         threadFactory, defaultHandler);
```

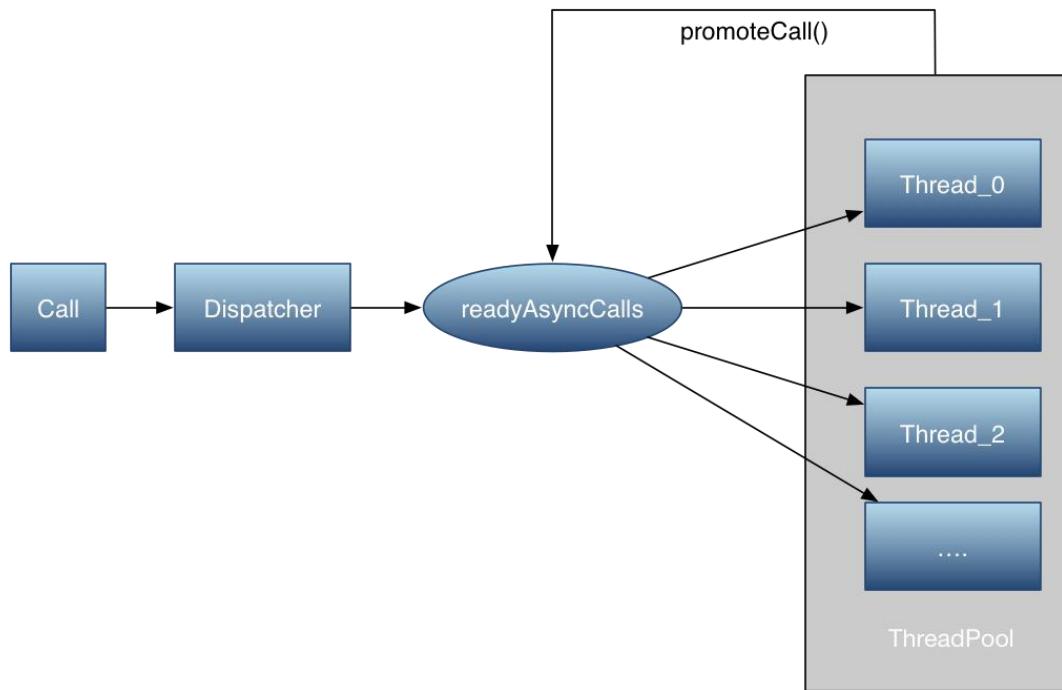
- int corePoolSize: 最小并发线程数, 这里并发同时包括空闲与活动的线程, 如果是 0 的话, 空闲一段时间后所有线程将全部被销毁
- int maximumPoolSize: 最大线程数, 当任务进来时可以扩充的线程最大值, 当大于了这个值就会根据丢弃处理机制来处理
- long keepAliveTime: 当线程数大于 corePoolSize 时, 多余的空闲线程的最大存活时间, 类似于 HTTP 中的 Keep-alive
- TimeUnit unit: 时间单位, 一般用秒
- BlockingQueue workQueue: 工作队列, 先进先出, 可以看出并不像 Picasso 那样设置优先队列
- ThreadFactory threadFactory: 单个线程的工厂, 可以打 Log, 设置 Daemon (即当 JVM 退出时, 线程自动结束) 等

可以看出, 在 Okhttp 中, 构建了一个阀值为 [0, Integer.MAX_VALUE] 的线程池, 它不保留任何最小线程数, 随时创建更多的线程数, 当线程空闲时只能活 60 秒, 它使用了一个不存储元素的阻塞工作队列, 一个叫做 "OkHttp Dispatcher" 的线程工厂。

也就是说, 在实际运行中, 当收到 10 个并发请求时, 线程池会创建十个线程, 当工作完成后, 线程池会在 60s 后相继关闭所有线程。

3. Dispatcher 分发器

dispatcher 分发器类似于 Ngnix 中的反向代理，通过 Dispatcher 将任务分发到合适的空闲线程，实现非阻塞，高可用，高并发连接



1. 同步请求

当我们使用 OkHttp 进行同步请求时，一般构造如下：

```
OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()

    .url("http://publicobject.com/helloworld.txt")

    .build();

Response response = client.newCall(request).execute();
```

接下来看看 `RealCall.execute`

```
@Override public Response execute() throws IOException {

    synchronized (this) {

        if (executed) throw new IllegalStateException("Already Executed");

        executed = true;
```

```
}

captureStackTrace();

try {

    client.dispatcher().executed(this);

    Response result = getResponseWithInterceptorChain();

    if (result == null) throw new IOException("Canceled");

    return result;

} finally {

    client.dispatcher().finished(this);

}

}
```

同步调用的执行逻辑是：

- 将对应任务加入分发器
- 执行任务
- 执行完成后通知 dispatcher 对应任务已完成，对应任务出队

2. 异步请求

异步请求一般构造如下：

```
OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()

    .url("http://publicobject.com/helloworld.txt")

    .build();

client.newCall(request).enqueue(new Callback() {

    @Override

    public void onFailure(Call call, IOException e) {
```

```
        Log.d("OkHttp", "Call Failed:" + e.getMessage());

    }

    @Override

    public void onResponse(Call call, Response response) throws IOException {

        Log.d("OkHttp", "Call succeeded:" + response.message());

    }

});
```

当 HttpClient 的请求入队时，根据代码，我们可以发现实际上是 Dispatcher 进行了入队操作。

```
synchronized void enqueue(AsyncCall call) {

    if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) <
maxRequestsPerHost) {

        //添加正在运行的请求

        runningAsyncCalls.add(call);

        //线程池执行请求

        executorService().execute(call);

    } else {

        //添加到缓存队列排队等待

        readyAsyncCalls.add(call);

    }
}
```

如果满足条件：

- 当前请求数小于最大请求数（64）
- 对单一 host 的请求小于阈值（5）

将该任务插入正在执行任务队列，并执行对应任务。如果不满足则将其放入待执行队列。

接下来看看 `AsyncCall.execute`

```
@Override protected void execute() {  
  
    boolean signalledCallback = false;  
  
    try {  
  
        //执行耗时 IO 任务  
  
        Response response = getResponseWithInterceptorChain(forWebSocket);  
  
        if (canceled) {  
  
            signalledCallback = true;  
  
            //回调，注意这里回调是在线程池中，而不是想当然的主线程回调  
  
            responseCallback.onFailure(RealCall.this, new IOException("Canceled"));  
  
        } else {  
  
            signalledCallback = true;  
  
            //回调，同上  
  
            responseCallback.onResponse(RealCall.this, response);  
  
        }  
  
    } catch (IOException e) {  
  
        if (signalledCallback) {  
  
            // Do not signal the callback twice!  
  
            logger.log(Level.INFO, "Callback failure for " + toLoggableString(),  
e);  
  
        } else {  
  
            responseCallback.onFailure(RealCall.this, e);  
  
        }  
  
    } finally {  
  
    }  
}
```

```

    //最关键的代码

    client.dispatcher().finished(this);

}

}

```

当任务执行完成后，无论成功与否都会调用 dispatcher.finished 方法，通知分发器相关任务已结束：

```

private <T> void finished(Deque<T> calls, T call, boolean promoteCalls) {

    int runningCallsCount;

    Runnable idleCallback;

    synchronized (this) {

        if (!calls.remove(call)) throw new AssertionError("Call wasn't in-flight!");

        if (promoteCalls) promoteCalls();

        runningCallsCount = runningCallsCount();

        idleCallback = this.idleCallback;

    }

    if (runningCallsCount == 0 && idleCallback != null) {

        idleCallback.run();

    }

}

```

- 空闲出多余线程，调用 promoteCalls 调用待执行的任务
- 如果当前整个线程池都空闲下来，执行空闲通知回调线程(idleCallback)

接下来看看 promoteCalls：

```

private void promoteCalls() {

    if (runningAsyncCalls.size() >= maxRequests) return; // Already running max capacity.

```

```

if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); )
{
    AsyncCall call = i.next();

    if (runningCallsForHost(call) < maxRequestsPerHost) {

        i.remove();

        runningAsyncCalls.add(call);

        executorService().execute(call);

    }
}

if (runningAsyncCalls.size() >= maxRequests) return; // Reached max capacity.

}
}

```

promoteCalls 的逻辑也很简单：扫描待执行任务队列，将任务放入正在执行任务队列，并执行该任务。

4. 总结

以上就是整个任务队列的实现细节，总结起来有以下几个特点：

- OkHttp 采用 Dispatcher 技术，类似于 Nginx，与线程池配合实现了高并发，低阻塞的运行
- Okhttp 采用 Deque 作为缓存，按照入队的顺序先进先出
- OkHttp 最出彩的地方就是在 try/finally 中调用了 finished 函数，可以主动控制等待队列的移动，而不是采用锁或者 wait/notify，极大减少了编码复杂性

6.4OkHttp 3.7 源码分析（四）——缓存策

略

简介：合理地利用本地缓存可以有效地减少网络开销，减少响应延迟。HTTP 报头也定义了很多与缓存有关的域来控制缓存。今天就来讲讲 OkHttp 中关于缓存部分的实现细节。

OkHttp3.7 源码分析文章列表如下：

- [OkHttp 源码分析——整体架构](#)
- [OkHttp 源码分析——拦截器](#)
- [OkHttp 源码分析——任务队列](#)
- [OkHttp 源码分析——缓存策略](#)
- [OkHttp 源码分析——多路复用](#)

合理地利用本地缓存可以有效地减少网络开销，减少响应延迟。HTTP 报头也定义了很多与缓存有关的域来控制缓存。今天就来讲讲 OkHttp 中关于缓存部分的实现细节。

1. HTTP 缓存策略

首先来了解下 HTTP 协议中缓存部分的相关域。

1.1 Expires

超时时间，一般用在服务器的 response 报头中用于告知客户端对应资源的过期时间。当客户端需要再次请求相同资源时先比较其过期时间，如果尚未超过过期时间则直接返回缓存结果，如果已经超过则重新请求。

1.2 Cache-Control

相对值，单位时秒，表示当前资源的有效期。Cache-Control 比 Expires 优先级更高：

```
Cache-Control:max-age=31536000,public
```

1.3 条件 GET 请求

1.3.1 Last-Modified-Date

客户端第一次请求时，服务器返回：

```
Last-Modified: Tue, 12 Jan 2016 09:31:27 GMT
```

当客户端二次请求时，可以头部加上如下 header:

```
If-Modified-Since: Tue, 12 Jan 2016 09:31:27 GMT
```

如果当前资源没有被二次修改，服务器返回 304 告知客户端直接复用本地缓存。

1. 3. 2 ETag

ETag 是对资源文件的一种摘要，可以通过 ETag 值来判断文件是否有修改。当客户端第一次请求某资源时，服务器返回：

```
ETag: "5694c7ef-24dc"
```

客户端再次请求时，可在头部加上如下域：

```
If-None-Match: "5694c7ef-24dc"
```

如果文件并未改变，则服务器返回 304 告知客户端可以复用本地缓存。

1. 4 no-cache/no-store

不使用缓存

1. 5 only-if-cached

只使用缓存

2. Cache 源码分析

OkHttp 的缓存工作都是在 `CacheInterceptor` 中完成的, Cache 部分有以下几个关键类：

- `Cache`: Cache 管理器，其内部包含一个 `DiskLruCache` 将 cache 写入文件系统:
 - * <h3>Cache Optimization</h3>
 - *
 - * <p>To measure cache effectiveness, this class tracks three statistics:
 - *

```

*      <li><strong>{@linkplain #requestCount() Request Count:</strong>}<br/>
the number of HTTP

*      requests issued since this cache was created.

*      <li><strong>{@linkplain #networkCount() Network Count:</strong>}<br/>
the number of those

*      requests that required network use.

*      <li><strong>{@linkplain #hitCount() Hit Count:</strong>}<br/>
the number of those requests

*      whose responses were served by the cache.

* </ul>

*

* Sometimes a request will result in a conditional cache hit.
If the cache contains a stale copy of

* the response, the client will issue a conditional {@code GET}.
The server will then send either

* the updated response if it has changed, or a short 'not modified' response if the client's copy

* is still valid. Such responses increment both the network count and hit count.

*

* <p>The best way to improve the cache hit rate is by configuring the web server to return

* cacheable responses. Although this client honors all <a href="http://tools.ietf.org/html/rfc7234">HTTP/1.1 (RFC 7234)</a> cache headers, it doesn't cache

* partial responses.

```

Cache 内部通过 `requestCount`, `networkCount`, `hitCount` 三个统计指标来优化缓存效率

- **CacheStrategy:** 缓存策略。其内部维护一个 `request` 和 `response`, 通过指定 `request` 和 `response` 来描述是通过网络还是缓存获取 `response`, 抑或二者同时使用

```
[CacheStrategy.java]/**
 * Given a request and cached response, this figures out whether
 * to use the network, the cache, or
 * both.
 *
 * <p>Selecting a cache strategy may add conditions to the request (like the "If-Modified-Since"
 * header for conditional GETs) or warnings to the cached response (if the cached data is
 * potentially stale).
 */
public final class CacheStrategy {
    /**
     * The request to send on the network, or null if this call doesn't use the network. */
    public final Request networkRequest;

    /**
     * The cached response to return or validate; or null if this call doesn't use a cache. */
    public final Response cacheResponse;
    .....
}
```

- CacheStrategy\$Factory:缓存策略工厂类根据实际请求返回对应的缓存策略

既然实际的缓存工作都是在 CacheInterceptor 中完成的，那么接下来看下 CacheInterceptor 的核心方法 intercept 方法源码:

```
[CacheInterceptor.java]@Override public Response intercept(Chain chain) throws IOException {
    //首先尝试获取缓存
    Response cacheCandidate = cache != null
        ? cache.get(chain.request())
        : null;
    long now = System.currentTimeMillis();
    //获取缓存策略
    CacheStrategy strategy = new CacheStrategy.Factory(now, chain.request(),
        cacheCandidate).get();
```

```
Request networkRequest = strategy.networkRequest;

Response cacheResponse = strategy.cacheResponse;

//如果有缓存，更新下相关统计指标：命中率

if (cache != null) {

    cache.trackResponse(strategy);

}

//如果当前缓存不符合要求，将其close

if (cacheCandidate != null && cacheResponse == null) {

    closeQuietly(cacheCandidate.body()); // The cache candidate wasn't applicable. Close it.

}

// 如果不能使用网络，同时又没有符合条件的缓存，直接抛504错误

if (networkRequest == null && cacheResponse == null) {

    return new Response.Builder()

        .request(chain.request())

        .protocol(Protocol.HTTP_1_1)

        .code(504)

        .message("Unsatisfiable Request (only-if-cached)")

        .body(Util.EMPTY_RESPONSE)

        .sentRequestAtMillis(-1L)

        .receivedResponseAtMillis(System.currentTimeMillis())

        .build();

}
```

```
// 如果有缓存同时又不使用网络，则直接返回缓存结果

if (networkRequest == null) {

    return cacheResponse.newBuilder()

        .cacheResponse(stripBody(cacheResponse))

        .build();

}

// 尝试通过网络获取回复

Response networkResponse = null;

try {

    networkResponse = chain.proceed(networkRequest);

} finally {

    // If we're crashing on I/O or otherwise, don't leak the cache body.

    if (networkResponse == null && cacheCandidate != null) {

        closeQuietly(cacheCandidate.body());

    }

}

// 如果既有缓存，同时又发起了请求，说明此时是一个 Conditional Get 请求

if (cacheResponse != null) {

    // 如果服务端返回的是 NOT_MODIFIED，缓存有效，将本地缓存和网络响应做合并

    if (networkResponse.code() == HTTP_NOT_MODIFIED) {

        Response response = cacheResponse.newBuilder()

            .headers(combine(cacheResponse.headers(), networkResponse.headers()))
    }
}
```

```
.sentRequestAtMillis(networkResponse.sentRequestAtMillis())

.receivedResponseAtMillis(networkResponse.receivedResponseAtMillis())

.cacheResponse(stripBody(cacheResponse))

.networkResponse(stripBody(networkResponse))

.build();

networkResponse.body().close();

// Update the cache after combining headers but before stripping the
// Content-Encoding header (as performed by initContentStream()).

cache.trackConditionalCacheHit();

cache.update(cacheResponse, response);

return response;

} else {// 如果响应资源有更新，关掉原有缓存

closeQuietly(cacheResponse.body());

}

}

Response response = networkResponse.newBuilder()

.cacheResponse(stripBody(cacheResponse))

.networkResponse(stripBody(networkResponse))

.build();

if (cache != null) {

if (HttpHeaders.hasBody(response) && CacheStrategy.isCacheable(response, networkRequest)) {
```

```

    // 将网络响应写入 cache 中

    CacheRequest cacheRequest = cache.put(response);

    return cacheWritingResponse(cacheRequest, response);
}

if (HttpMethod.invalidateCache(networkRequest.method())) {

    try {

        cache.remove(networkRequest);

    } catch (IOException ignored) {

        // The cache cannot be written.

    }
}

return response;
}

```

核心逻辑都以中文注释的形式在代码中标注出来了，大家看代码即可。通过上面的代码可以看出，几乎所有的动作都是以 CacheStrategy 缓存策略为依据做出的，那么接下来看下缓存策略是如何生成的，相关代码实现在 CacheStrategy\$Factory.get()方法中：

```

[CacheStrategy$Factory]

/**
 * Returns a strategy to satisfy {@code request} using the a cached response {@code response}.
 */
public CacheStrategy get() {

```

```
CacheStrategy candidate = getCandidate();  
  
        if (candidate.networkRequest != null && request.cacheControl().onlyIfCached()) {  
  
            // We're forbidden from using the network and the cache is insufficient.  
  
            return new CacheStrategy(null, null);  
  
        }  
  
        return candidate;  
    }  
  
    /** Returns a strategy to use assuming the request can use the network. */  
    /  
  
    private CacheStrategy getCandidate() {  
  
        // 若本地没有缓存，发起网络请求  
  
        if (cacheResponse == null) {  
  
            return new CacheStrategy(request, null);  
  
        }  
  
        // 如果当前请求是 HTTPS，而缓存没有 TLS 握手，重新发起网络请求  
  
        if (request.isHttps() && cacheResponse.handshake() == null) {  
  
            return new CacheStrategy(request, null);  
  
        }  
  
        // If this response shouldn't have been stored, it should never be used  
        // as a response source. This check should be redundant as long as the
```

```
// persistence store is well-behaved and the rules are constant.

if (!isCacheable(cacheResponse, request)) {

    return new CacheStrategy(request, null);
}

//如果当前的缓存策略是不缓存或者是 conditional get, 发起网络请求

CacheControl requestCaching = request.cacheControl();

if (requestCaching.noCache() || hasConditions(request)) {

    return new CacheStrategy(request, null);
}

//ageMillis:缓存 age

long ageMillis = cacheResponseAge();

//freshMillis: 缓存保鲜时间

long freshMillis = computeFreshnessLifetime();

if (requestCaching.maxAgeSeconds() != -1) {

    freshMillis = Math.min(freshMillis, SECONDS.toMillis(requestCaching.
maxAgeSeconds()));

}

long minFreshMillis = 0;

if (requestCaching.minFreshSeconds() != -1) {

    minFreshMillis = SECONDS.toMillis(requestCaching.minFreshSeconds());
}
```

```
long maxStaleMillis = 0;

CacheControl responseCaching = cacheResponse.cacheControl();

if (!responseCaching.mustRevalidate() && requestCaching.maxStaleSeconds() != -1) {

    maxStaleMillis = SECONDS.toMillis(requestCaching.maxStaleSeconds());

}

//如果 age + min-fresh >= max-age && age + min-fresh < max-age + max-stale, 则虽然缓存过期了,      //但是缓存继续可以使用, 只是在头部添加 110 警告码

if (!responseCaching.noCache() && ageMillis + minFreshMillis < freshMillis + maxStaleMillis) {

    Response.Builder builder = cacheResponse.newBuilder();

    if (ageMillis + minFreshMillis >= freshMillis) {

        builder.addHeader("Warning", "110 HttpURLConnection \\"Response is stale\\"");

    }

    long oneDayMillis = 24 * 60 * 60 * 1000L;

    if (ageMillis > oneDayMillis && isFreshnessLifetimeHeuristic()) {

        builder.addHeader("Warning", "113 HttpURLConnection \\"Heuristic expiration\\"");

    }

    return new CacheStrategy(null, builder.build());

}

// 发起conditional get 请求

String conditionName;
```

```

String conditionValue;

if (etag != null) {

    conditionName = "If-None-Match";

    conditionValue = etag;

} else if (lastModified != null) {

    conditionName = "If-Modified-Since";

    conditionValue = lastModifiedString;

} else if (servedDate != null) {

    conditionName = "If-Modified-Since";

    conditionValue = servedDateString;

} else {

    return new CacheStrategy(request, null); // No condition! Make a regular request.

}

Headers.Builder conditionalRequestHeaders = request.headers().newBuilder();

Internal.instance.addLenient(conditionalRequestHeaders, conditionName,
conditionValue);

Request conditionalRequest = request.newBuilder()

.headers(conditionalRequestHeaders.build())

.build();

return new CacheStrategy(conditionalRequest, cacheResponse);

}

```

可以看到其核心逻辑在 getCandidate 函数中。基本就是 HTTP 缓存协议的实现，核心代码逻辑已通过中文注释说明，大家直接看代码就好。

3. DiskLruCache

Cache 内部通过 DiskLruCache 管理 cache 在文件系统层面的创建，读取，清理等等工作，
接下来看下 DiskLruCache 的主要逻辑：

```
public final class DiskLruCache implements Closeable, Flushable {  
  
    final FileSystem fileSystem;  
  
    final File directory;  
  
    private final File journalFile;  
  
    private final File journalFileTmp;  
  
    private final File journalFileBackup;  
  
    private final int appVersion;  
  
    private long maxSize;  
  
    final int valueCount;  
  
    private long size = 0;  
  
    BufferedSink journalWriter;  
  
    final LinkedHashMap<String, Entry> lruEntries = new LinkedHashMap<>(0, 0.75f, true);  
  
  
    // Must be read and written when synchronized on 'this'.  
  
    boolean initialized;  
  
    boolean closed;  
  
    boolean mostRecentTrimFailed;  
  
    boolean mostRecentRebuildFailed;  
  
    /**
```

```
* To differentiate between old and current snapshots, each entry is given  
a sequence number each  
  
* time an edit is committed. A snapshot is stale if its sequence number is  
not equal to its  
  
* entry's sequence number.  
  
*/  
  
private long nextSequenceNumber = 0;  
  
  
/** Used to run 'cleanupRunnable' for journal rebuilds. */  
  
private final Executor executor;  
  
private final Runnable cleanupRunnable = new Runnable() {  
  
    public void run() {  
  
        .....  
  
    }  
  
};  
  
...  
  
}
```

3.1 journalFile

DiskLruCache 内部日志文件，对 cache 的每一次读写都对应一条日志记录，DiskLruCache 通过分析日志分析和创建 cache。日志文件格式如下：

```
libcore.io.DiskLruCache  
  
1  
100  
2  
  
  
CLEAN 3400330d1dfc7f3f7f4b8d4d803dfcf6 832 21054
```

```
DIRTY 335c4c6028171cfddfbbae1a9c313c52
CLEAN 335c4c6028171cfddfbbae1a9c313c52 3934 2342
REMOVE 335c4c6028171cfddfbbae1a9c313c52
DIRTY 1ab96a171faeeee38496d8b330771a7a
CLEAN 1ab96a171faeeee38496d8b330771a7a 1600 234
READ 335c4c6028171cfddfbbae1a9c313c52
READ 3400330d1dfc7f3f7f4b8d4d803dfcf6
```

前 5 行固定不变，分别为：常量：`libcore.io.DiskLruCache`; `diskCache` 版本; 应用程序版本; `valueCount`(后文介绍)，空行

接下来每一行对应一个 `cache entry` 的一次状态记录，其格式为：[状态 (DIRTY, CLEAN, READ, REMOVE) , key, 状态相关 value(可选)]:

- **DIRTY**: 表明一个 `cache entry` 正在被创建或更新，每一个成功的 DIRTY 记录都应该对应一个 CLEAN 或 REMOVE 操作。如果一个 DIRTY 缺少预期匹配的 CLEAN/REMOVE，则对应 entry 操作失败，需要将其从 `lruEntries` 中删除

- **CLEAN**: 说明 `cache` 已经被成功操作，当前可以被正常读取。每一个 CLEAN 行还需要记录其每一个 value 的长度

- **READ**: 记录一次 `cache` 读取操作
- **REMOVE**: 记录一次 `cache` 清除

日志文件的应用场景主要有四个：

- `DiskCacheLru` 初始化时通过读取日志文件创建 `cache` 容器：`lruEntries`。同时通过日志过滤操作不成功的 `cache` 项。相关逻辑在 `DiskLruCache.readJournalLine`, `DiskLruCache.processJournal`
- 初始化完成后，为避免日志文件不断膨胀，对日志进行重建精简，具体逻辑在 `DiskLruCache.rebuildJournal`
- 每当有 `cache` 操作时将其记录入日志文件中以备下次初始化时使用
- 当冗余日志过多时，通过调用 `cleanUpRunnable` 线程重建日志

3.2 DiskLruCache.Entry

每一个 DiskLruCache.Entry 对应一个 cache 记录:

```
private final class Entry {  
  
    final String key;  
  
    /** Lengths of this entry's files. */  
    final long[] lengths;  
  
    final File[] cleanFiles;  
  
    final File[] dirtyFiles;  
  
    /** True if this entry has ever been published. */  
    boolean readable;  
  
    /** The ongoing edit or null if this entry is not being edited. */  
    Editor currentEditor;  
  
    /** The sequence number of the most recently committed edit to this entry. */  
    long sequenceNumber;  
  
    Entry(String key) {  
        this.key = key;  
  
        lengths = new long[valueCount];  
        cleanFiles = new File[valueCount];  
        dirtyFiles = new File[valueCount];  
    }  
}
```

```
// The names are repetitive so re-use the same builder to avoid allocations.

StringBuilder fileBuilder = new StringBuilder(key).append('.');

int truncateTo = fileBuilder.length();

for (int i = 0; i < valueCount; i++) {

    fileBuilder.append(i);

    cleanFiles[i] = new File(directory, fileBuilder.toString());

    fileBuilder.append(".tmp");

    dirtyFiles[i] = new File(directory, fileBuilder.toString());

    fileBuilder.setLength(truncateTo);

}

}

...

/** *

 * Returns a snapshot of this entry. This opens all streams eagerly to guarantee that we see a

 * single published snapshot. If we opened streams lazily then the streams could come from

 * different edits.

 */

Snapshot snapshot() {

    if (!Thread.holdsLock(DiskLruCache.this)) throw new AssertionError();

    Source[] sources = new Source[valueCount];
```

```
long[] lengths = this.lengths.clone(); // Defensive copy since these can be zeroed out.

try {

    for (int i = 0; i < valueCount; i++) {

        sources[i] = fileSystem.source(cleanFiles[i]);

    }

    return new Snapshot(key, sequenceNumber, sources, lengths);

} catch (FileNotFoundException e) {

    // A file must have been deleted manually!

    for (int i = 0; i < valueCount; i++) {

        if (sources[i] != null) {

            Util.closeQuietly(sources[i]);

        } else {

            break;

        }

    }

    // Since the entry is no longer valid, remove it so the metadata is accurate (i.e. the cache

    // size.)

}

try {

    removeEntry(this);

} catch (IOException ignored) {

}

return null;

}

}
```

```
}
```

一个 Entry 主要由以下几部分构成:

- **key:** 每个 cache 都有一个 key 作为其标识符。当前 cache 的 key 为其对应 URL 的 MD5 字符串
- **cleanFiles/dirtyFiles:** 每一个 Entry 对应多个文件，其对应的文件数由 DiskLruCache.valueCount 指定。当前在 OkHttp 中 valueCount 为 2。即每个 cache 对应 2 个 cleanFiles, 2 个 dirtyFiles。其中第一个 cleanFiles/dirtyFiles 记录 cache 的 meta 数据(如 URL, 创建时间, SSL 握手记录等等)，第二个文件记录 cache 的真正内容。cleanFiles 记录处于稳定状态的 cache 结果，dirtyFiles 记录处于创建或更新状态的 cache
- **currentEditor:** entry 编辑器，对 entry 的所有操作都是通过其编辑器完成。编辑器内部添加了同步锁

3.3 cleanupRunnable

清理线程，用于重建精简日志:

```
private final Runnable cleanupRunnable = new Runnable() {  
  
    public void run() {  
  
        synchronized (DiskLruCache.this) {  
  
            if (!initialized | closed) {  
  
                return; // Nothing to do  
  
            }  
  
            try {  
  
                trimToSize();  
  
            } catch (IOException ignored) {  
  
                mostRecentTrimFailed = true;  
  
            }  
  
            try {
```

```
    if (journalRebuildRequired()) {  
  
        rebuildJournal();  
  
        redundantOpCount = 0;  
    }  
  
} catch (IOException e) {  
  
    mostRecentRebuildFailed = true;  
  
    journalWriter = Okio.buffer(Okio.blackhole());  
}  
  
}  
  
};
```

其触发条件在 journalRebuildRequired()方法中:

```
/**  
  
 * We only rebuild the journal when it will halve the size of the journal a  
nd eliminate at least  
  
 * 2000 ops.  
 */  
  
boolean journalRebuildRequired() {  
  
    final int redundantOpCompactThreshold = 2000;  
  
    return redundantOpCount >= redundantOpCompactThreshold  
  
        && redundantOpCount >= lruEntries.size();  
}
```

当冗余日志超过日志文件本身的一般且总条数超过 2000 时执行

3.4 SnapShot

cache 快照，记录了特定 cache 在某一个特定时刻的内容。每次向 DiskLruCache 请求时返回的都是目标 cache 的一个快照，相关逻辑在 DiskLruCache.get 中：

```
[DiskLruCache.java] /**
 * Returns a snapshot of the entry named {@code key}, or null if it doesn't
exist is not currently
 * readable. If a value is returned, it is moved to the head of the LRU que
ue.
*/
public synchronized Snapshot get(String key) throws IOException {
    initialize();

    checkNotClosed();

    validateKey(key);

    Entry entry = lruEntries.get(key);

    if (entry == null || !entry.readable) return null;

    Snapshot snapshot = entry.snapshot();

    if (snapshot == null) return null;

    redundantOpCount++;

    //日志记录

    journalWriter.writeUtf8(READ).writeByte(' ').writeUtf8(key).writeByte(
    '\n');

    if (journalRebuildRequired()) {

        executor.execute(cleanupRunnable);

    }
}
```

```
return snapshot;
```

3.5 lruEntries

管理 cache entry 的容器，其数据结构是 LinkedHashMap。通过 LinkedHashMap 本身的实现逻辑达到 cache 的 LRU 替换

3.6 FileSystem

使用 Okio 对 File 的封装，简化了 I/O 操作。

3.7 DiskLruCache.edit

DiskLruCache 可以看成是 Cache 在文件系统层的具体实现，所以其基本操作接口存在一一对应的关系：

- Cache.get() → DiskLruCache.get()
 - Cache.put() → DiskLruCache.edit() //cache 插入
 - Cache.remove() → DiskLruCache.remove()
 - Cache.update() → DiskLruCache.edit() //cache 更新

其中 `get` 操作在 3.4 已经介绍了，`remove` 操作较为简单，`put` 和 `update` 大致逻辑相似，因为篇幅限制，这里仅介绍 `Cache.put` 操作的逻辑，其他的操作大家看代码就好：

```
[okhttp3.Cache.java]

CacheRequest put(Response response) {
    String requestMethod = response.request().method();
    if (HttpMethod.invalidateCache(response.request().method())) {
        try {
            remove(response.request());
        } catch (IOException ignored) {
            // The cache cannot be written.
        }
    }
}
```

```
        return null;

    }

    if (!requestMethod.equals("GET")) {

        // Don't cache non-GET responses. We're technically allowed to cache
        // HEAD requests and some POST requests, but the complexity of doing
        // so is high and the benefit is low.

        return null;
    }

    if (HttpHeaders.hasVaryAll(response)) {

        return null;
    }

Entry entry = new Entry(response);

DiskLruCache.Editor editor = null;

try {

    editor = cache.edit(key(response.request().url()));

    if (editor == null) {

        return null;
    }

    entry.writeTo(editor);

    return new CacheRequestImpl(editor);
} catch (IOException e) {

    abortQuietly(editor);

    return null;
}
```

```
}
```

```
}
```

可以看到核心逻辑在 `editor = cache.edit(key(response.request().url()));`, 相关代码在 `DiskLruCache.edit`:

```
[okhttp3.internal.cache.DiskLruCache.java] synchronized Editor edit(String key, long expectedSequenceNumber) throws IOException {  
    initialize();  
  
    checkNotClosed();  
  
    validateKey(key);  
  
    Entry entry = lruEntries.get(key);  
  
    if (expectedSequenceNumber != ANY_SEQUENCE_NUMBER && (entry == null  
        || entry.sequenceNumber != expectedSequenceNumber)) {  
  
        return null; // Snapshot is stale.  
    }  
  
    if (entry != null && entry.currentEditor != null) {  
  
        return null; // 当前 cache entry 正在被其他对象操作  
    }  
  
    if (mostRecentTrimFailed || mostRecentRebuildFailed) {  
  
        // The OS has become our enemy! If the trim job failed, it means we are  
        // storing more data than  
  
        // requested by the user. Do not allow edits so we do not go over that  
        // limit any further. If  
  
        // the journal rebuild failed, the journal writer will not be active, meaning  
        // we will not be  
  
        // able to record the edit, causing file leaks. In both cases, we want  
        // to retry the clean up  
  
        // so we can get out of this state!
```

```

        executor.execute(cleanupRunnable);

        return null;
    }

    // 日志接入 DIRTY 记录

    journalWriter.writeUtf8(DIRTY).writeByte(' ').writeUtf8(key).writeByte
    ('\n');

    journalWriter.flush();

    if (hasJournalErrors) {

        return null; // Don't edit; the journal can't be written.

    }

    if (entry == null) {

        entry = new Entry(key);

        lruEntries.put(key, entry);

    }

    Editor editor = new Editor(entry);

    entry.currentEditor = editor;

    return editor;

}

```

edit 方法返回对应 CacheEntry 的 editor 编辑器。接下来再来看下 Cache.put()方法的 entry.writeTo(editor); 其相关逻辑：

```
[okhttp3.internal.cache.DiskLruCache.java]    public void writeTo(DiskLruCache.Editor editor) throws IOException {

    BufferedSink sink = Okio.buffer(editor.newSink(ENTRY_METADATA));

```

```
sink.writeUtf8(url)

    .writeByte('\n');

sink.writeUtf8(requestMethod)

    .writeByte('\n');

sink.writeDecimalLong(varyHeaders.size())

    .writeByte('\n');

for (int i = 0, size = varyHeaders.size(); i < size; i++) {

    sink.writeUtf8(varyHeaders.name(i))

        .writeUtf8(": ")

        .writeUtf8(varyHeaders.value(i))

        .writeByte('\n');

}

sink.writeUtf8(new StatusLine(protocol, code, message).toString())

    .writeByte('\n');

sink.writeDecimalLong(responseHeaders.size() + 2)

    .writeByte('\n');

for (int i = 0, size = responseHeaders.size(); i < size; i++) {

    sink.writeUtf8(responseHeaders.name(i))

        .writeUtf8(": ")

        .writeUtf8(responseHeaders.value(i))

        .writeByte('\n');

}

sink.writeUtf8(SENT_MILLIS)
```

```

        .writeUtf8(": ")

        .writeDecimalLong(sentRequestMillis)

        .writeByte('\n');

sink.writeUtf8(RECEIVED_MILLIS)

        .writeUtf8(": ")

        .writeDecimalLong(receivedResponseMillis)

        .writeByte('\n');

}

if (isHttps()) {

    sink.writeByte('\n');

    sink.writeUtf8(handshake.cipherSuite().javaName())

        .writeByte('\n');

    writeCertList(sink, handshake.peerCertificates());

    writeCertList(sink, handshake.localCertificates());

    // The handshake's TLS version is null on HttpsURLConnection and on older cached responses.

    if (handshake.tlsVersion() != null) {

        sink.writeUtf8(handshake.tlsVersion().javaName())

            .writeByte('\n');

    }

}

sink.close();

}

```

其主要逻辑就是将对应请求的 meta 数据写入对应 CacheEntry 的索引为 ENTRY_METADATA_A (0) 的 dirtyfile 中。

最后再来看 Cache.put()方法的 return new CacheRequestImpl(editor);:

```
[okhttp3.Cache$CacheRequestImpl]private final class CacheRequestImpl implements CacheRequest {

    private final DiskLruCache.Editor editor;

    private Sink cacheOut;

    private Sink body;

    boolean done;

    public CacheRequestImpl(final DiskLruCache.Editor editor) {

        this.editor = editor;

        this.cacheOut = editor.newSink(ENTRY_BODY);

        this.body = new ForwardingSink(cacheOut) {

            @Override public void close() throws IOException {

                synchronized (Cache.this) {

                    if (done) {

                        return;

                    }

                    done = true;

                    writeSuccessCount++;

                }

                super.close();

                editor.commit();

            }

        };

    }

    @Override public void abort() {
```

```

    synchronized (Cache.this) {

        if (done) {

            return;

        }

        done = true;

        writeAbortCount++;

    }

    Util.closeQuietly(cacheOut);

    try {

        editor.abort();

    } catch (IOException ignored) {

    }

}

@Override public Sink body() {

    return body;

}

}

```

其中 close,abort 方法会调用 editor.abort 和 editor.commit 来更新日志, editor.commit 还会将 dirtyFile 重置为 cleanFile 作为稳定可用的缓存, 相关逻辑在 okhttp3.internal.cache.DiskLruCache\$Editor.completeEdit 中:

```

[okhttp3.internal.cache.DiskLruCache$Editor.completeEdit] synchronized void
completeEdit(Editor editor, boolean success) throws IOException {

    Entry entry = editor.entry;

    if (entry.currentEditor != editor) {

        throw new IllegalStateException();
    }
}

```

```
}

// If this edit is creating the entry for the first time, every index must have a value.

if (success && !entry.readable) {

    for (int i = 0; i < valueCount; i++) {

        if (!editor.written[i]) {

            editor.abort();

            throw new IllegalStateException("Newly created entry didn't create value for index " + i);

        }

        if (!fileSystem.exists(entry.dirtyFiles[i])) {

            editor.abort();

            return;

        }

    }

}

for (int i = 0; i < valueCount; i++) {

    File dirty = entry.dirtyFiles[i];

    if (success) {

        if (fileSystem.exists(dirty)) {

            File clean = entry.cleanFiles[i];

            fileSystem.rename(dirty, clean); // 将dirtyfile 置为cleanfile

            long oldLength = entry.lengths[i];

            long newLength = fileSystem.size(clean);

        }

    }

}
```

```
        entry.lengths[i] = newLength;

        size = size - oldLength + newLength;

    }

} else {

    fileSystem.delete(dirty); //若失败则删除dirtyfile

}

}

redundantOpCount++;

entry.currentEditor = null;

//更新日志

if (entry.readable | success) {

    entry.readable = true;

    journalWriter.writeUtf8(CLEAN).writeByte(' ');

    journalWriter.writeUtf8(entry.key);

    entry.writeLengths(journalWriter);

    journalWriter.writeByte('\n');

    if (success) {

        entry.sequenceNumber = nextSequenceNumber++;

    }

} else {

    lruEntries.remove(entry.key);

    journalWriter.writeUtf8(REMOVE).writeByte(' ');

    journalWriter.writeUtf8(entry.key);

    journalWriter.writeByte('\n');
```

```
}

journalWriter.flush();

if (size > maxSize || journalRebuildRequired()) {

    executor.execute(cleanupRunnable);

}

}
```

CacheRequestImpl 实现 CacheRequest 接口，向外部类(主要是 CacheInterceptor)透出，外部对象通过 CacheRequestImpl 更新或写入缓存数据。

3.8 总结

总结起来 DiskLruCache 主要有以下几个特点：

- 通过 LinkedHashMap 实现 LRU 替换
- 通过本地维护 Cache 操作日志保证 Cache 原子性与可用性，同时为防止日志过分膨胀定时执行日志精简
- 每一个 Cache 项对应两个状态副本：DIRTY,CLEAN。CLEAN 表示当前可用状态 Cache，外部访问到的 cache 快照均为 CLEAN 状态；DIRTY 为更新态 Cache。由于更新和创建都只操作 DIRTY 状态副本，实现了 Cache 的读写分离
- 每一个 Cache 项有四个文件，两个状态（DIRTY,CLEAN），每个状态对应两个文件：一个文件存储 Cache meta 数据，一个文件存储 Cache 内容数据

6.5OkHttp 3.7 源码分析（五）——连接池

简介：接下来讲下 OkHttp 的连接池管理，这也是 OkHttp 的核心部分。通过维护连接池，最大限度重用现有连接，减少网络连接的创建开销，以此提升网络请求效率。

OkHttp3.7 源码分析文章列表如下：

- [OkHttp 源码分析——整体架构](#)
- [OkHttp 源码分析——拦截器](#)
- [OkHttp 源码分析——任务队列](#)

- OkHttp 源码分析——缓存策略
- OkHttp 源码分析——多路复用



接下来讲下 OkHttp 的连接池管理，这也是 OkHttp 的核心部分。通过维护连接池，最大限度重用现有连接，减少网络连接的创建开销，以此提升网络请求效率。

1. 背景

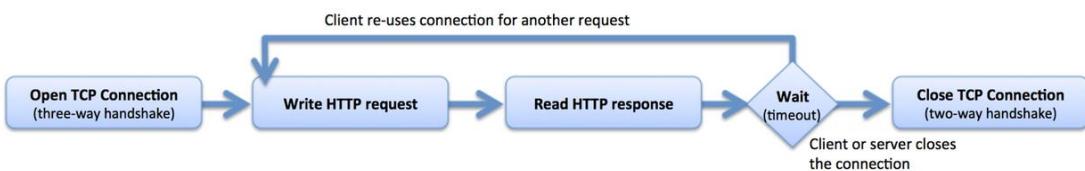
1.1 keep-alive 机制

在 HTTP1.0 中 HTTP 的请求流程如下：



这种方法的好处是简单，各个请求互不干扰。但在复杂的网络请求场景下这种方式几乎不可用。例如：浏览器加载一个 HTML 网页，HTML 中可能需要加载数十个资源，典型场景下这些资源中大部分来自同一个站点。按照 HTTP1.0 的做法，这需要建立数十个 TCP 连接，每个连接负责一个资源请求。创建一个 TCP 连接需要 3 次握手，而释放连接则需要 2 次或 4 次握手。重复的创建和释放连接极大地影响了网络效率，同时也增加了系统开销。

为了有效地解决这一问题，HTTP/1.1 提出了 Keep-Alive 机制：当一个 HTTP 请求的数据传输结束后，TCP 连接不立即释放，如果此时有新的 HTTP 请求，且其请求的 Host 通上次请求相同，则可以直接复用为释放的 TCP 连接，从而省去了 TCP 的释放和再次创建的开销，减少了网络延时：



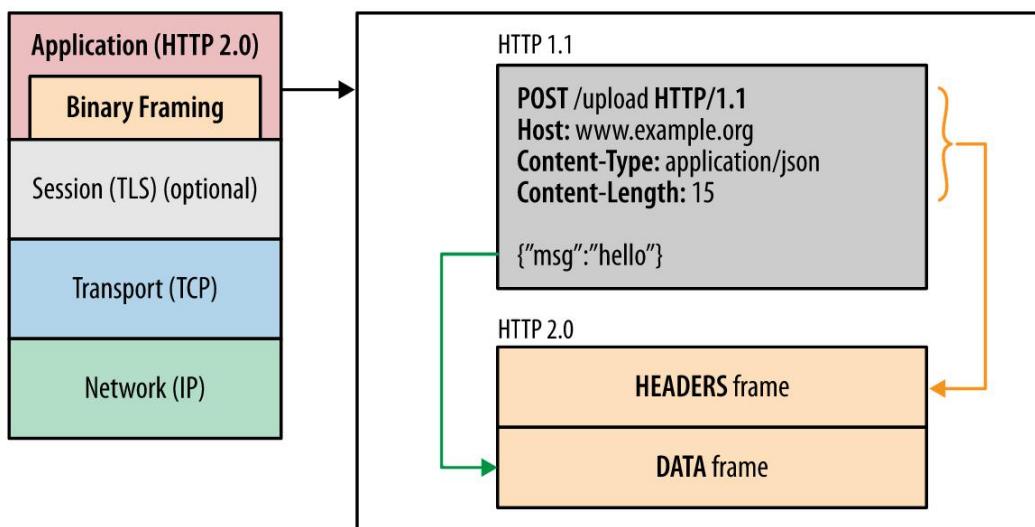
在现代浏览器中，一般同时开启 6~8 个 keepalive connections 的 socket 连接，并保持一定的链路生命，当不需要时再关闭；而在服务器中，一般是由软件根据负载情况(比如 FD 最大值、Socket 内存、超时时间、栈内存、栈数量等)决定是否主动关闭。

1.2 HTTP/2

在 HTTP/1.x 中，如果客户端想发起多个并行请求必须建立多个 TCP 连接，这无疑增大了网络开销。另外 HTTP/1.x 不会压缩请求和响应报头，导致了不必要的网络流量；HTTP/1.x 不支持资源优先级导致底层 TCP 连接利用率低下。而这些问题都是 HTTP/2 要着力解决的。简单来说 HTTP/2 主要解决了以下问题：

- 报头压缩：HTTP/2 使用 HPACK 压缩格式压缩请求和响应报头数据，减少不必要的流量开销
- 请求与响应复用：HTTP/2 通过引入新的二进制分帧层实现了完整的请求和响应复用，客户端和服务器可以将 HTTP 消息分解为互不依赖的帧，然后交错发送，最后再在另一端将其重新组装
- 指定数据流优先级：将 HTTP 消息分解为很多独立的帧之后，我们就可以复用多个数据流中的帧，客户端和服务器交错发送和传输这些帧的顺序就成为关键的性能决定因素。为了做到这一点，HTTP/2 标准允许每个数据流都有一个关联的权重和依赖关系
- 流控制：HTTP/2 提供了一组简单的构建块，这些构建块允许客户端和服务器实现其自己的数据流和连接级流控制

HTTP/2 所有性能增强的核心在于新的二进制分帧层，它定义了如何封装 HTTP 消息并在客户端与服务器之间进行传输：



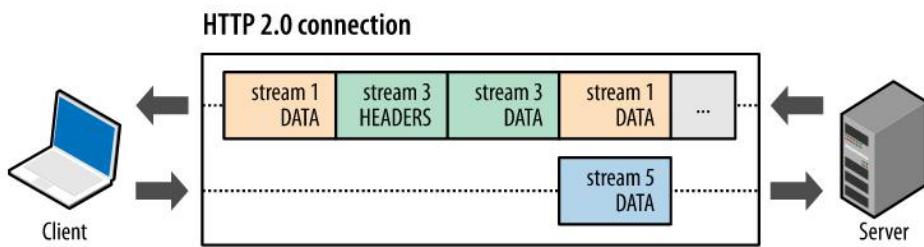
同时 HTTP/2 引入了三个新的概念：

- 数据流：基于 TCP 连接之上的逻辑双向字节流，对应一个请求及其响应。客户端每发起一个请求就建立一个数据流，后续该请求及其响应的所有数据都通过该数据流传输
- 消息：一个请求或响应对应的一系列数据帧

- 帧：HTTP/2 的最小数据切片单位

上述概念之间的逻辑关系：

- 所有通信都在一个 TCP 连接上完成，此连接可以承载任意数量的双向数据流
- 每个数据流都有一个唯一的标识符和可选的优先级信息，用于承载双向消息
- 每条消息都是一条逻辑 HTTP 消息（例如请求或响应），包含一个或多个帧
- 帧是最小的通信单位，承载着特定类型的数据，例如 HTTP 标头、消息负载，等等。
- 来自不同数据流的帧可以交错发送，然后再根据每个帧头的数据流标识符重新组装
- 每个 HTTP 消息被分解为多个独立的帧后可以交错发送，从而在宏观上实现了多个请求或响应并行传输的效果。这类似于多进程环境下的时间分片机制



2. 连接池的使用与分析

无论是 HTTP/1.1 的 Keep-Alive 机制还是 HTTP/2 的多路复用机制，在实现上都需要引入连接池来维护网络连接。接下来看下 OkHttp 中的连接池实现。

OkHttp 内部通过 ConnectionPool 来管理连接池，首先来看下 ConnectionPool 的主要成员：

```
public final class ConnectionPool {
    private static final Executor executor = new ThreadPoolExecutor(0 /* corePoolSize */,
            Integer.MAX_VALUE /* maximumPoolSize */, 60L /* keepAliveTime */, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp ConnectionPool", true));

    /**
     * The maximum number of idle connections for each address.
     */
    private final int maxIdleConnections;
```

```
private final long keepAliveDurationNs;

private final Runnable cleanupRunnable = new Runnable() {

    @Override public void run() {

        .....

    }

};

private final Deque<RealConnection> connections = new ArrayDeque<>();

final RouteDatabase routeDatabase = new RouteDatabase();

boolean cleanupRunning;

.....



/**



 * 返回符合要求的可重用连接，如果没有返回NULL

 */

RealConnection get(Address address, StreamAllocation streamAllocation, Rou
te route) {

    .....

}

/*



 * 去除重复连接。主要针对多路复用场景下一个address 只需要一个连接

 */

Socket deduplicate(Address address, StreamAllocation streamAllocation) {

    .....

}
```

```
/*
 * 将连接加入连接池
 */
void put(RealConnection connection) {
    .....
}

/*
 * 当有连接空闲时唤起 cleanup 线程清洗连接池
 */
boolean connectionBecameIdle(RealConnection connection) {
    .....
}

/**
 * 扫描连接池，清除空闲连接
 */
long cleanup(long now) {
    .....
}

/*
 * 标记泄露连接
*/

```

```
private int pruneAndGetAllocationCount(RealConnection connection, long now)
{
    .....
}
```

相关概念：

- `Call`: 对 Http 请求的封装
- `Connection/RealConnection`: 物理连接的封装，其内部有 `List<WeakReference<StreamAllocation>>` 的引用计数
- `StreamAllocation`: okhttp 中引入了 StreamAllocation 负责管理一个连接上的流，同时在 connection 中也通过一个 StreamAllocation 的引用的列表来管理一个连接的流，从而使得连接与流之间解耦。关于 StreamAllocation 的定义可以看下这篇文章：[okhttp 源码学习笔记（二）-- 连接与连接管理](#)
- `connections`: Deque 双端队列，用于维护连接的容器
- `routeDatabase`: 用来记录连接失败的 Route 的黑名单，当连接失败的时候就会把失败的线路加进去

2.1 实例化

首先来看下 ConnectionPool 的实例化过程，一个 OkHttpClient 只包含一个 ConnectionPool，其实例化过程也在 OkHttpClient 的实例化过程中实现，值得一提的是 ConnectionPool 各个方法的调用并没有直接对外暴露，而是通过 OkHttpClient 的 Internal 接口统一对外暴露：

```
public class OkHttpClient implements Cloneable, Call.Factory, WebSocket.Factory {
    static {
        Internal.instance = new Internal() {
            @Override public void addLenient(Headers.Builder builder, String line) {
                builder.addLenient(line);
            }
        };
    }
}
```

```
    @Override public void addLenient(Headers.Builder builder, String name,
String value) {

    builder.addLenient(name, value);

}

    @Override public void setCache(Builder builder, InternalCache internal
Cache) {

    builder.setInternalCache(internalCache);

}

    @Override public boolean connectionBecameIdle(
    ConnectionPool pool, RealConnection connection) {

    return pool.connectionBecameIdle(connection);

}

    @Override public RealConnection get(ConnectionPool pool, Address addre
ss,
    StreamAllocation streamAllocation, Route route) {

    return pool.get(address, streamAllocation, route);

}

    @Override public boolean equalsNonHost(Address a, Address b) {

    return a.equalsNonHost(b);

}

    @Override public Socket deduplicate(
```

```
        ConnectionPool pool, Address address, StreamAllocation streamAllocation) {

    return pool.deduplicate(address, streamAllocation);

}

@Override public void put(ConnectionPool pool, RealConnection connection) {

    pool.put(connection);

}

@Override public RouteDatabase routeDatabase(ConnectionPool connectionPool) {

    return connectionPool.routeDatabase;

}

@Override public int code(Response.Builder responseBuilder) {

    return responseBuilder.code;

}

@Override

public void apply(ConnectionSpec tlsConfiguration, SSLSocket sslSocket,
boolean isFallback) {

    tlsConfiguration.apply(sslSocket, isFallback);

}

@Override public HttpUrl getHttpUrlChecked(String url)

throws MalformedURLException, UnknownHostException {
```

```
        return HttpUrl.getChecked(url);

    }

    @Override public StreamAllocation streamAllocation(Call call) {
        return ((RealCall) call).streamAllocation();
    }

    @Override public Call newWebSocketCall(OkHttpClient client, Request originalRequest) {
        return new RealCall(client, originalRequest, true);
    }

};

.....
}
```

这样做的原因是：

```
Escalate internal APIs in {@code okhttp3} so they can be used from OkHttp's implementation  
packages. The only implementation of this interface is in {@link OkHttpClient}.
```

Internal 的唯一实现在 OkHttpClient 中，OkHttpClient 通过这种方式暴露其 API 给外部类使用。

2.2 连接池维护

ConnectionPool 内部通过一个双端队列(dequeue)来维护当前所有连接，主要涉及到的操作包括：

- put: 放入新连接
- get: 从连接池中获取连接
- evictAll: 关闭所有连接

- connectionBecameIdle: 连接变空闲后调用清理线程
- deduplicate: 清除重复的多路复用线程

2. 2. 1 StreamAllocation.findConnection

get 是 ConnectionPool 中最为重要的方法, StreamAllocation 在其 findConnection 方法内部通过调用 get 方法为其找到 stream 找到合适的连接, 如果没有则新建一个连接。首先来看下 findConnection 的逻辑:

```
private RealConnection findConnection(int connectTimeout, int readTimeout, i
nt writeTimeout,
                                      boolean connectionRetryEnabled) throws
IOException {

    Route selectedRoute;

    synchronized (connectionPool) {

        if (released) throw new IllegalStateException("released");

        if (codec != null) throw new IllegalStateException("codec != null");

        if (canceled) throw new IOException("Canceled");

        // 一个StreamAllocation 刻画的是一个Call 的数据流动, 一个Call 可能存在多次
        // 请求(重定向, Authenticate 等), 所以当发生类似重定向等事件时优先使用原有的连接

        RealConnection allocatedConnection = this.connection;

        if (allocatedConnection != null && !allocatedConnection.noNewStreams)
        {

            return allocatedConnection;

        }

        // 尝试从连接池中找到可复用的连接

        Internal.instance.get(connectionPool, address, this, null);

        if (connection != null) {
```

```
        return connection;

    }

    selectedRoute = route;

}

// 获取路由配置，所谓路由其实就是代理，ip 地址等参数的一个组合

if (selectedRoute == null) {

    selectedRoute = routeSelector.next();

}

RealConnection result;

synchronized (connectionPool) {

    if (canceled) throw new IOException("Canceled");

    //拿到路由后可以尝试重新从连接池中获取连接，这里主要针对http2 协议下清除域名
    //碎片机制

    Internal.instance.get(connectionPool, address, this, selectedRoute);

    if (connection != null) return connection;

}

//新建连接

route = selectedRoute;

refusedStreamCount = 0;

result = new RealConnection(connectionPool, selectedRoute);

//修改 result 连接 stream 计数，方便 connection 标记清理

acquire(result);
```

```

    }

    // Do TCP + TLS handshakes. This is a blocking operation.

    result.connect(connectTimeout, readTimeout, writeTimeout, connectionRetr
yEnabled);

    routeDatabase().connected(result.route());

    Socket socket = null;

    synchronized (connectionPool) {

        // 将新建的连接放入到连接池中

        Internal.instance.put(connectionPool, result);

        // 如果同时存在多个连向同一个地址的多路复用连接，则关闭多余连接，只保留一个

        if (result.isMultiplexed()) {

            socket = Internal.instance.deduplicate(connectionPool, address, thi
s);

            result = connection;

        }

    }

    closeQuietly(socket);

}

return result;
}

```

其主要逻辑大致分为以下几个步骤：

- 查看当前 streamAllocation 是否有之前已经分配过的连接，有则直接使用
- 从连接池中查找可复用的连接，有则返回该连接

- 配置路由，配置后再次从连接池中查找是否有可复用连接，有则直接返回
- 新建一个连接，并修改其 StreamAllocation 标记计数，将其放入连接池中
- 查看连接池是否有重复的多路复用连接，有则清除

2.2.2 ConnectionPool.get

接下来再来看 get 方法的源码：

```
[ConnectionPool.java]

    RealConnection get(Address address, StreamAllocation streamAllocation, Route route) {

        assert (Thread.holdsLock(this));

        for (RealConnection connection : connections) {

            if (connection.isEligible(address, route)) {

                streamAllocation.acquire(connection);

                return connection;

            }

        }

        return null;

    }
```

其逻辑比较简单，遍历当前连接池，如果有符合条件的连接则修改器标记计数，然后返回。这里的关键逻辑在 RealConnection.isEligible 方法：

```
[RealConnection.java]/**

 * Returns true if this connection can carry a stream allocation to {@code
address}. If non-null

 * {@code route} is the resolved route for a connection.

 */

public boolean isEligible(Address address, Route route) {

    // If this connection is not accepting new streams, we're done.
```

```
if (allocations.size() >= allocationLimit || noNewStreams) return false;

// If the non-host fields of the address don't overlap, we're done.

if (!Internal.instance.equalsNonHost(this.route.address(), address)) return false;

// If the host exactly matches, we're done: this connection can carry the address.

if (address.url().host().equals(this.route().address().url().host())) {

    return true; // This connection is a perfect match.

}

// At this point we don't have a hostname match. But we still be able to carry the request if

// our connection coalescing requirements are met. See also:

// https://hpbn.co/optimizing-application-delivery/#eliminate-domain-sharding

// https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing/

// 1. This connection must be HTTP/2.

if (http2Connection == null) return false;

// 2. The routes must share an IP address. This requires us to have a DNS address for both

// hosts, which only happens after route planning. We can't coalesce connections that use a

// proxy, since proxies don't tell us the origin server's IP address.

if (route == null) return false;
```

```

    if (route.proxy().type() != Proxy.Type.DIRECT) return false;

    if (this.route.proxy().type() != Proxy.Type.DIRECT) return false;

    if (!this.route.socketAddress().equals(route.socketAddress())) return false;

    // 3. This connection's server certificate's must cover the new host.

    if (route.address().hostnameVerifier() != OkHostnameVerifier.INSTANCE) return false;

    if (!supportsUrl(address.url())) return false;

    // 4. Certificate pinning must match the host.

    try {

        address.certificatePinner().check(address.url().host(), handshake().peerCertificates());

    } catch (SSLPeerUnverifiedException e) {

        return false;

    }

    return true; // The caller's address can be carried by this connection.

}

```

- 连接没有达到共享上限
- 非 host 域必须完全一样
- 如果此时 host 域也相同，则符合条件，可以被复用
- 如果 host 不相同，在 HTTP/2 的域名切片场景下一样可以复用，具体细节可以参考：
<https://hpbn.co/optimizing-application-delivery/#eliminate-domain-sharding>

2.2.3 deduplicate

`deduplicate` 方法主要是针对在 HTTP/2 场景下多个多路复用连接清除的场景。如果当前连接是 HTTP/2，那么所有指向该站点的请求都应该基于同一个 TCP 连接：

[ConnectionPool.java]

```
/*
 * Replaces the connection held by {@code streamAllocation} with a shared
connection if possible.

 * This recovers when multiple multiplexed connections are created concurrently.

 */

Socket deduplicate(Address address, StreamAllocation streamAllocation) {

    assert (Thread.holdsLock(this));

    for (RealConnection connection : connections) {

        if (connection.isEligible(address, null)

            && connection.isMultiplexed()

            && connection != streamAllocation.connection()) {

            return streamAllocation.releaseAndAcquire(connection);

        }

    }

    return null;
}
```

`put` 和 `evictAll` 比较简单，在这里就不写了，大家自行看源码。

2.3 自动回收

连接池中有 `socket` 回收，而这个回收是以 `RealConnection` 的弱引用 `List<Reference<StreamAllocation>>` 是否为 0 来为依据的。`ConnectionPool` 有一个独立的线程 `cleanupRunnable` 来清理连接池，其触发时机有两个：

- 当连接池中 `put` 新的连接时

- 当 connectionBecameIdle 接口被调用时

其代码如下：

```
while (true) {

    //执行清理并返回下次需要清理的时间

    long waitNanos = cleanup(System.nanoTime());

    if (waitNanos == -1) return;

    if (waitNanos > 0) {

        synchronized (ConnectionPool.this) {

            try {

                //在timeout 内释放锁与时间片

                ConnectionPool.this.wait(TimeUnit.NANOSECONDS.toMillis(waitNanos));

            } catch (InterruptedException ignored) {

            }

        }
    }
}
```

这段死循环实际上是一个阻塞的清理任务，首先进行清理(clean)，并返回下次需要清理的间隔时间，然后调用 `wait(timeout)` 进行等待以释放锁与时间片，当等待时间到了后，再次进行清理，并返回下次要清理的间隔时间...

接下来看下 `cleanup` 函数：

```
[ConnectionPool.java]long cleanup(long now) {

    int inUseConnectionCount = 0;

    int idleConnectionCount = 0;

    RealConnection longestIdleConnection = null;

    long longestIdleDurationNs = Long.MIN_VALUE;
```

```
//遍历`Deque` 中所有的`RealConnection`， 标记泄漏的连接

synchronized (this) {

    for (RealConnection connection : connections) {

        // 查询此连接内部 StreamAllocation 的引用数量

        if (pruneAndGetAllocationCount(connection, now) > 0) {

            inUseConnectionCount++;

            continue;

        }

        idleConnectionCount++;

    }

}

//选择排序法， 标记出空闲连接

long idleDurationNs = now - connection.idleAtNanos;

if (idleDurationNs > longestIdleDurationNs) {

    longestIdleDurationNs = idleDurationNs;

    longestIdleConnection = connection;

}

}

if (longestIdleDurationNs >= this.keepAliveDurationNs

    || idleConnectionCount > this.maxIdleConnections) {

    //如果(`空闲 socket 连接超过 5 个`)

    //且`keepalive`时间大于 5 分钟`)

    //就将此泄漏连接从`Deque`中移除

    connections.remove(longestIdleConnection);

}
```

```

} else if (idleConnectionCount > 0) {

    //返回此连接即将到期的时间，供下次清理

    //这里依据是在上文`connectionBecameIdle`中设定的计时

    return keepAliveDurationNs - longestIdleDurationNs;

} else if (inUseConnectionCount > 0) {

    //全部都是活跃的连接，5分钟后再次清理

    return keepAliveDurationNs;

} else {

    //没有任何连接，跳出循环

    cleanupRunning = false;

    return -1;

}

}

//关闭连接，返回`0`，也就是立刻再次清理

closeQuietly(longestIdleConnection.socket());

return 0;
}

```

其基本逻辑如下：

- 遍历连接池中所有连接，标记泄露连接
- 如果被标记的连接满足(空闲 socket 连接超过 5 个&&keepalive 时间大于 5 分钟)，就将此连接从 Deque 中移除，并关闭连接，返回 0，也就是将要执行 wait(0)，提醒立刻再次扫描
- 如果(目前还可以塞得下 5 个连接，但是有可能泄漏的连接(即空闲时间即将达到 5 分钟))，就返回此连接即将到期的剩余时间，供下次清理
- 如果(全部都是活跃的连接)，就返回默认的 keep-alive 时间，也就是 5 分钟后再执行清理

而 pruneAndGetAllocationCount 负责标记并找到不活跃连接:

```
[ConnnectionPool.java]//类似于引用计数法, 如果引用全部为空, 返回立刻清理 private  
int pruneAndGetAllocationCount(RealConnection connection, long now) {  
  
    //虚引用列表  
  
    List<Reference<StreamAllocation>> references = connection.allocations;  
  
    //遍历弱引用列表  
  
    for (int i = 0; i < references.size(); ) {  
  
        Reference<StreamAllocation> reference = references.get(i);  
  
        //如果正在被使用, 跳过, 接着循环  
  
        //是否置空是在上文`connectionBecameIdle`的`release`控制的  
  
        if (reference.get() != null) {  
  
            //非常明显的引用计数  
  
            i++;  
  
            continue;  
  
        }  
  
        //否则移除引用  
  
        references.remove(i);  
  
        connection.noNewStreams = true;  
  
        //如果所有分配的流均没了, 标记为已经距离现在空闲了5分钟  
  
        if (references.isEmpty()) {  
  
            connection.idleAtNanos = now - keepAliveDurationNs;  
  
            return 0;  
  
        }  
    }  
}
```

```
    return references.size();  
}
```

OkHttp 的连接池通过计数+标记清理的机制来管理连接池，使得无用连接可以被会回收，并保持多个健康的 keep-alive 连接。这也是 OkHttp 的连接池能保持高效的关键原因。

7.深入解析 ButterKnife 源码

概述版

- 写代码时对目标元素进行注解，编译时注解处理器会扫描注解，并通过 JavaPoet 自动生成 Java 文件
- 调用 ButterKnife.bind()方法时，通过反射拿到编译时生成的类，调用其构造方法完成目标类和绑定类的绑定

分析版

1、注解处理器 ButterKnifeProcessor 在编译时会扫描和处理所有注解，通过 JavaPoet 自动生成 .java 文件

2、调用 ButterKnife 的 bind 方法，通过反射拿到目标类类名，后面拼接

ViewBinding 找到刚刚自动生成的 Java 类，并根据参数找到相匹配的构造器

3、调用 ViewBinding 类构造器的构造方法，将 ViewBinding 类和 目标类进行绑定

4、ViewBinding 类的构造方法中会对目标类的成员变量进行初始化及点击事件配置

[ButterKnife 源码分析完整版地址](#)

8.深入解析 Okio 源码（一套简洁高效的 I/O 库）

从前面的 OkHttp 源码解析中我们可以知道，OkHttp 中的 I/O 都不是通过我们平时所使用的 IOStream 来实现，而是使用了 Okio 这个第三方库，那它与寻常的 IOStream 有什么区别呢？让我们来分析一下它的源码。

Okio 中有两个非常重要的接口——`Sink` 以及 `Source`，它们都继承了 `Closeable`，其中 `Sink` 对应了我们原来所使用的 `OutputStream`，而 `Source` 则对应了我们原来所使用的 `InputStream`。

Okio 的入口就是 `Okio` 类，它是一个工厂类，可以通过它内部的一些 `static` 方法来创建 `Sink`、`Source` 等对象。

Sink

`Sink` 实际上只是一个接口，让我们看看 `Sink` 中有哪些方法：

```
public interface Sink extends Closeable, Flushable {
    void write(Buffer source, long byteCount) throws IOException;
    @Override void flush() throws IOException;
    Timeout timeout();
    @Override void close() throws IOException;
}
```

可以看到，它主要包含了 `write`、`flush`、`timeout`、`close` 这几个方法，我们可以
以通过 `Okio.sink` 方法基于 `OutputStream` 获取一个 `Sink`:

```
private static Sink sink(final OutputStream out, final Timeout timeout) {
    if (out == null) throw new IllegalArgumentException("out == null");
    if (timeout == null) throw new IllegalArgumentException("timeout == null");
    return new Sink() {
        @Override public void write(Buffer source, long byteCount) throws IOException {
            checkOffsetAndCount(source.size, 0, byteCount);
            while (byteCount > 0) {
                timeout.throwIfReached();
                Segment head = source.head;
                int toCopy = (int) Math.min(byteCount, head.limit - head.pos);
                out.write(head.data, head.pos, toCopy);
                head.pos += toCopy;
                byteCount -= toCopy;
                source.size -= toCopy;
                if (head.pos == head.limit) {
                    source.head = head.pop();
                    SegmentPool.recycle(head);
                }
            }
        }
        @Override public void flush() throws IOException {
            out.flush();
        }
        @Override public void close() throws IOException {
            out.close();
        }
        @Override public Timeout timeout() {
            return timeout;
        }
        @Override public String toString() {
            return "sink(" + out + ")";
        }
    };
}
```

这里构建并实现了一个 `Sink` 的匿名内部类并返回，主要实现了它的 `write` 方法，
剩余方法都是简单地转调到 `OutputStream` 的对应方法。

在 `write` 方法中，首先进行了一些状态检验，这里貌似在 `Timeout` 类中实现了对超时的处理，我们稍后再分析。之后从 `Buffer` 中获取了一个 `Segment`，并从中取出数据，计算出写入的量后将其写入 `Sink` 所对应的 `OutputStream`。

`Segment` 采用了一种类似链表的形式进行连接，看来 `Buffer` 中维护了一个 `Segment` 链表，代表了数据的其中一段。这里将 `Buffer` 中的数据分段取出并写入了 `OutputStream` 中。

最后，通过 `SegmentPool.recycle` 方法对当前 `Segment` 进行回收。

从上面的代码中我们可以获取到如下信息：

1. `Buffer` 其实就是内存中的一段数据的抽象，其中通过 `Segment` 以链表的形式保存用于存储数据。
2. `Segment` 存储数据采用了分段的存储方式，因此获取数据时需要分段从 `Segment` 中获取数据。
3. 有一个 `SegmentPool` 池用于实现 `Segment` 的复用。
4. `Segment` 的使用有点类似链表。

Source

`Source` 与 `Sink` 一样，也仅仅是一个接口：

```
public interface Source extends Closeable {  
    long read(Buffer sink, long byteCount) throws IOException;  
    Timeout timeout();  
    @Override void close() throws IOException;
```

```
}
```

在 Okio 中可以通过 `source` 方法根据 `InputStream` 创建一个 `Source`:

```
private static Source source(final InputStream in, final Timeout timeout) {
    if (in == null) {
        throw new IllegalArgumentException("in == null");
    } else if (timeout == null) {
        throw new IllegalArgumentException("timeout == null");
    } else {
        return new Source() {
            public long read(Buffer sink, long byteCount) throws IOException {
                if (byteCount < 0L) {
                    throw new IllegalArgumentException("byteCount < 0: " + byteC
ount);
                } else if (byteCount == 0L) {
                    return 0L;
                } else {
                    try {
                        timeout.throwIfReached();
                        Segment tail = sink.writableSegment(1);
                        int maxToCopy = (int) Math.min(byteCount, (long)(8192 - t
ail.limit));
                        int bytesRead = in.read(tail.data, tail.limit, maxToCop
y);
                        if (bytesRead == -1) {
                            return -1L;
                        } else {
                            tail.limit += bytesRead;
                            sink.size += (long)bytesRead;
                            return (long)bytesRead;
                        }
                    } catch (AssertionError var7) {
                        if (Okio.isAndroidGetsocknameError(var7)) {
                            throw new IOException(var7);
                        } else {
                            throw var7;
                        }
                    }
                }
            }
            public void close() throws IOException {
                in.close();
            }
        };
    }
}
```

```
        public Timeout timeout() {
            return timeout;
        }
        public String toString() {
            return "source(" + in + ")";
        }
    };
}
}
```

这里构建并实现了 `Source` 的一个匿名内部类并返回，应该就是 `Source` 的默认实现了。

它除了 `read` 方法其他都只是简单地调用了 `InputStream` 的对应方法，我们重点看 `read` 方法：

首先它进行了一些相关状态检测，之后通过 `sink.writeableSegment` 获取到了一个可以写入的 `Segment`。之后从 `InputStream` 中读取数据向 `Segment` 中写入，读取的大小被限制为了 8192 个字节。

Buffer

`Buffer` 在 `Sink` 及 `Source` 中都担任了一个十分重要的地位，它对应了我们内存中存储的数据，对这些数据进行了抽象。下面让我们对 `Buffer` 进行分析：

`Buffer` 虽然是我们内存中数据的抽象，但数据实际上并不是存储在 `Buffer` 中的，它在内部维护了一个 `Segment` 的循环链表，`Segment` 才是真正存储数据的地方。它通过 `Segment` 将数据分成了几段，通过链表进行连接。在 `Buffer` 内部封装了许多 I/O 操作，都是在对 `Segment` 中的数据进行处理。

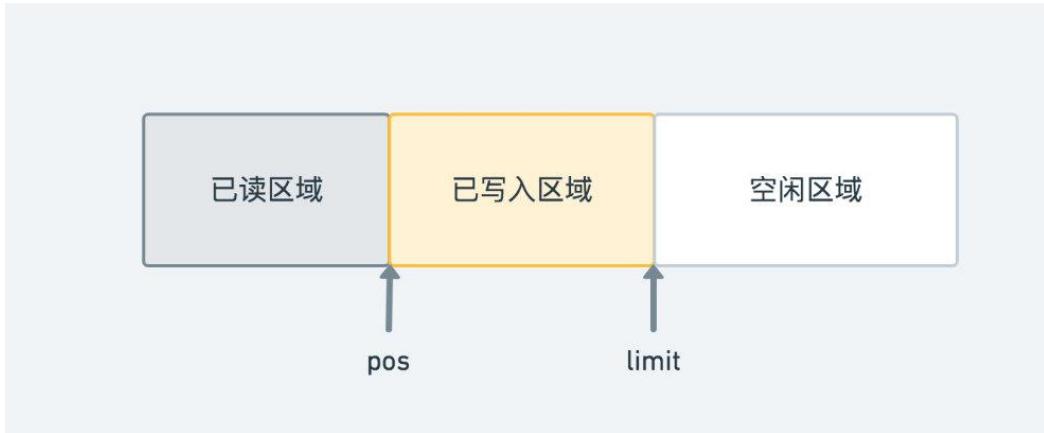
为什么要使用 `Segment` 对数据进行分段存储而不直接存储整个数据呢？由于数据是分段存放的，这些段中的某一部分可能与另一个 `Buffer` 中的数据恰好是相同的，此时就体现出了 `Segment` 的灵活性，我们不需要将数据拷贝到另一个 `Buffer` 中，只需要将其 `Segment` 指向这个重复段的 `Segment` 即可。同时，对于一些如将数据从 `Source` 转移到 `Sink` 中这种情况，也不需要进行拷贝，只需要将链表指向我们的 `Segment` 即可，极大地提高了效率，同时节省了内存空间。

Segment

我们首先看一下存储数据的 `Segment`，它代表了数据中的一段，是一个双向的循环链表，主要有以下的参数：

```
final class Segment {  
    // Segment 存储数据的大小  
    static final int SIZE = 8192;  
    // 进行数据共享的最小字节数  
    static final int SHARE_MINIMUM = 1024;  
    // 存储数据的字节数组  
    final byte[] data;  
    // 用户读取数据的下一个起始位置  
    int pos;  
    // 可以被写入的下一个起始位置  
    int limit;  
    // 数据是否已被共享  
    boolean shared;  
    // 该字节数组是否属于该 Segment  
    boolean owner;  
    // 链表指针  
    Segment next;  
    // 链表指针  
    Segment prev;  
    //...  
}
```

可以看到，其中 `pos` 代表了下一次读取的起始位置，而 `limit` 代表了下一次写入的起始位置，我们可以根据它们两个值将整个 `Segment` 的空间分为如图的三段：



其中已读区域的数据我们以后都不会再用到，已写入区域的数据正在等待读取，而空闲区域还没有填入数据，可以进行写入。

共享机制

同时，`Segment` 还支持了对数据的共享，通过 `shared` 及 `owner` 字段分别表明了数据是否已被共享以及其是否属于当前 `Segment`。同时它提供了两种拷贝方式：`sharedCopy` 以及 `unsharedCopy`。

`unsharedCopy` 返回了一个新的 `Segment`，并将 `data` 数组通过 `clone` 方法拷贝到了新 `Segment` 中：

```
/** Returns a new segment that its own private copy of the underlying byte array. */
final Segment unsharedCopy() {
    return new Segment(data.clone(), pos, limit, false, true);
}
```

而 `sharedCopy` 同样返回了一个新的 `Segment`, 但其 `data` 数组是与新 `Segment` 进行共享的:

```
/**  
 * Returns a new segment that shares the underlying byte array with this. Adjusting pos and limit  
 * are safe but writes are forbidden. This also marks the current segment as shared, which  
 * prevents it from being pooled.  
 */  
final Segment sharedCopy() {  
    shared = true;  
    return new Segment(data, pos, limit, true, false);  
}
```

同时通过注释我们可以看到, 当数据共享后, 为了保证安全性, 禁止了写入操作。同时将被拷贝的 `Segment` 也标记为了 `shared`, 从而防止其被回收。

这样的设计同样是为了减少拷贝, 从而提高 I/O 的效率。

合并与分割

`Segment` 还支持了与前一个 `Segment` 的合并以及对自身的分割操作, 从而使得使用者能够更灵活地操作。

合并操作会在当前 `Segment` 与它的前一个节点都没有超过其大小的一半时, 将二者的数据进行合并, 并将当前 `Segment` 进行回收, 从而增大内存的利用效率:

```
/**  
 * Call this when the tail and its predecessor may both be less than half full. This will copy data so that segments can be recycled.  
 */  
public final void compact() {  
    if (prev == this) throw new IllegalStateException();  
    // 上一个节点的数据不是可以写入的(是共享数据), 取消合并  
    if (!prev.owner) return;
```

```

    // 计算当前节点与前一个节点的剩余空间
    int byteCount = limit - pos;
    int availableByteCount = SIZE - prev.limit + (prev.shared ? 0 : prev.pos);
    // 没有足够的写入空间时，不进行合并
    if (byteCount > availableByteCount) return;
    // 进行合并，将当前节点数据写入前一节点
    writeTo(prev, byteCount);
    // 从链表中删除当前节点，并进行回收
    pop();
    SegmentPool.recycle(this);
}

```

而分割操作则会将 Segment 中的数据分割为 [pos,

pos+byteCount) 及 [pos+byteCount, limit) 的两段：

```

public final Segment split(int byteCount) {
    if (byteCount <= 0 || byteCount > limit - pos) throw new IllegalArgumentException();
    Segment prefix;
    // We have two competing performance goals:
    // - Avoid copying data. We accomplish this by sharing segments.
    // - Avoid short shared segments. These are bad for performance because they
    // are readonly and
    // may lead to long chains of short segments.
    // To balance these goals we only share segments when the copy will be large.
    if (byteCount >= SHARE_MINIMUM) {
        // 如果拷贝量大于1024 字节，通过共享的形式
        prefix = sharedCopy();
    } else {
        // 拷贝量低于1024 字节，通过arrayCopy 进行拷贝
        prefix = SegmentPool.take();
        System.arraycopy(data, pos, prefix.data, 0, byteCount);
    }
    // 对Limit 及 pos 进行修改
    prefix.limit = prefix.pos + byteCount;
    pos += byteCount;
    prev.push(prefix);
    return prefix;
}

```

这里首先对不同数据段的数据进行了处理，如果数据段大于了 1024 字节，则将数据通过共享交给了分割的前一个节点，两端 `Segment` 公用同一个 `data` 数组，否则通过拷贝的形式构建一个新的 `Segment`。

为什么这里需要对数据大小的不同采用不同的处理方式呢？我们可以看到上面的注释，里面给出了答案：首先，为了避免拷贝数据带来的性能开销，加入了共享 `Segment` 的功能。但是由于共享的数据是只读的，如果有很多很短的数据段的话，使用的表现并不会很好，因此只有当拷贝的数据量比较大时，才会进行 `Segment` 的共享。

之后，将二者的 `pos` 及 `limit` 都进行了设置。由于 `pos` 之前的部分及 `limit` 之后的部分都不会影响到我们正常的读取和写入，因此我们可以不用关心它们目前的状态，没必要再对它们进行一些如填充零之类的操作。

SegmentPool

同时，Okio 还使用了 `SegmentPool` 来实现一个对象池，从而避免 `Segment` 频繁地创建及销毁所带来的性能开销。

`SegmentPool` 的实现十分简单，它内部维护了一个单链表，用于存储被回收存在池中的 `Segment`，其最大容量被限制在了 64 k。

当需要 `Segment` 时，可以通过 `take` 方法来获取一个被回收的对象：

```
static Segment take() {
    synchronized (SegmentPool.class) {
        if (next != null) {
            Segment result = next;
            next = result.next;
        }
    }
}
```

```
        result.next = null;
        byteCount -= Segment.SIZE;
        return result;
    }
}

return new Segment(); // Pool is empty. Don't zero-fill while holding a Lock.
}
```

它会在单链表中找到一个空闲的 `Segment` 并初始化后返回。若当前链表中没有对象，则会创建一个新的 `Segment`。

当 `Segment` 使用完毕时，首先可以通过 `Segment` 的 `pop` 操作将其从链表中移除，之后可以调用 `SegmentPool.recycle` 方法对其进行回收：

```
static void recycle(Segment segment) {
    if (segment.next != null || segment.prev != null) throw new IllegalArgumentException();
    if (segment.shared) return; // This segment cannot be recycled.
    synchronized (SegmentPool.class) {
        if (byteCount + Segment.SIZE > MAX_SIZE) return; // Pool is full.
        byteCount += Segment.SIZE;
        segment.next = next;
        segment.pos = segment.limit = 0;
        next = segment;
    }
}
```

回收 `Segment` 时，不会对只读的 `Segment` 进行回收，若 `Segment` 个数超过了上限，则不会对该 `Segment` 进行回收。

数据转移

Okio 与 `java.io` 有个很大的不同，体现在 `Buffer` 的数据转移上，我们可以通过其 `copyTo` 方法来完成数据的转移。之所以叫转移，因为它相对于复制来说，是

有很大的数据提升的。例如我们可以看到两个 `Buffer` 之间的数据转移是如何进行的：

```
public final Buffer copyTo(Buffer out, long offset, long byteCount) {
    if (out == null) throw new IllegalArgumentException("out == null");
    checkOffsetAndCount(size, offset, byteCount);
    if (byteCount == 0) return this;
    out.size += byteCount;
    // 跳过不进行拷贝的 Segment
    Segment s = head;
    for (; offset >= (s.limit - s.pos); s = s.next) {
        offset -= (s.limit - s.pos);
    }
    for (; byteCount > 0; s = s.next) {
        // 通过 sharedCopy 将数据拷贝到 copy 中
        Segment copy = s.sharedCopy();
        copy.pos += offset;
        copy.limit = Math.min(copy.pos + (int) byteCount, copy.limit);
        // 插入 Segment
        if (out.head == null) {
            out.head = copy.next = copy.prev = copy;
        } else {
            out.head.prev.push(copy);
        }
        byteCount -= copy.limit - copy.pos;
        offset = 0;
    }
    return this;
}
```

从上面的代码中可以看出，实际上这个过程是通过了 `Segment` 共享实现的，因此不需要进行拷贝，极大地提高了数据转移的效率。

BufferedSource

我们可以通过 `Okio.buffer` 方法对一个普通的 `Source` 进行包装，获取一个具有缓冲能力的 `BufferSource`，它是一个接口，定义了一系列读取的方法：

```

public interface BufferedSource extends Source, ReadableByteChannel {
    @Deprecated
    Buffer buffer();
    Buffer getBuffer();
    boolean exhausted() throws IOException;
    void require(long byteCount) throws IOException;
    boolean request(long byteCount) throws IOException;
    byte readByte() throws IOException;
    short readShort() throws IOException;
    short readShortLe() throws IOException;
    // ... 一系列读取方法
    long indexOf(byte b, long fromIndex) throws IOException;
    long indexOf(byte b, long fromIndex, long toIndex) throws IOException;
    long indexOf(ByteString bytes) throws IOException;
    long indexOf(ByteString bytes, long fromIndex) throws IOException;
    long indexOfElement(ByteString targetBytes) throws IOException;
    long indexOfElement(ByteString targetBytes, long fromIndex) throws IOException;
    boolean rangeEquals(long offset, ByteString bytes) throws IOException;
    boolean rangeEquals(long offset, ByteString bytes, int bytesOffset, int byteCount)
        throws IOException;
    BufferedSource peek();
    InputStream inputStream();
}

```

它主要有两个实现类: `Buffer` 与 `RealBufferedSource`。其中 `RealBufferedSource` 显然是我们通过 `buffer` 方法包装后得到的类, 而 `Buffer` 实际上对 `BufferSource` 也进行了实现, 通过一系列 `read` 方法可以从 `Segment` 中读取处对应的数据。而我们的 `RealBufferedSource` 则是 `Source` 的一个包装类, 并且其维护了一个 `Buffer`, 从而提高 `Input` 的效率。我们先分析其思路, 再来讨论为什么这样能提高 `Input` 的效率。

我们可以首先看到 `RealBufferedSource` 的读取方法, 这里以 `readByteArray` 方法举例:

```

@Override public byte[] readByteArray(long byteCount) throws IOException {

```

```
    require(byteCount);
    return buffer.readByteArray(byteCount);
}
```

这里首先调用了 `require` 方法，之后再从 `buffer` 中将数据读出，看来在 `require` 中将数据先读取到了 `buffer` 中。

我们看到 `require` 方法：

```
@Override public void require(long byteCount) throws IOException {
    if (!request(byteCount)) throw new EOFException();
}
```

它实际上转调到了 `request` 方法：

```
@Override public boolean request(long byteCount) throws IOException {
    if (byteCount < 0) throw new IllegalArgumentException("byteCount < 0: " + byteCount);
    if (closed) throw new IllegalStateException("closed");
    while (buffer.size < byteCount) {
        if (source.read(buffer, Segment.SIZE) == -1) return false;
    }
    return true;
}
```

`request` 方法中，不断地向 `buffer` 中读取，每次读取 `Segment.SIZE` 也就是 8192 个字节。也就是它读取的量是 `byteCount` 以 8192 字节向上取整。为什么它不刚好读取 `byteCount` 个字节，要读满 8192 个字节呢？

这就是一种预取思想，因为 I/O 操作往往是非常频繁的，如果进行了一次读取，那就很有可能还会进行下一次读取，因此我们预先把它下一次可能读取的部分一起读取出来，这样下次读取时，就不需要再对系统进行请求以获取数据了，可以直接从我们的 `buffer` 中拿到。这就是为什么说加入 `buffer` 提高了我们 I/O 的效率。

可能还有人会问，为什么这样能提高 I/O 效率呢，不都是读了一样的量么？这个就涉及到一些操作系统的知识了。在现代的操作系统中，我们的程序往往运行在用户态，而用户态实际上是没有进行 I/O 的权限的，因此往往都是向操作系统发起请求，切换到内核态，再进行 I/O，完成后再次回到用户态。这样的用户态及内核态的切换实际上是非常耗时的，并且这个过程中也伴随着拷贝。因此采用上面的 `buffer` 可以有效地减少我们的这种系统 I/O 调用，加快我们的效率。

BufferedSink

我们同样可以通过 `Okio.buffer` 方法对一个普通的 `Sink` 进行包装，从而获取一个带有 `buffer` 缓冲能力的 `BufferedSink`。`BufferedSink` 也是一个接口，内部定义了一系列写入的方法：

```
public interface BufferedSink extends Sink, WritableByteChannel {
    Buffer buffer();
    BufferedSink write(ByteString byteString) throws IOException;
    BufferedSink write(byte[] source) throws IOException;
    BufferedSink write(byte[] source, int offset, int byteCount) throws IOException;
    long writeAll(Source source) throws IOException;
    BufferedSink write(Source source, long byteCount) throws IOException;
    // ... 一些对 write 的封装
    @Override void flush() throws IOException;
    BufferedSink emit() throws IOException;
    BufferedSink emitCompleteSegments() throws IOException;
    OutputStream outputStream();
}
```

`BufferedSink` 同样有两个实现类：`Buffer` 和 `RealBufferedSink`，我们可以先看到 `RealBufferedSink`，它是一个 `Sink` 的包装类，并且内部维护了一个 `Buffer`。

write

我们先看看其写入方法：

```
@Override public BufferedSink write(byte[] source, int offset, int byteCount)
throws IOException {
    if (closed) throw new IllegalStateException("closed");
    buffer.write(source, offset, byteCount);
    return emitCompleteSegments();
}
```

这里拿了一个简单的写入 `byte[]` 的方法进行了举例，它首先将数据写入了 `buffer` 中，之后调用了 `emitCompleteSegments` 方法。可以看到这里并没有对 `sink` 真正进行写入，那写入究竟是在哪里进行的呢？我们看看 `emitCompleteSegments` 方法中做了什么：

```
@Override public BufferedSink emitCompleteSegments() throws IOException {
    if (closed) throw new IllegalStateException("closed");
    long byteCount = buffer.completeSegmentByteCount();
    if (byteCount > 0) sink.write(buffer, byteCount);
    return this;
}
```

这里首先调用 `buffer.completeSegmentByteCount` 方法获取到了 `buffer` 中已写入但未被读取的部分字节数（只包括已经被写满了的 `Segment` 中的），之后调用 `sink.write` 将其写入到了 `sink` 中。

这里其实很奇怪，按道理来说 `buffer` 的作用是通过缓存来进行一些优化，但这个方法将数据写入 `buffer` 后，数据又立即被写入到了 `sink` 中。这样相比直接写入到 `sink` 中，反而会带来性能的损耗啊。这里为什么要这样做呢？

我看到这里时对这段也比较奇怪，但考虑到 `Okio` 的整体设计来说，应该是把 `Buffer` 当做了一个数据统一的中转站，将读写的优化统一放在了 `Buffer` 中进行，因此考虑到整体的一致性，将 `RealBufferedSink` 也采用了通过 `Buffer` 中转

的方式编写，应该算是一种妥协吧。并且采用 `Buffer` 还有好处就是，一份数据既可以用于读也可以用于写。

flush

`RealBufferedSink` 还支持了 `flush` 操作，通过 `flush` 方法可以将缓冲区的所有数据写入 `sink` 中：

```
@Override public void flush() throws IOException {
    if (closed) throw new IllegalStateException("closed");
    if (buffer.size > 0) {
        sink.write(buffer, buffer.size);
    }
    sink.flush();
}
```

emit

`RealBufferedSink` 还具有 `emit` 功能，分别是 `emitCompleteSegments` 方法及 `emit` 方法，前者是将所有已填满的 `Segment` 中已写入未读取的数据写入 `sink`，后者则是将 `buffer` 中所有已写入未读取数据写入 `sink`（类似 `flush`）：

```
@Override public BufferedSink emitCompleteSegments() throws IOException {
    if (closed) throw new IllegalStateException("closed");
    long byteCount = buffer.completeSegmentByteCount();
    if (byteCount > 0) sink.write(buffer, byteCount);
    return this;
}
@Override public BufferedSink emit() throws IOException {
    if (closed) throw new IllegalStateException("closed");
    long byteCount = buffer.size();
    if (byteCount > 0) sink.write(buffer, byteCount);
    return this;
}
```

Timeout 超时机制

Okio 中通过 `Timeout` 类实现了 `Sink` 与 `Source` 的超时机制，在 `Sink` 的写入与 `Source` 的读取时对超时进行判断，如果超时则中断写入等操作。其中对于包装了普通 `InputStream / OutputStream` 的使用了普通的 `Timeout`，而对于对 `Socket` 进行了包装的则使用 `AsyncTimeout`。

Timeout

我们先对普通的 `Timeout` 进行研究，`Timeout` 中主要有两个值，`timeout` 与 `deadline`，分别代表了 `wait` 的最大等待时间与完成某个工作的超时时间。

deadline

对于 `deadline`，我们可以通过 `deadline` 方法进行设定：

```
/** Set a deadline of now plus {@code duration} time. */
public final Timeout deadline(long duration, TimeUnit unit) {
    if (duration <= 0) throw new IllegalArgumentException("duration <= 0: " + duration);
    if (unit == null) throw new IllegalArgumentException("unit == null");
    return deadlineNanoTime(System.nanoTime() + unit.toNanos(duration));
}
```

之后，在每个需要检查超时的地方需要调用该 `Timeout` 的 `throwIfReached` 方法（如 `Sink` 的 `write` 方法）：

```
public void throwIfReached() throws IOException {
    if (Thread.interrupted()) {
        Thread.currentThread().interrupt(); // Retain interrupted status.
        throw new InterruptedIOException("interrupted");
    }
}
```

```
        }
        if (hasDeadline && deadlineNanoTime - System.nanoTime() <= 0) {
            throw new InterruptedIOException("deadline reached");
        }
    }
}
```

这里很简单，就是进行时间的校验，若达到了设定的时间，则抛出异常从而中断后续操作。

timeout

同时，`Timeout` 还实现了对 `monitor` 进行 `wait` 的超时机制，通过 `waitForNotified` 方法可以等待 `monitor` 被 `notify`，若等待的过程超过了 `Timeout` 所设定的时间或当前线程被中断，则会抛出异常，从而避免一直进行等待。并且，该方法需要在 `synchronized` 代码块中调用，以保证线程安全。

在我们构造了一个 `Timeout` 后，可以使用 `timeout` 方法对其 `wait` 超时时间进行设定：

```
public Timeout timeout(long timeout, TimeUnit unit) {
    if (timeout < 0) throw new IllegalArgumentException("timeout < 0: " + timeout);
    if (unit == null) throw new IllegalArgumentException("unit == null");
    this.timeoutNanos = unit.toNanos(timeout);
    return this;
}
```

这里主要是将 `timeoutNanos` 设置为了对应的值。接着我们看到 `waitForNotified` 方法：

```
/*
 * Waits on {@code monitor} until it is notified. Throws {@link InterruptedIOException}
 * if either
 *   * the thread is interrupted or if this timeout elapses before {@code monitor}
 *     is notified. The
 *   * caller must be synchronized on {@code monitor}.
 */
```

```

/*
public final void waitUntilNotified(Object monitor) throws InterruptedException {
    try {
        boolean hasDeadline = hasDeadline();
        long timeoutNanos = timeoutNanos();
        // 没有 timeout 的设定的话, 直接调用 monitor 的 wait 方法
        if (!hasDeadline && timeoutNanos == 0L) {
            monitor.wait(); // There is no timeout: wait forever.
            return;
        }
        // 计算我们要 wait 的时间
        long waitNanos;
        long start = System.nanoTime();
        // 下面主要就是等待 timeout 与 deadline 中最小的那个
        if (hasDeadline && timeoutNanos != 0) {
            long deadlineNanos = deadlineNanoTime() - start;
            waitNanos = Math.min(timeoutNanos, deadlineNanos);
        } else if (hasDeadline) {
            waitNanos = deadlineNanoTime() - start;
        } else {
            waitNanos = timeoutNanos;
        }
        // wait 相应时间
        long elapsedNanos = 0L;
        if (waitNanos > 0L) {
            long waitMillis = waitNanos / 1000000L;
            monitor.wait(waitMillis, (int) (waitNanos - waitMillis * 1000000
L));
            elapsedNanos = System.nanoTime() - start;
        }
        // 如果还没有 notify, 则抛出异常
        if (elapsedNanos >= waitNanos) {
            throw new InterruptedException("timeout");
        }
    } catch (InterruptedException e) {
        // 线程如果在这个过程中被 interrupt, 则抛出异常
        Thread.currentThread().interrupt();
        throw new InterruptedException("interrupted");
    }
}

```

AsyncTimeout

`AsyncTimeout` 是 `Timeout` 的子类，接下来我们看看 `AsyncTimeout` 是如何对 `Socket` 中的超时进行处理的。

首先可以看到 `AsyncTimeout` 中保存了一个 `head` 及一个 `next` 引用，显然这里是有一个链表存储的 `AsyncTimeout` 队列的：

```
// AsyncTimeout 队列的头部
static @Nullable AsyncTimeout head;
// 当前节点是否在队列中
private boolean inQueue;
// 下一个节点
private @Nullable AsyncTimeout next;
```

这里感觉与 `MessageQueue` 有点相似，猜测 `AsyncTimeout` 会根据超时的时间按序存储在队列中。

并且从 `AsyncTimeout` 的 JavaDoc 中可以看到，它需要使用者在异步的事件开始时调用 `enter` 方法，结束时调用 `exit` 方法。同时它在背后开辟了一个线程对超时进行定时检查。

enter & exit

让我们先看到 `enter` 方法：

```
public final void enter() {
    if (inQueue) throw new IllegalStateException("Unbalanced enter/exit");
    long timeoutNanos = timeoutNanos();
    boolean hasDeadline = hasDeadline
    // 时间到了就不加入队列了
    if (timeoutNanos == 0 && !hasDeadline) {
        return;
    }
    inQueue = true;
    // 开启线程对超时进行检查
```

```
    scheduleTimeout(this, timeoutNanos, hasDeadline);
}
```

上面主要是将其 `inQueue` 设置为了 `true`, 之后调用 `scheduleTimeout` 方法对超时进行定时检查。我们暂时先不关注 `scheduleTimeout` 的具体实现。

接着我们看到 `exit` 方法:

```
/** Returns true if the timeout occurred. */
public final boolean exit() {
    if (!inQueue) return false;
    inQueue = false;
    return cancelScheduledTimeout(this);
}
```

这里也非常简单, 就是将 `inQueue` 设置为了 `false`, 并调用 `cancelScheduledTimeout` 方法停止前面的定时校验线程。

scheduleTimeout

我们接下来看看这个定时校验的具体实现, 我们先看到 `scheduleTimeout` 方法:

```
private static synchronized void scheduleTimeout(
    AsyncTimeout node, long timeoutNanos, boolean hasDeadline) {
    // 如果队列中还没有节点, 构造一个头节点并启动 Watchdog
    if (head == null) {
        head = new AsyncTimeout();
        new Watchdog().start();
    }
    long now = System.nanoTime();
    // 计算具体超时时间, 主要是取 timeout 与 deadline 的最小值
    if (timeoutNanos != 0 && hasDeadline) {
        // Compute the earliest event; either timeout or deadline. Because nano
        // Time can wrap around,
        // Math.min() is undefined for absolute values, but meaningful for rela
        // tive ones.
        node.timeoutAt = now + Math.min(timeoutNanos, node.deadlineNanoTime() -
now);
```

```

} else if (timeoutNanos != 0) {
    node.timeoutAt = now + timeoutNanos;
} else if (hasDeadline) {
    node.timeoutAt = node.deadlineNanoTime();
} else {
    throw new AssertionError();
}
long remainingNanos = node.remainingNanos(now);
// 按剩余时间从小到大插入到队列中
for (AsyncTimeout prev = head; true; prev = prev.next) {
    if (prev.next == null || remainingNanos < prev.next.remainingNanos(now))
    {
        node.next = prev.next;
        prev.next = node;
        if (prev == head) {
            // 插入在队列头部, 进行 notify
            AsyncTimeout.class.notify();
        }
        break;
    }
}
}

```

上面的逻辑主要分为以下几步：

1. 若队列中还没有节点，构造一个头节点并且启动 `Watchdog`, `Watchdog` 是一个 `Thread` 的子类，也就是我们的定时扫描线程。
2. 计算该 `Timeout` 的超时时间，取了 `timeout` 与 `deadline` 的最小值
3. 将该 `timeout` 按剩余时间从小到大的顺序插入队列中
4. 若插入的位置是队列的头部，则进行 `notify` (这里还无法了解到意图，我们可以往后看看)

cancelScheduledTimeout

接着我们看看 `cancelScheduledTimeout` 做了些什么：

```

private static synchronized boolean cancelScheduledTimeout(AsyncTimeout node)
{
    // Remove the node from the linked list.
    for (AsyncTimeout prev = head; prev != null; prev = prev.next) {
        if (prev.next == node) {
            prev.next = node.next;
            node.next = null;
            return false;
        }
    }
    // The node wasn't found in the linked list: it must have timed out!
    return true;
}

```

这里很简单，就是将该 `AsyncTimeout` 从队列中移除，若返回 `true`，则代表超时已经发生，若返回 `false`，则代表超时还未发生，该 `Timeout` 被移除。这个返回值同样反映到了我们的 `exit` 方法返回值中。

Watchdog

接着我们看看 `Watchdog` 究竟是如何对超时进行检测的：

```

private static final class Watchdog extends Thread {
    public void run() {
        while (true) {
            try {
                AsyncTimeout timedOut;
                synchronized (AsyncTimeout.class) {
                    timedOut = awaitTimeout();
                    // 找不到要 interrupt 的节点，继续寻找
                    if (timedOut == null) continue;
                    // 队列已空，停止线程
                    if (timedOut == head) {
                        head = null;
                        return;
                    }
                }
                // 调用 timeout 方法通知超时
                timedOut.timedOut();
            } catch (InterruptedException ignored) {

```

```
        }
    }
}
}
```

Watchdog 中不断调用 `awaitTimeout` 方法尝试获取一个可以停止的 `Timeout`, 之后调用了其 `timeOut` 方法通知外部已超时。

awaitTimeout

我们可以看看 `awaitTimeout` 做了什么:

```
static @Nullable AsyncTimeout awaitTimeout() throws InterruptedException {
    AsyncTimeout node = head.next;
    // 队列为空, wait 直到有新的节点加入
    if (node == null) {
        long startNanos = System.nanoTime();
        AsyncTimeout.class.wait(IDLE_TIMEOUT_MILLIS);
        return head.next == null && (System.nanoTime() - startNanos) >= IDLE_TIMEOUT_NANOS
            ? head // The idle timeout elapsed.
            : null; // The situation has changed.
    }
    long waitNanos = node.remainingNanos(System.nanoTime());
    // 计算该节点需要 wait 的时间
    if (waitNanos > 0) {
        // wait 对应的时间
        long waitMillis = waitNanos / 1000000L;
        waitNanos -= (waitMillis * 1000000L);
        AsyncTimeout.class.wait(waitMillis, (int) waitNanos);
        return null;
    }
    // wait 过后已超时, 将其移出队列
    head.next = node.next;
    node.next = null;
    return node;
}
```

这里主要有以下几步:

1. 如果队列为空，`wait` 直到有新的节点加入队列
2. 计算节点需要 `wait` 的时间并 `wait` 对应时间
3. 时间到后，说明该节点超时，将其移出队列

通过这里的代码，我们就知道为什么前面在链表头部加入节点时需要进行一次 `notify` 了，主要有两个目的：

1. 若队列中没有元素，可以通过 `notify` 通知此处有新元素加入队列。
2. 由于插入在头部，说明其比后面的节点的需要等待时间更少，因此需要停止前一次 `wait` 来计算该新的 `Timeout` 所需要的等待时间，并对其进行超时处理。

这里的处理和 Android 中 `MessageQueue` 的设计还是有异曲同工之妙的，我们可以学习一波。

sink & source

`AsyncTimeout` 还实现了 `sink` 及 `source` 方法来实现了支持 `AsyncTimeout` 超时机制的 `Sink` 及 `Source`，主要是通过在其各种操作前后分别调用 `enter` 及 `exit`。下面以 `Sink` 为例：

```
public final Sink sink(final Sink sink) {
    return new Sink() {
        @Override public void write(Buffer source, long byteCount) throws IOException {
            checkOffsetAndCount(source.size(), 0, byteCount);
            while (byteCount > 0L) {
                // Count how many bytes to write. This loop guarantees we split on a segment boundary.
                long toWrite = 0L;
                for (Segment s = source.head; toWrite < TIMEOUT_WRITE_SIZE; s = s.next)
                {
                    int segmentSize = s.limit - s.pos;
```

```
        toWrite += segmentSize;
        if (toWrite >= byteCount) {
            toWrite = byteCount;
            break;
        }
    }
    // Emit one write. Only this section is subject to the timeout.
    boolean throwOnTimeout = false;
    enter();
    try {
        sink.write(source, toWrite);
        byteCount -= toWrite;
        throwOnTimeout = true;
    } catch (IOException e) {
        throw exit(e);
    } finally {
        exit(throwOnTimeout);
    }
}
@Override public void flush() throws IOException {
    boolean throwOnTimeout = false;
    enter();
    try {
        sink.flush();
        throwOnTimeout = true;
    } catch (IOException e) {
        throw exit(e);
    } finally {
        exit(throwOnTimeout);
    }
}
@Override public void close() throws IOException {
    boolean throwOnTimeout = false;
    enter();
    try {
        sink.close();
        throwOnTimeout = true;
    } catch (IOException e) {
        throw exit(e);
    } finally {
        exit(throwOnTimeout);
    }
}
```

```
@Override public Timeout timeout() {
    return AsyncTimeout.this;
}
@Override public String toString() {
    return "AsyncTimeout.sink(" + sink + ")";
}
}
```

比较简单，这里就不做太多解释了。

总结

Okio 是一套基于 `java.io` 进行了一系列优化的十分优秀的 I/O 库，它通过引入了 `Segment` 机制大大降低了数据迁移的成本，减少了拷贝的次数，并且对 `java.io` 繁琐的体系进行了简化，使得整个库更易于使用。在 `okio` 中还实现了很多有其它功能的 `Source` 及 `Sink`，感兴趣的读者可以自行翻阅一下源码。同时各位可以去回顾一下前面的 `OkHttp` 源码解析中，`OkHttp` 是如何使用 `Okio` 进行 `Socket` 的数据写入及读取的。

9.深入解析 `SharedPreferences` 源码

`SharedPreferences` 是一个 Android 开发自带的适合保存轻量级数据的 K-V 存储库，它使用了 XML 的方式来存储数据，比如我就经常用它保存一些如用户登录信息等轻量级数据。那么今天就让我们来分析一下它的源码，研究一下其内部实现。

至于为什么今天会分析 `SharedPreferences` 呢？先卖个关子，其实是为了后面的文章做铺垫。

获取 `SharedPreferences`

我们在使用 `SharedPreferences` 时首先是需要获取到这个 `SharedPreferences` 的，因此我们首先从 `SharedPreferences` 的获取入手，来分析其源码。

根据名称获取 SP

不论是在 `Activity` 中调用 `getPreferences()` 方法还是调用 `Context` 的 `getSharedPreferences` 方法，最终都是调用到了 `ContextImpl` 的 `getSharedPreferences(String name, int mode)` 方法。我们先看看它的代码：

```
@Override
public SharedPreferences getSharedPreferences(String name, int mode) {
    // At least one application in the world actually passes in a null
    // name. This happened to work because when we generated the file name
    // we would stringify it to "null.xml". Nice.
    if (mPackageManager.getApplicationInfo().targetSdkVersion <
        Build.VERSION_CODES.KITKAT) {
        if (name == null) {
            name = "null";
        }
    }
    File file;
    synchronized (ContextImpl.class) {
        if (mSharedPrefsPaths == null) {
            mSharedPrefsPaths = new ArrayMap<>();
        }
        file = mSharedPrefsPaths.get(name);
        if (file == null) {
            file = getSharedPreferencesPath(name);
            mSharedPrefsPaths.put(name, file);
        }
    }
    return getSharedPreferences(file, mode);
}
```

可以看到，它首先对 Android 4.4 以下的设备做了特殊处理，之后将对 `mSharedPrefsPaths` 的操作加了锁。`mSharedPrefsPaths` 的声明如下：

```
private ArrayMap<String, File> mSharedPrefsPaths;
```

可以看到它是一个以 name 为 key, name 对应的 File 为 value 的 HashMap。首先调用了 getSharedPreferencesPath 方法构建出了 name 对应的 File, 将其放入 map 后再调用了 getSharedPreferences(File file, int mode) 方法。

获取 SP 名称对应的 File 对象

我们先看看是如何构建出 name 对应的 File 的。

```
@Override  
public File getSharedPreferencesPath(String name) {  
    return makeFilename(getPreferencesDir(), name + ".xml");  
}
```

可以看到, 调用了 makeFilename 方法来创建一个名为 name.xml 的 File。
makeFilename 中仅仅是做了一些判断, 之后 new 出了这个 File 对象并返回。

可以看到, SharedPreference 确实是使用 xml 来保存其中的 K-V 数据的, 而具体存储的路径我们这里就不再关心了, 有兴趣的可以点进去看看。

根据创建的 File 对象获取 SP

我们接着看到获取到 File 并放入 Map 后调用的 getSharedPreferences(file, mode) 方法:

```
@Override  
public SharedPreferences getSharedPreferences(File file, int mode) {  
    SharedPreferencesImpl sp;  
    synchronized (ContextImpl.class) {
```

```

        final ArrayMap<File, SharedPreferencesImpl> cache = getSharedPreferencesCacheLocked(); // 1
        sp = cache.get(file);
        if (sp == null) {
            checkMode(mode);
            if (getApplicationInfo().targetSdkVersion >= android.os.Build.VERSION_CODES.O) {
                if (isCredentialProtectedStorage())
                    && !getSystemService(UserManager.class)
                        .isUserUnlockingOrUnlocked(UserHandle.myUserId());
            }
            throw new IllegalStateException("SharedPreferences in credential encrypted "
                    + "storage are not available until after user is unlocked");
        }
        sp = new SharedPreferencesImpl(file, mode); // 2
        cache.put(file, sp);
        return sp;
    }
}

if ((mode & Context.MODE_MULTI_PROCESS) != 0 || getApplicationInfo().targetSdkVersion < android.os.Build.VERSION_CODES.HONEYCOMB) {
    // If somebody else (some other process) changed the prefs
    // file behind our back, we reload it. This has been the
    // historical (if undocumented) behavior.
    sp.startReloadIfChangedUnexpectedly();
}
return sp;
}

```

首先可以看到注释 1 处，这里调用了 `getSharedPreferencesCacheLocked` 来获取到了一个 `ArrayMap<file, SharedPreferencesImpl>`，之后再从这个 Map 中尝试获取到对应的 `SharedPreferencesImpl` 实现类（简称 SPI）。

之后看到注释 2 处，当获取不到对应 SPI 时，再创建一个对应的 SPI，并将其加入这个 ArrayMap 中。

这里很明显是一个缓存机制的实现，以加快之后获取 SP 的速度，同时可以发现，SP 其实只是一个接口，而 SPI 才是其具体的实现类。

缓存机制

那么我们先来看看其缓存机制，进入 getSharedPreferencesCacheLocked 方法：

```
private ArrayMap<File, SharedPreferencesImpl> getSharedPreferencesCacheLocked()
{
    if (sSharedPrefsCache == null) {
        sSharedPrefsCache = new ArrayMap<>();
    }
    final String packageName = getPackageName();
    ArrayMap<File, SharedPreferencesImpl> packagePrefs = sSharedPrefsCache.get(
        packageName);
    if (packagePrefs == null) {
        packagePrefs = new ArrayMap<>();
        sSharedPrefsCache.put(packageName, packagePrefs);
    }
    return packagePrefs;
}
```

显然，这里有个全局的 ArrayMap：sSharedPrefsCache。

它是一个 ArrayMap<string, arraymap<file,="" sharedpreferencesimpl="" style="box-sizing: border-box;">> 类型的 Map，而从代码中可以看出，它是根据 PackageName 来保存不同的 SP 缓存 Map 的，通过这样的方式，就保证了不同 PackageName 中相同 name 的 SP 从缓存中拿到的数据是不同的。

SharedPreferencesImpl

那么终于到了我们 SPI 的创建了，在 cache 中找不到对应的 SPI 时，就会 new 出一个 SPI，看看它的构造函数：

```
SharedPreferencesImpl(File file, int mode) {
    mFile = file;
    mBackupFile = makeBackupFile(file);      // 1
    mMode = mode;
    mLoaded = false;
    mMap = null;
    startLoadFromDisk();      // 2
}
```

可以看到注释 1 处它调用了 makeBackupFile 来进行备份文件的创建。

之后在注释 2 处则调用了 startLoadFromDisk 来开始从 Disk 载入信息。

首先我们看看 makeBackupFile 方法：

```
static File makeBackupFile(File prefsFile) {
    return new File(prefsFile.getPath() + ".bak");
}
```

很简单，返回了一个同目录下的后缀名为 .bak 的同名文件对象。

从 Disk 加载数据

接着，我们看看 startLoadFromDisk 方法：

```
private void startLoadFromDisk() {
    synchronized (mLock) {
        mLoaded = false;
    }
    new Thread("SharedPreferencesImpl-load") {
```

```
    public void run() {
        loadFromDisk();
    }
}.start();
}
```

可以看到，首先在加锁的情况下对 `mLoaded` 进行了修改，之后则开了个名为「`SharedPreferencesImpl-load`」的线程来加载数据。

我们看到 `loadFromDisk` 方法：

```
private void loadFromDisk() {
    synchronized (mLock) { // 1
        if (mLoaded) {
            return;
        }
        if (mBackupFile.exists()) {
            mFile.delete();
            mBackupFile.renameTo(mFile);
        }
    }
    // Debugging
    if (mFile.exists() && !mFile.canRead()) {
        Log.w(TAG, "Attempt to read preferences file " + mFile + " without permission");
    }
    Map map = null;
    StructStat stat = null;
    try { // 2
        stat = Os.stat(mFile.getPath());
        if (mFile.canRead()) {
            BufferedInputStream str = null;
            try {
                str = new BufferedInputStream(
                    new FileInputStream(mFile), 16*1024);
                map = XmlUtils.readMapXml(str);
            } catch (Exception e) {
                Log.w(TAG, "Cannot read " + mFile.getAbsolutePath(), e);
            } finally {
                IoUtils.closeQuietly(str);
            }
        }
    }
```

```
    } catch (ErrnoException e) {
        /* ignore */
    }
    synchronized (mLock) {      // 3
        mLoaded = true;
        if (map != null) {
            mMap = map;
            mStatTimestamp = stat.st_mtim;
            mStatSize = stat.st_size;
        } else {
            mMap = new HashMap<>();
        }
        mLock.notifyAll();
    }
}
```

代码比较长，我们慢慢分析

首先在注释 1 处，如果已经加载过，则不再进行加载，之后又开始判断是否存在备份文件，若存在则直接将备份文件直接修改为数据文件 \${name}.xml。

之后在注释 2 处，通过 XmlUtils 将 xml 中的数据读取为一个 Map。由于本文主要是对 SP 的大致流程的解读，因此关于 XML 文件的具体读取部分，有兴趣的读者可以自己进入源码研究。

之后在注释 3 处，进行了一些收尾处理，将 mLoaded 置为 true，并对 mMap 进行了判空处理，以保证在 xml 没有数据的情况下其仍不为 null，最后释放了这个读取的锁，表示读取成功。

编辑 SharedPreferences

我们都知道，真正对 SP 的操作其实都是在 Editor 中的，它其实是一个接口，具体的实现类为 EditorImpl。让我们先看看 Editor 的获取：

获取 Editor

看到 SharedPreferencesImpl 的 edit 方法:

```
public Editor edit() {
    // TODO: remove the need to call awaitLoadedLocked() when
    // requesting an editor. will require some work on the
    // Editor, but then we should be able to do:
    //
    //     context.getSharedPreferences(..).edit().putString(..).apply()
    //
    // ... all without blocking.
    synchronized (mLock) {
        awaitLoadedLocked();
    }
    return new EditorImpl();
}
```

可以看到，这里首先先调用了 awaitLoadedLocked() 方法来等待读取的完成，当读取完成后才会真正创建并返回 EditorImpl 对象。

等待读取机制

由于读取过程是一个异步的过程，很有可能导致读取还没结束，我们就开始编辑，因此这里用到了一个 awaitLoadedLocked 方法来阻塞线程，直到读取过程完成，下面我们可以先看看 awaitLoadedLocked 方法:

```
private void awaitLoadedLocked() {
    if (!mLoaded) {
        // Raise an explicit StrictMode onReadFromDisk for the
        // thread, since the real read will be in a different
        // thread and otherwise ignored by StrictMode.
        BlockGuard.getThreadPolicy().onReadFromDisk();
    }
    while (!mLoaded) {
        try {
            mLock.wait();
        }
```

```
        } catch (InterruptedException unused) {
        }
    }
}
```

可以看到，这里会阻塞直到 `mLoaded` 为 `true`，这样就保证了该方法后的方法都会在读取操作进行后执行。

EditorImpl

前面我们提到了 `Edit` 的真正实现类是 `EditorImpl`，它其实是 SPI 的一个内部类。它内部维护了一个 `Map<String, Object>`，通过 `mModified` 来存放对 SP 进行的操作，此时还不会提交到 SPI 中的 `mMap`，我们做的操作都是在改变 `mModified`。

下面列出一些 `EditorImpl` 对外提供的修改接口，其实都是在对 `mModified` 这个 `Map` 进行修改，具体代码就不再讲解，比较简单：

```
public Editor putString(String key, @Nullable String val
    synchronized (mLock) {
        mModified.put(key, value);
        return this;
    }
}

public Editor putStringSet(String key, @Nullable Set<Str
    synchronized (mLock) {
        mModified.put(key,
            (values == null) ? null : new HashSet<St
        return this;
    }
}

public Editor.putInt(String key, int value) {
    synchronized (mLock) {
        mModified.put(key, value);
        return this;
    }
}
```

```
}

public Editor putLong(String key, long value) {
    synchronized (mLock) {
        mModified.put(key, value);
        return this;
    }
}

public Editor putFloat(String key, float value) {
    synchronized (mLock) {
        mModified.put(key, value);
        return this;
    }
}

public Editor putBoolean(String key, boolean value) {
    synchronized (mLock) {
        mModified.put(key, value);
        return this;
    }
}

public Editor remove(String key) {
    synchronized (mLock) {
        mModified.put(key, this);
        return this;
    }
}

public Editor clear() {
    synchronized (mLock) {
        mClear = true;
        return this;
    }
}
```

提交 SharedPreferences

提交本来可以放到编辑中的，但因为它才是重头戏，因此我们单独拎出来讲一下。

我们都知道 SP 的提交有两种方式——apply 和 commit。下面我们来分别分析：

apply

```

public void apply() {
    final long startTime = System.currentTimeMillis();
    final MemoryCommitResult mcr = commitToMemory();      // 1
    final Runnable awaitCommit = new Runnable() {
        public void run() {
            try {
                mcr.writtenToDiskLatch.await();
            } catch (InterruptedException ignored) {
            }
            if (DEBUG && mcr.wasWritten) {
                Log.d(TAG, mFile.getName() + ":" + mcr.memoryStateGeneration
                    + " applied after " + (System.currentTimeMillis() - s
                    tartTime)
                    + " ms");
            }
        }
    };
    QueuedWork.addFinisher(awaitCommit);
    Runnable postWriteRunnable = new Runnable() {
        public void run() {
            awaitCommit.run();
            QueuedWork.removeFinisher(awaitCommit);
        }
    };
    SharedPreferencesImpl.this.enqueueDiskWrite(mcr, postWriteRunnable);    // 2
    // Okay to notify the listeners before it's hit disk
    // because the listeners should always get the same
    // SharedPreferences instance back, which has the
    // changes reflected in memory.
    notifyListeners(mcr);
}

```

首先看到注释 1 处，调用了 `commitToMemory` 方法，它内部就是将原先读取进来的 `mMap` 与刚刚修改过的 `mModified` 进行合并，并存储于返回的 **MemoryCommitResult mcr** 中。

而在注释 2 处，调用了 `enqueueDiskWrite` 方法，传入了之前构造的 `Runnable` 对象，这里的目的是进行一个异步的写操作。

前面提到的两个方法，我们放到后面分析。

也就是说，**apply** 方法会将数据先提交到内存，再开启一个异步过程来将数据写入硬盘。

commit

```
public boolean commit() {
    long startTime = 0;
    if (DEBUG) {
        startTime = System.currentTimeMillis();
    }
    MemoryCommitResult mcr = commitToMemory();      // 1
    SharedPreferencesImpl.this.enqueueDiskWrite(      // 2
        mcr, null /* sync write on this thread okay */);
    try {
        mcr.writtenToDiskLatch.await();
    } catch (InterruptedException e) {
        return false;
    } finally {
        if (DEBUG) {
            Log.d(TAG, mFile.getName() + ":" + mcr.memoryStateGeneration
                + " committed after " + (System.currentTimeMillis() - startTime)
                + " ms");
        }
    }
    notifyListeners(mcr);
    return mcr.writeToDiskResult;
}
```

看到注释 1 处，可以发现，同样调用了 `commitToMemory` 方法进行了合并。

之后看到 2 处，同样调用了 `enqueueDiskWrite` 方法，不过传入的第二个不再是 `Runnable` 方法。这里提一下，如果 `enqueueDiskWrite` 方法传入的第二个参数为 `null`，则会在当前线程执行写入操作。

也就是说，**commit** 方法会将数据先提交到内存，但之后则是一个同步的过程写入硬盘。

同步数据至内存

下面我们来看看两个方法中都调用了的 `commitToMemory` 的具体实现：

```
private MemoryCommitResult commitToMemory() {
    ...
    synchronized (mLock) {
        boolean changesMade = false;
        if (mClear) {
            if (!mMap.isEmpty()) {
                changesMade = true;
                mMap.clear();
            }
            mClear = false;
        }
        for (Map.Entry<String, Object> e : mModified.entrySet()) {
            String k = e.getKey();
            Object v = e.getValue();
            if (v == this || v == null) {
                if (!mMap.containsKey(k)) {
                    continue;
                }
                mMap.remove(k);
            } else {
                if (mMap.containsKey(k)) {
                    Object existingValue = mMap.get(k);
                    if (existingValue != null && existingValue.equals(v)) {
                        continue;
                    }
                }
                mMap.put(k, v);
            }
            changesMade = true;
            if (hasListeners) {
                keysModified.add(k);
            }
        }
    }
}
```

```

        mModified.clear();
        if (changesMade) {
            mCurrentMemoryStateGeneration++;
        }
        memoryStateGeneration = mCurrentMemoryStateGeneration;
    }

    return new MemoryCommitResult(memoryStateGeneration, keysModified, listeners,
        mapToWriteToDisk);
}

```

可以看到，具体的代码就如同我们之前所说的一样，将 `mMap` 的数据与 `mModified` 的数据进行了整合，之后将 `mModified` 重新清空。最后将合并的数据放入了 `MemoryCommitResult` 中。

写入数据至硬盘

我们同样看到 `apply` 和 `commit` 都调用了的方法 `enqueueDiskWrite`:

```

private void enqueueDiskWrite(final MemoryCommitResult mcr,
                            final Runnable postWriteRunnable) {
    final boolean isFromSyncCommit = (postWriteRunnable == null); // 1
    final Runnable writeToDiskRunnable = new Runnable() {
        public void run() {
            synchronized (mWritingToDiskLock) {
                writeToFile(mcr, isFromSyncCommit);
            }
            synchronized (mLock) {
                mDiskWritesInFlight--;
            }
            if (postWriteRunnable != null) {
                postWriteRunnable.run();
            }
        }
    };
    // Typical #commit() path with fewer allocations, doing a write on
    // the current thread.
    if (isFromSyncCommit) {
        boolean wasEmpty = false;
        synchronized (mLock) {

```

```
        wasEmpty = mDiskWritesInFlight == 1;
    }
    if (wasEmpty) {
        writeToDiskRunnable.run(); // 2
        return;
    }
}
QueuedWork.queue(writeToDiskRunnable, !isFromSyncCommit);
}
```

可以看到 1 处，若第二个 Runnable 为 null 的话，则会将 isFromSyncCommit 置为 true，也就是写入会是一个同步的过程。之后在注释 2 处便进行了同步写入。否则会构造一个 Runnable 来提供给 QueueWork 进行异步写入。

QueueWork 类内部维护了一个 single 线程池，这样可以达到我们异步写入的目的。

而 writeToFile 方法中其实就是又调用了之前的 XmlUtils 来进行 XML 的写入。

总结

SharedPreferences 其实就是一个用使用 XML 进行保存的 K-V 存储库。

在获取 SP 时会进行数据的加载，将 name 对应的 xml 文件以 Map 的形式读入到内存。

而 SP 的编辑操作其实都是在 Editor 内实现，它内部维护了一个新的 Map，所有的编辑操作其实都是在操作这个 Map，只有提交时才会与之前读取的数据进行合并。

其提交分为两种，`apply` 和 `commit`，它们的特性如下

apply

- 会将数据先提交到内存，再开启一个异步过程来将数据写入硬盘。
- 返回值时可能写入操作还没有结束
- 写入失败时不会有任何提示

commit

会将数据先提交到内存，但之后则是一个同步的过程写入硬盘。

写入操作结束后才会返回值

写入失败会有提示

因此，当我们对写入的结果不那么关心的情况下，可以使用 `apply` 进行异步写入，而当我们对写入结果十分关心且提交后有后续操作的话最好使用 `commit` 来进行同步写入。

10. 深入解析 EventBus 源码

之前在自己的博客写过了很多常用的库的源码解析，但都需要大量参考其他大牛的博客才能彻底理解一个库的原理。现在想检验一下自己的代码阅读能力，因此尝试靠自己去独立地完成一篇源码解析，看看效果如何。我们先从 `EventBus` 入手，这篇文章分析的 `EventBus` 版本为 3.1.1。

getDefault 方法

我们先从 `EventBus` 的入口，`getDefalut` 方法入手：

```
public static EventBus getDefault() {
    if (defaultInstance == null) {
        synchronized (EventBus.class) {
            if (defaultInstance == null) {
                defaultInstance = new EventBus();
            }
        }
    }
    return defaultInstance;
}
```

从 getDefault 方法可以看出，EventBus 类是一个采用了 Double Check 的单例类。

我们接下来看到它的构造函数，它的无参构造函数 EventBus() 实际上是调用了 EventBus(EventBusBuilder) 这个有参构造函数的，传入的参数是 DEFAULT_BUILDER，而 DEFAULT_BUILDER 则是一个调用了 EventBusBuilder 默认构造器的对象。可以看出，这里用到了 Builder 模式来支持用 EventBusBuilder 进行一些配置。

下面我们看到 EventBus(EventBusBuilder):

```
EventBus(EventBusBuilder builder) {
    logger = builder.getLogger();
    subscriptionsByEventType = new HashMap<>();
    typesBySubscriber = new HashMap<>();
    stickyEvents = new ConcurrentHashMap<>();
    mainThreadSupport = builder.getMainThreadSupport();
    mainThreadPoster = mainThreadSupport != null ? mainThreadSupport.createPoster(this) : null;
    backgroundPoster = new BackgroundPoster(this);
    asyncPoster = new AsyncPoster(this);
    indexCount = builder.subscriberInfoIndexes != null ? builder.subscriberInfoIndexes.size() : 0;
    subscriberMethodFinder = new SubscriberMethodFinder(builder.subscriberInfoIndexes,
        builder.strictMethodVerification, builder.ignoreGeneratedIndex);
```

```
    logSubscriberExceptions = builder.logSubscriberExceptions;
    logNoSubscriberMessages = builder.logNoSubscriberMessages;
    sendSubscriberExceptionEvent = builder.sendSubscriberExceptionEvent;
    sendNoSubscriberEvent = builder.sendNoSubscriberEvent;
    throwSubscriberException = builder.throwSubscriberException;
    eventInheritance = builder.eventInheritance;
    executorService = builder.executorService;
}
```

这个构造方法中主要进行的是一些容器的初始化以及将一些配置参数从 Builder 中取出。

register 方法

下面我们再从 register 方法的角度进行分析，看看 EventBus 在我们进行 register 时做了一些什么事：

```
public void register(Object subscriber) {
    Class<?> subscriberClass = subscriber.getClass();
    List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscriberMethods(subscriberClass);
    synchronized (this) {
        for (SubscriberMethod subscriberMethod : subscriberMethods) {
            subscribe(subscriber, subscriberMethod);
        }
    }
}
```

register 传入的 register 是一个 Object 类型，也就是任意类都可以向 EventBus 进行 register。它首先获取到了 subscriber 的类型信息，然后将其传递给了 subscriberMethodFinder 的 findSubscriberMethods() 方法。

通过名称可以很容易看出，SubscriberMethodFinder 类是一个专门用来搜寻 subscriber 中含有 [@Subscribe](#) 注解的方法的类。这里进行的操作就是将所有被 [@Subscribe](#) 标记的方法都加入到 List 中。

而在找到了这些方法后，则一个个进行遍历，并将其执行 subscribe 操作。

也就是说 register 可以分为两部分来看——**搜寻及订阅**

搜寻过程

我们先进入 findSubscriberMethods() 看看它的搜寻过程

```
List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {
    List<SubscriberMethod> subscriberMethods = METHOD_CACHE.get(subscriberClasses);
    if (subscriberMethods != null) {
        return subscriberMethods;
    }
    if (ignoreGeneratedIndex) { // 1
        subscriberMethods = findUsingReflection(subscriberClass);
    } else {
        subscriberMethods = findUsingInfo(subscriberClass);
    }
    if (subscriberMethods.isEmpty()) {
        throw new EventBusException("Subscriber " + subscriberClass
            + " and its super classes have no public methods with the @Subscribe annotation");
    } else {
        METHOD_CACHE.put(subscriberClass, subscriberMethods);
        return subscriberMethods;
    }
}
```

可以分别看到 1 和 2 处的 if 语句，在 ignoreGeneratedIndex 为 true 时，调用了 findUsingReflection 来使用反射搜寻方法。反之则调用了 findUsingInfo 方法进行搜寻。ignoreGeneratedIndex 是用于标记是否忽略由 Builder 传入的 SubscriberMethodInfo。

直接搜寻

```
private List<SubscriberMethod> findUsingReflection(Class<?> subscriberClass) {
    FindState findState = prepareFindState();
    findState.initForSubscriber(subscriberClass);
    while (findState.clazz != null) {
        findUsingReflectionInSingleClass(findState);
        findState.moveToSuperclass();
    }
    return getMethodsAndRelease(findState);
}
```

可以看出，搜寻过程的结果是用了一个名为 `FindState` 的类进行存储的，它是 `SubscriberMethodFinder` 的一个内部类。我们重点关注的是这里的 `while` 循环，它先执行了 `findUsingReflectionInSingleClass` 方法通过反射找到所有被 `@Subscribe` 标注的方法，再通过 `moveToSuperclass` 向这个类的父类进行搜寻。

查找

我们看到 `findUsingReflectionInSingleClass` 的实现：

```
private void findUsingReflectionInSingleClass(FindState findState) {
    Method[] methods;
    try {
        // This is faster than getMethods, especially when subscribers are fat
        // classes like Activities
        methods = findState.clazz.getDeclaredMethods();
    } catch (Throwable th) {
        // Workaround for java.lang.NoClassDefFoundError, see https://github.co
        // m/greenrobot/EventBus/issues/149
        methods = findState.clazz.getMethods();
        findState.skipSuperClasses = true;
    }
    for (Method method : methods) {
        int modifiers = method.getModifiers();
        if ((modifiers & Modifier.PUBLIC) != 0 && (modifiers & MODIFIERS_IGNORE)
            == 0) {
            Class<?>[] parameterTypes = method.getParameterTypes();
            if (parameterTypes.length == 1) {
```

```
        Subscribe subscribeAnnotation = method.getAnnotation(Subscribe.class);

        if (subscribeAnnotation != null) {
            Class<?> eventType = parameterTypes[0];
            if (findState.checkAdd(method, eventType)) {
                ThreadMode threadMode = subscribeAnnotation.threadMode();
                findState.subscriberMethods.add(new SubscriberMethod(method, eventType, threadMode,
                        subscribeAnnotation.priority(), subscribeAnnotation.sticky()));
            }
        } else if (strictMethodVerification && method.isAnnotationPresent(Subscribe.class)) {
            String methodName = method.getDeclaringClass().getName() + "." + method.getName();
            throw new EventBusException("@Subscribe method " + methodName +
                    " must have exactly 1 parameter but has " + parameterTypes.length);
        } else if (strictMethodVerification && method.isAnnotationPresent(Subscri
rbe.class)) {
            String methodName = method.getDeclaringClass().getName() + "." + method.getName();
            throw new EventBusException(methodName +
                    " is a illegal @Subscribe method: must be public, non-static, and non-abstract");
        }
    }
}
```

这里的代码比较长，我们慢慢分析。

首先进行的是 Method 列表的获取。这里通过注释可以看出，使用 getDeclaredMethods() 方法其实是比 getMethods() 方法的效率更高的，但有时会导致 NoClassDefFoundError，此时采取备用方案，使用 getMethods() 进行获取。

之后遍历 Method 数组，其中进行了两次校验：第一次校验用于检查被 `@Subscribe` 修饰的方法是否是 public、non-static、non-abstract 的。第二次检查则是检查其参数个数是否符合 1 的要求。当不满足时会抛出对应异常，而满足要求则会进行 `@Subscribe` 注解的搜索。

当找到了对应注解后会进行一次 `checkAdd`，在这个方法中会将 方法及其对应的 Event 放入一个 `HashMap anyMethodByEventType`，同时还会将方法的签名（形式为 方法名>Event 类型名）及对应方法放入另一个 `HashMap subscriberClassByMethodKey`。

当 `checkAdd` 检查没有放入过这个方法及 Event 后，就会将方法的信息包装为一个 `SubscriberMethod` 类，然后放入我们需要的结果列表。

向上查找

当查找完成后，会向调用 `moveToSuperclass` 方法向其父类进行查询，直到遇到了系统提供的库。通过这种向上查找的方式可以使得 `EventBus` 支持继承这一 OOP 特性。

```
void moveToSuperclass() {
    if (skipSuperClasses) {
        clazz = null;
    } else {
        clazz = clazz.getSuperclass();
        String clazzName = clazz.getName();
        /** Skip system classes, this just degrades performance. */
        if (clazzName.startsWith("java.") || clazzName.startsWith("javax.") ||
            clazzName.startsWith("android.")) {
            clazz = null;
        }
    }
}
```

```
    }
}
```

非直接搜寻

下面我们看到搜寻过程的另一种实现：

```
private List<SubscriberMethod> findUsingInfo(Class<?> subscriberClass) {
    FindState findState = prepareFindState();
    findState.initForSubscriber(subscriberClass);
    while (findState.clazz != null) {
        findState.subscriberInfo = getSubscriberInfo(findState);
        if (findState.subscriberInfo != null) {
            SubscriberMethod[] array = findState.subscriberInfo.getSubscriberMethods();
            for (SubscriberMethod subscriberMethod : array) {
                if (findState.checkAdd(subscriberMethod.method, subscriberMethod.eventType)) {
                    findState.subscriberMethods.add(subscriberMethod);
                }
            }
        } else {
            findUsingReflectionInSingleClass(findState);
        }
        findState.moveToSuperclass();
    }
    return getMethodsAndRelease(findState);
}
```

这里可以看到，先通过 `getSubscriberInfo` 获取到了由 `Builder` 过程传入的 `SubscriberInfoIndex`，从中获取需要的信息。当其值为 `null` 时，再使用反射搜寻的方式进行搜寻。

而具体添加过程，则与直接搜寻中的方式类似，这里不再赘述。

FindState 复用

这里要注意，无论是直接搜索还是非直接搜索，它们所使用的 `FindState` 都是通过 `prepareFindState()` 来进行获取，同时，它们的结果最后都会通过 `getMethodsAndRelease(findState)` 来进行返回。其实 `SubscriberMethodFinder` 类中维护了一个 `FindState` 池，是一个默认大小为 4 的数组，在 `prepareFindState` 中会遍历数组找到非 `null` 的 `FindState` 进行返回。而在 `getMethodsAndRelease(findState)` 中则是将搜寻的结果取出后，对 `FindState` 进行 `recycle`，之后再将其放回 `FindState` 池中。这种池的复用思想非常值得我们在设计库的时候学习。

订阅过程

我们回到 `register` 方法中来：

```
public void register(Object subscriber) {
    Class<?> subscriberClass = subscriber.getClass();
    List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscriberMethods(subscriberClass);
    synchronized (this) {
        for (SubscriberMethod subscriberMethod : subscriberMethods) {
            subscribe(subscriber, subscriberMethod);
        }
    }
}
```

可以看到 `EventBus` 为方法的订阅过程进行了加锁，保证了线程安全。然后遍历了每一个被标记的方法，一一将其订阅。

下面我们看看具体的订阅流程，进入 `subscribe` 方法：

```
private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {
    Class<?> eventType = subscriberMethod.eventType;
```

```

Subscription newSubscription = new Subscription(subscriber, subscriberMethod);
CopyOnWriteArrayList<Subscription> subscriptions = subscriptionsByEventType.get(eventType);
if (subscriptions == null) {
    subscriptions = new CopyOnWriteArrayList<>();
    subscriptionsByEventType.put(eventType, subscriptions);
} else {
    if (subscriptions.contains(newSubscription)) {
        throw new EventBusException("Subscriber " + subscriber.getClass() +
" already registered to event "
+ eventType);
    }
}
int size = subscriptions.size();
for (int i = 0; i <= size; i++) {
    if (i == size || subscriberMethod.priority > subscriptions.get(i).subscriberMethod.priority) {
        subscriptions.add(i, newSubscription);
        break;
    }
}
List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber);
if (subscribedEvents == null) {
    subscribedEvents = new ArrayList<>();
    typesBySubscriber.put(subscriber, subscribedEvents);
}
subscribedEvents.add(eventType);
if (subscriberMethod.sticky) {
    if (eventInheritance) {
        // Existing sticky events of all subclasses of eventType have to be
considered.
        // Note: Iterating over all events may be inefficient with lots of s
ticky events,
        // thus data structure should be changed to allow a more efficient L
ookup
        // (e.g. an additional map storing sub classes of super classes: Cl
ass -> List<Class>)
        Set<Map.Entry<Class<?>, Object>> entries = stickyEvents.entrySet();
        for (Map.Entry<Class<?>, Object> entry : entries) {
            Class<?> candidateEventType = entry.getKey();
            if (eventType.isAssignableFrom(candidateEventType)) {
                Object stickyEvent = entry.getValue();

```

```
        checkPostStickyEventToSubscription(newSubscription, stickyEvent);
    }
}
} else {
    Object stickyEvent = stickyEvents.get(eventType);
    checkPostStickyEventToSubscription(newSubscription, stickyEvent);
}
}
}
```

这里的代码也是很长...让我们慢慢进行分析。

先看看 `Subscription`, 它就是一个将 `Subscriber` 及 `SubscriberMethod` 进行包装的类。

首先, 这个方法进行了一个安全检查, 检查了同样的方法是否已经被注册过, 没有则将其放入 `Map` 中, 否则抛出异常。

之后再将其按照优先级放入以该 `Event` 为参数的方法列表中。

然后, 又将这个方法加入了以该 `Subscriber` 为 key, 以它内部所有被 `@Subscribe` 标记的方法的列表为 value 的 `Map typesBySubscriber` 中。

最后, 进行了一个对 `sticky` 事件的特殊处理, 如果为 `sticky` 事件则会在 `register` 时进行一次对 `Method` 的调用。主要逻辑是: 首先判断了 `Event` 是否子 `Event`, 若是一个子 `Event` 则找到其父 `Event` 作为参数 `Event`, 否则将其作为参数 `Event`, 然后在判 `null` 的情况下调用 `postToSubscription` 方法来执行这个方法。关于具体的执行过程, 会在下文中进行讲解。

post 方法

post 方法开始，就会涉及我们的 Event 的执行过程了。这里根据 register 的分析过程可以大概猜出方法应该是由反射的方式被执行的，下面让我们进入 post 方法的源码看看是否是这样：

```
public void post(Object event) {
    PostingThreadState postingState = currentPostingThreadState.get();
    List<Object> eventQueue = postingState.eventQueue;
    eventQueue.add(event);
    if (!postingState.isPosting) {
        postingState.isMainThread = isMainThread();
        postingState.isPosting = true;
        if (postingState.canceled) {
            throw new EventBusException("Internal error. Abort state was not reset");
        }
        try {
            while (!eventQueue.isEmpty()) {
                postSingleEvent(eventQueue.remove(0), postingState);
            }
        } finally {
            postingState.isPosting = false;
            postingState.isMainThread = false;
        }
    }
}
```

存放线程信息

首先，它获取到了 currentPostingThreadState 这一个对象，它的类型是 PostingThreadState，这个类的主要用途是记录事件的发布者的线程信息。

```
final static class PostingThreadState {
    final List<Object> eventQueue = new ArrayList<>();
    boolean isPosting;
    boolean isMainThread;
    Subscription subscription;
    Object event;
    boolean canceled;
```

```
}
```

我们看到 `currentPostingThreadState` 的创建过程：

```
private final ThreadLocal<PostingThreadState> currentPostingThreadState = new  
ThreadLocal<PostingThreadState>() {  
    @Override  
    protected PostingThreadState initialValue() {  
        return new PostingThreadState();  
    }  
};
```

这里可以看到 `currentPostingThreadState` 是通过 `ThreadLocal` 进行保存。

`ThreadLocal` 是一个用于创建线程局部变量的类。它创建的变量只能被当前线程访问，其他线程则无法访问和修改。

之后在 `post` 中进行的操作就是为 `currentPostingThreadState` 填充各种线程相关信息了。

```
List<Object> eventQueue = postingState.eventQueue;  
eventQueue.add(event);  
if (!postingState.isPosting) {  
    postingState.isMainThread = isMainThread();  
    postingState.isPosting = true;  
    if (postingState.canceled) {  
        throw new EventBusException("Internal error. Abort state was not reset  
    ");  
    }  
    try {  
        while (!eventQueue.isEmpty()) {  
            postSingleEvent(eventQueue.remove(0), postingState);  
        }  
    } finally {  
        postingState.isPosting = false;  
        postingState.isMainThread = false;  
    }  
}
```

我们回到 `post` 中，可以发现，在 `currentPostingThreadState` 中维护了一个 `eventQueue`。

首先，将 Event 插入了 eventQueue 中，之后将 isMainThread 等信息进行填充。同时将 postingState 的 isPosting 置为了 true，使得事件 post 的过程中当前线程的其他 post 事件无法被相应，当 post 过程结束后，再将其置为 true。

之后，便开始遍历 eventQueue，将事件一个个出队并执行 postSingleEvent 方法，接下来我们看到 postSingleEvent 方法：

```
private void postSingleEvent(Object event, PostingThreadState postingState) throws Error {
    Class<?> eventClass = event.getClass();
    boolean subscriptionFound = false;
    if (eventInheritance) {
        List<Class<?>> eventTypes = lookupAllEventTypes(eventClass);
        int countTypes = eventTypes.size();
        for (int h = 0; h < countTypes; h++) {
            Class<?> clazz = eventTypes.get(h);
            subscriptionFound |= postSingleEventForEventType(event, postingState, clazz);
        }
    } else {
        subscriptionFound = postSingleEventForEventType(event, postingState, eventClass);
    }
    if (!subscriptionFound) {
        if (logNoSubscriberMessages) {
            logger.log(Level.FINE, "No subscribers registered for event " + eventClass);
        }
        if (sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class &&
            eventClass != SubscriberExceptionEvent.class) {
            post(new NoSubscriberEvent(this, event));
        }
    }
}
```

可以看到，这里通过 `postSingleEventForEventType` 来进行搜寻对应 `Subscription`，如果 `Event` 是子 `Event`，则获取它的所有父 `Event` 列表，再遍历列表进行搜寻。否则直接调用 `postSingleEventForEventType` 进行搜寻。

搜寻 `Subscription`

下面我们看到 `postSingleEventForEventType`:

```
private boolean postSingleEventForEventType(Object event, PostingThreadState postingState, Class<?> eventClass) {
    CopyOnWriteArrayList<Subscription> subscriptions;
    synchronized (this) {
        subscriptions = subscriptionsByEventType.get(eventClass);
    }
    if (subscriptions != null && !subscriptions.isEmpty()) {
        for (Subscription subscription : subscriptions) {
            postingState.event = event;
            postingState.subscription = subscription;
            boolean aborted = false;
            try {
                postToSubscription(subscription, event, postingState.isMainThread);
                aborted = postingState.canceled;
            } finally {
                postingState.event = null;
                postingState.subscription = null;
                postingState.canceled = false;
            }
            if (aborted) {
                break;
            }
        }
        return true;
    }
    return false;
}
```

首先，这里根据 event 找到了所有对应的 Subscription，然后遍历 subscription 列表调用 postToSubscription() 方法，这个方法在之前 register 的分析中针对 sticky 事件的代码中也有调用，它的作用是调用 event 对应的方法。

未找到对应 Subscription 的处理

我们在这里先不关心 postToSubscription 的具体实现，先看看在 Subscription 调用结束后做了什么事，我们回到 postSingleEvent 方法中：

```
if (eventInheritance) {
    List<Class<?>> eventTypes = lookupAllEventTypes(eventClass);
    int countTypes = eventTypes.size();
    for (int h = 0; h < countTypes; h++) {
        Class<?> clazz = eventTypes.get(h);
        subscriptionFound |= postSingleEventForEventType(event, postingState,
                clazz);
    }
} else {
    subscriptionFound = postSingleEventForEventType(event, postingState, event
            class);
}
if (!subscriptionFound) {
    if (logNoSubscriberMessages) {
        logger.log(Level.FINE, "No subscribers registered for event " + eventCl
            ass);
    }
    if (sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class &&
            eventClass != SubscriberExceptionEvent.class) {
        post(new NoSubscriberEvent(this, event));
    }
}
```

可以看到，如果在没有找到对应的 subscription，则会创建一个 NoSubscriberEvent 再调用 post 请求。这样，无论 Event 是否有 Subscriber，它都会进行一次检测。

执行对应 Subscription

下面我们看到 postToSubscription, 看看是如何调用 Event 对应的方法的:

```
private void postToSubscription(Subscription subscription, Object event, boolean isMainThread) {
    switch (subscription.subscriberMethod.threadMode) {
        case POSTING:
            invokeSubscriber(subscription, event);
            break;
        case MAIN:
            if (isMainThread) {
                invokeSubscriber(subscription, event);
            } else {
                mainThreadPoster.enqueue(subscription, event);
            }
            break;
        case MAIN_ORDERED:
            if (mainThreadPoster != null) {
                mainThreadPoster.enqueue(subscription, event);
            } else {
                // temporary: technically not correct as poster not decoupled from subscriber
                invokeSubscriber(subscription, event);
            }
            break;
        case BACKGROUND:
            if (isMainThread) {
                backgroundPoster.enqueue(subscription, event);
            } else {
                invokeSubscriber(subscription, event);
            }
            break;
        case ASYNC:
            asyncPoster.enqueue(subscription, event);
            break;
        default:
            throw new IllegalStateException("Unknown thread mode: " + subscription.subscriberMethod.threadMode);
    }
}
```

可以看到，这里对订阅者的线程进行了判断，采用不同的 Poster 进行入队操作，采用队列的方式一一处理。对某些特殊情况则直接调用 invokeSubscriber(subscription, event) 进行处理。关于 Poster 的设计我们在之后进行分析，在 Poster 内部调用的是 invokeSubscriber(PendingPost) 方法。

invokeSubscriber(subscription, event)

我们先看到 invokeSubscriber(subscription, event) 方法：

```
void invokeSubscriber(Subscription subscription, Object event) {
    try {
        subscription.subscriberMethod.method.invoke(subscription.subscriber, event);
    } catch (InvocationTargetException e) {
        handleSubscriberException(subscription, event, e.getCause());
    } catch (IllegalAccessException e) {
        throw new IllegalStateException("Unexpected exception", e);
    }
}
```

果然，如我们之前猜测的一样，它是通过反射来对方法进行调用的，代码很简单，就不再赘述了。

invokeSubscriber(PendingPost)

下面我们看到 invokeSubscriber(PendingPost) 方法：

```
void invokeSubscriber(PendingPost pendingPost) {
    Object event = pendingPost.event;
    Subscription subscription = pendingPost.subscription;
    PendingPost.releasePendingPost(pendingPost);
    if (subscription.active) {
        invokeSubscriber(subscription, event);
    }
}
```

可以看到，它在内部判断了 Subscription 的 active 状态，如果为 active，再调用 invokeSubscriber(subscription, event) 方法对 Method 进行执行。

这个 active 状态可以使得 unregister 的类中的对应 Method 不再被执行。

postSticky 方法

这时你可能会想到，还有一个方法 postSticky 呢，我们接下来看到 postSticky 方法：

```
public void postSticky(Object event) {
    synchronized (stickyEvents) {
        stickyEvents.put(event.getClass(), event);
    }
    // Should be posted after it is putted, in case the subscriber wants to remove immediately
    post(event);
}
```

这里的逻辑十分简单，先将 event 加入 stickyEvents 列表，再执行 post 请求。

unregister 方法

我们接着看到 unregister 方法：

```
public synchronized void unregister(Object subscriber) {
    List<Class<?>> subscribedTypes = typesBySubscriber.get(subscriber);
    if (subscribedTypes != null) {
        for (Class<?> eventType : subscribedTypes) {
            unsubscribeByEventType(subscriber, eventType);
        }
        typesBySubscriber.remove(subscriber);
    } else {
        logger.log(Level.WARNING, "Subscriber to unregister was not registered before: " + subscriber.getClass());
```

```
    }  
}
```

unregister 方法相对比较简单，首先它先判断了这个 Event 是否还未注册，若已注册则遍历在 typesBySubscriber 中 Subscriber 所对应的每一个 EventType，并一一对其执行 unsubscribeByEventType 方法取消订阅，同时 Subscriber 所对应的信息从 typesBySubscriber 这个 Map 中移除。

下面我们看看 unsubscribeByEventType 中具体做了什么：

```
private void unsubscribeByEventType(Object subscriber, Class<?> eventType) {  
    List<Subscription> subscriptions = subscriptionsByEventType.get(eventType);  
    if (subscriptions != null) {  
        int size = subscriptions.size();  
        for (int i = 0; i < size; i++) {  
            Subscription subscription = subscriptions.get(i);  
            if (subscription.subscriber == subscriber) {  
                subscription.active = false;  
                subscriptions.remove(i);  
                i--;  
                size--;  
            }  
        }  
    }  
}
```

可以看到，它是将 subscriptionsByEventType 中与 EventType 对应的 Subscription 列表取出，遍历并将该 Subscriber 对应的 Subscription 的 active 标记为 false 并删除。

Poster

前面提到了 post 方法中会用到 Poster 来进行方法的按队列执行，EventBus 中定义了一种 Poster 这个接口，用于处理 post 后方法执行的调度。

```
/**  
 * Posts events.  
 *  
 * @author William Ferguson  
 */  
interface Poster {  
    /**  
     * Enqueue an event to be posted for a particular subscription  
     *  
     * @param subscription Subscription which will receive the eve  
     * @param event Event that will be posted to subscriber  
     */  
    void enqueue(Subscription subscription, Object event);  
}
```

可以看到，它内部只有一个方法 enqueue，用于将 Event 放入待定队列。

在 EventBus 中的 Poster 用到了三种，我们分别介绍：

HandlerPoster

HandlerPoster 是默认情况下 mainThreadPoster 的类型，它的内部是使用 Handler 实现进程的调度，主要关注其 handleMessage 的实现：

```
@Override  
public void handleMessage(Message msg) {  
    boolean rescheduled = false;  
    try {  
        long started = SystemClock.uptimeMillis();  
        while (true) {  
            PendingPost pendingPost = queue.poll();  
            if (pendingPost == null) {  
                synchronized (this) {  
                    // Check again, this time in synchronized
```

```

        pendingPost = queue.poll();
        if (pendingPost == null) {
            handlerActive = false;
            return;
        }
    }
    eventBus.invokeSubscriber(pendingPost);
    long timeInMethod = SystemClock.uptimeMillis() - started;
    if (timeInMethod >= maxMillisInsideHandleMessage) {
        if (!sendMessage(obtainMessage())) {
            throw new EventBusException("Could not send handler message");
        }
        rescheduled = true;
        return;
    }
}
} finally {
    handlerActive = rescheduled;
}
}

```

可以看到，内部是通过一个死循环遍历 PendingPost 的队列，分别对其执行 invokeSubscriber。

AsyncPoster

AsyncPoster 对应了 EventBus 中的 asyncPoster，下面是它的代码：

```

class AsyncPoster implements Runnable, Poster {
    private final PendingPostQueue queue;
    private final EventBus eventBus;
    AsyncPoster(EventBus eventBus) {
        this.eventBus = eventBus;
        queue = new PendingPostQueue();
    }
    public void enqueue(Subscription subscription, Object event) {
        PendingPost pendingPost = PendingPost.obtainPendingPost(subscription,
        event);
        queue.enqueue(pendingPost);
    }
}

```

```
        eventBus.getExecutorService().execute(this);
    }
    @Override
    public void run() {
        PendingPost pendingPost = queue.poll();
        if(pendingPost == null) {
            throw new IllegalStateException("No pending post available");
        }
        eventBus.invokeSubscriber(pendingPost);
    }
}
```

它的代码很简单，通过 EventBus 的线程池中取出一个线程并在该线程中调用 invokeSubscriber 方法。

BackgroundPoster

BackgroundPoster 对应 EventBus 中的 backgroundPoster，下面是它的代码：

```
final class BackgroundPoster implements Runnable, Poster {
    private final PendingPostQueue queue;
    private final EventBus eventBus;
    private volatile boolean executorRunning;
    BackgroundPoster(EventBus eventBus) {
        this.eventBus = eventBus;
        queue = new PendingPostQueue();
    }
    public void enqueue(Subscription subscription, Object event) {
        PendingPost pendingPost = PendingPost.obtainPendingPost(subscription,
event);
        synchronized (this) {
            queue.enqueue(pendingPost);
            if (!executorRunning) {
                executorRunning = true;
                eventBus.getExecutorService().execute(this);
            }
        }
    }
    @Override
    public void run() {
        try {
```

```

        try {
            while (true) {
                PendingPost pendingPost = queue.poll(1000);
                if (pendingPost == null) {
                    synchronized (this) {
                        // Check again, this time in synchronized
                        pendingPost = queue.poll();
                        if (pendingPost == null) {
                            executorRunning = false;
                            return;
                        }
                    }
                }
                eventBus.invokeSubscriber(pendingPost);
            }
        } catch (InterruptedException e) {
            eventBus.getLogger().log(Level.WARNING, Thread.currentThread().
                getName() + " was interrupted", e);
        }
    } finally {
        executorRunning = false;
    }
}
}

```

这段代码比较长，和 AsyncPoster 有些类似，它使用了 synchronized 以保证线程安全，在 run 方法中遍历了 PendingPost 列表，取出并执行 Event。

PendingPost 池

PendingPost 内部也是维护了一个 PendingPost 池的，上面的几个 Poster 都是调用 PendingPost.obtainPendingPost 方法从其中取出 PendingPost，它的代码如下：

```

final class PendingPost {
    private final static List<PendingPost> pendingPostPool = new ArrayList<PendingPost>();
    Object event;
    Subscription subscription;
}

```

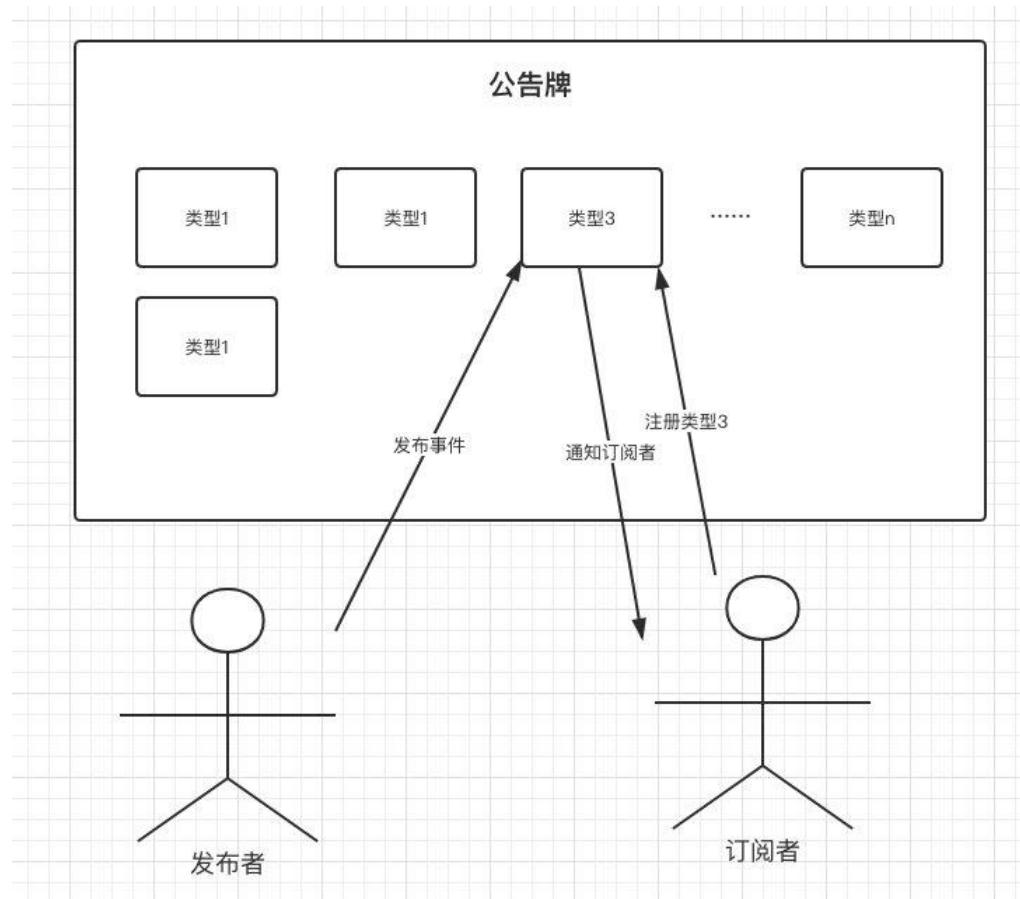
```
PendingPost next;
private PendingPost(Object event, Subscription subscription) {
    this.event = event;
    this.subscription = subscription;
}
static PendingPost obtainPendingPost(Subscription subscription, Object eve
nt) {
    synchronized (pendingPostPool) {
        int size = pendingPostPool.size();
        if (size > 0) {
            PendingPost pendingPost = pendingPostPool.remove(size - 1);
            pendingPost.event = event;
            pendingPost.subscription = subscription;
            pendingPost.next = null;
            return pendingPost;
        }
    }
    return new PendingPost(event, subscription);
}
static void releasePendingPost(PendingPost pendingPost) {
    pendingPost.event = null;
    pendingPost.subscription = null;
    pendingPost.next = null;
    synchronized (pendingPostPool) {
        // Don't Let the pool grow indefinitely
        if (pendingPostPool.size() < 10000) {
            pendingPostPool.add(pendingPost);
        }
    }
}
```

原理比较简单，就不再分析了。

总结

通过这次源码解析，收获了很多，果然还是自己去看源码才能对这些库有更深层的了解。

EventBus 实际上就是一个基于反射实现的『公告牌』，发布者可以将各种类型的公告放置到公告牌中，而订阅者可以订阅某种类型的公告，当公告牌上出现了这种类型的公告时，便会从上面取出并通知订阅者。



11. Android 自定义注解初探

由于之前用到的很多开源框架如 GreenDao、EventBus、ButterKnife、ARouter 等都使用了自定义注解，因此发觉自己有必要去研究一下自定义注解了，于是写下此篇文章。

什么是注解

首先，要明白什么是注解。

An annotation is a form of metadata, that can be added to Java source code. Classes, methods, variables, parameters and packages may be annotated. Annotations have no direct effect on the operation of the code they annotate.

注解是一种元数据，能添加到 Java 的代码中。类、方法、变量、参数、包都是可以添加注解的，注解对注解的代码并不会有直接的影响，仅仅是一个标记。之所以它能产生作用是因为对它解析之后做了一些处理，比如 ButterKnife 就是在解析后自动为我们生成了一些代码。

注解是一个非常实用的工具，它有如下的作用：

- 降低项目的耦合度。
- 自动完成一些规律性的代码。
- 自动生成 Java 代码，减轻开发者的工作量。

元注解

Java 中有一些常用的注解如我们很熟悉的 `Override`、`SuppressWarnings` 等等，

我们可以从我们最常用的 `@Override` 入手，首先我们打开它的源码：

```
•  
@Target(ElementType.METHOD)  
•  
•  
@Retention(RetentionPolicy.SOURCE)  
•  
•  
public @interface Override {  
•  
•  
}  
•
```

可以看到，`Override` 的源码非常简单，它上方有两个注解 `@Target` 以

及 `@Retention`。它们两个就是所谓的元注解。而 `@interface` 则是定义注解的
关键字。

元注解，简单点说就是用来定义注解的注解。用于定义注解的作用范围，在什么
元素上面等等信息。

元注解共有四种

- **@Retention:** 保留的范围，默认值为 `CLASS`，有下面三个值
 - **SOURCE:** 只在源码中可用

- CLASS: 只在源码和字节码中可用
 - RUNTIME: 源码、字节码、运行时都可用
-
- **@Target:** 用于表示可修饰哪些元素，有下面的几种值
 - TYPE: 类、接口、枚举、注解类型。
 - FIELD: 类成员（构造方法、方法、成员变量）。
 - METHOD: 方法。
 - PARAMETER: 参数。
 - CONSTRUCTOR: 构造器。
 - LOCAL_VARIABLE: 局部变量。
 - ANNOTATION_TYPE: 注解。

- PACKAGE: 包声明。
 - TYPE_PARAMETER: 类型参数。
 - TYPE_USE: 类型使用声明。
-
- **@Inherited**: 表示是否可以被继承, 默认 false
 - **@Documented**: 是否会添加到 Javadoc 文档中

`@Retention` 是定义保留策略, 决定了我们用何种方式对注解进行解析。

其中 `SOURCE` 级别是用于标记的, 而我们在真正使用时用得最多的往往是 `CLASS`(编译时) 和 `RUNTIME`(运行时)。

自定义注解

下面我们尝试实现一个自定义注解 `@MyAnnotation`



```
@Retention(RetentionPolicy.RUNTIME)
•
•
@Target(ElementType.FIELD)
•
•
public @interface MyAnnotation {
•
•
    String value() default "NotExpectError";
•
}
•
```

上面可以看到，这个注解是一个运行期，用于修饰 FIELD 的注解。

而需要解释的是注解内部，定义了两个注解值。

注解值的写法如下：

```
类型 参数名() [default 默认值];
```

当参数名为 value 的时候，我们只需如 `@MyAnnotation("AAA")` 这样使用即可。

但当参数名不是 value 的时候，我们则需要如 `@MyAnnotation(data = "AAA")` 这样使用。

运行时注解的处理

@Retention 的值为 RetentionPolicy.RUNTIME 的时候，它会保留到运行期，此时我们就可以用反射来解析注解：

```
•  
public class AnnotationDemo {  
    •  
    •  
    •  
    •  
    •  
    @MyAnnotation("AnnotationTest")  
    •  
    •  
    private String testStr;  
    •  
    •  
    @MyAnnotation()  
    •  
    •  
    private String testStr2;  
    •  
    •  
    •  
    public static void main(String[] args) {  
        •  
        •  
        try {  
            •  
            •  
            // 获取要解析的类  
            •  
            •  
            Class cls = Class.forName("AnnotationDemo");  
        •  
        •
```

```
// 获取所有 Field
•
•
Field[] declaredFields = cls.getDeclaredFields();
•
•
for(Field field : declaredFields){
•
•
// 获取 Field 上的注解
•
•
MyAnnotation annotation = field.getAnnotation(MyAnnotation.class);
•
•
if(annotation != null){
•
•
// 获取注解值
•
•
String value = annotation.value();
•
•
System.out.println(value);
•
•
}
•
•
}
•
•
}
•
•
} catch (ClassNotFoundException e) {
•
•
e.printStackTrace();
•
•
}
•
•
}
```

```
•  
•  
}  
•  
•  
}  
•
```

我们看到打印结果：

```
AnnotationTest  
N0tExpectErr0r
```

可以看到，我们指定 `value` 的值的时候，它会使用我们所指定的值。而当我们没有指定 `value` 的值的时候，它就会使用 `value` 的默认值 "N0tExpectErr0r"。

除了 `Field`，其他的如类、方法的注解也是这样进行定义。

编译时注解的处理

其实类似 `ButterKnife` 的功能可以用上面这种-反射的方法在运行时实现，比如下面这段 `Demo`：

```
private void getAllAnnotationView() {  
    // 获得成员变量  
    Field[] fields = this.getClass().getDeclaredFields();  
    for (Field field : fields) {
```

```
try {
    // 判断注解
    if (field.getAnnotations() != null) {
        // 确定注解类型
        if (field.isAnnotationPresent(BindView.class)) {
            // 允许修改反射属性
            field.setAccessible(true);
            BindView bindView = field.getAnnotation(BindView.class);

            // findViewById 将注解的 id, 找到 View 注入成员变量中
            field.set(this, findViewById(BindView.value()));
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
```

虽然上述代码可以帮助我们实现类似 `ButterKnife` 的效果，但实际上 `ButterKnife` 并不是这样子做的。

为什么呢？因为反射非常影响程序的效率。

`ButterKnife` 实际上是通过编译时注解，在编译的时候生成对应的 Java 代码，从而实现注入。

```
@Retention(CLASS)
@Target(FIELD)
public @interface BindView{
    @IdRes int value();
}
```

提到编译时注解，我们就需要提到注解处理器(Annotation Processor)。它是 `javac` 的一个工具，用于在编译时扫描和处理注解。

要自定义一个注解处理器，我们需要至少重写四个方法，并且注册自定义的 Processor。

- **@AutoService(Processor.class)**: 谷歌提供的自动注册注解，生成注册 Processor 所需要的格式文件（`com.google.auto` 相关包）。
- **init(ProcessingEnvironment env)**: 初始化处理器，一般在这里获取我们需要的工具类。
- **getSupportedAnnotationTypes()**: 指定注解处理器是注册给哪个注解的，返回指定支持的注解类集合。
- **getSupportedSourceVersion()**: 指定 Java 版本。
- **process()**: 处理器实际处理逻辑入口。

比如我们下面的这个 `CustomProcessor` 的例子：

```
@AutoService(Processor.class)
public class CustomProcessor extends AbstractProcessor {
```

```
/*
 * 注解处理器的初始化
 * 一般在这里获取我们需要的工具类
 * @param processingEnvironment 提供工具类Elements, Types 和Filer
 */
@Override
public synchronized void init(ProcessingEnvironment env){
    super.init(env);
    // Element 代表程序的元素，例如包、类、方法。
    mElementUtils = env.getElementUtils();
    // 处理TypeMirror 的工具类，用于取类信息
    mTypeUtils = env.getTypeUtils();
    // Filer 可以创建文件
    mFiler = env.getFiler();
    // 错误处理工具
    mMessages = env.getMessager();
}

/*
 * 处理器实际处理逻辑入口
 * @param set
 * @param roundEnvironment 所有注解的集合
 * @return
 */
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment env) {
    // 处理
}

// 指定注解处理器是注册给哪个注解的，返回指定支持的注解类集合。
@Override
public Set<String> getSupportedAnnotationTypes() {
    Set<String> sets = new LinkedHashSet<String>();
    // 大部分class 的getName、getCanonicalName 这两个方法没有什么不同。
    // 但是对于array 或内部类等就不同了。
    // getName 返回的是[[Ljava.lang.String 之类的表现形式,
    // getCanonicalName 返回的就是跟我们声明类似的形式。
    sets(BindView.class.getCanonicalName());
    return sets;
}

// 指定Java 版本，一般返回最新版本即可
@Override
public SourceVersion getSupportedSourceVersion() {
    return SourceVersion.latestSupported();
}
```

```
}
```

一般处理器的处理逻辑如下：

1. 遍历得到源码中需要解析的元素列表。
2. 判断元素是否可见和符合要求。
3. 组织数据结构得到输出类参数。
4. 输入生成 java 文件。
5. 处理错误。

Processor 处理过程中，会扫描全部 Java 源码，代码的每一个部分都是一个特定类型的 Element，像是 XML 一层的层级关系一般。如类、变量、方法等，每个 Element 代表一个静态的、语言级别的构件。

其中，Element 代表的是源代码，而 TypeElement 代表的是源代码中的类型元素，例如类。但 TypeElement 并不包含类本身的信息。你可以从 TypeElement 中获取类的名字，但是你获取不到类的其他信息，如它的父类这种信息需要通过 TypeMirror 获取。可以通过调用 elements.asType() 获取元素的 TypeMirror。

了解了 Element 之后，我们便能通过 process 中的 RoundEnvironment 去获取扫描到的所有元素，如下面的代码，通过 env.getElementsAnnotatedWith 方法，获取被@BindView 注解的元素的列表，用 validateElement`校验元素是否可用。

```
@Override
public boolean process(Set<? extends TypeElement> elements, RoundEnvironment env) {
    Map<TypeElement, BindingSet> bindingMap = findAndParseTargets(env);
    for (Element element : env.getElementsAnnotatedWith(BindView.class)) {
        if (!SuperficialValidation.validateElement(element))
            continue;
        try {
            parseResourceArray(element, builderMap, erasedTargetNames);
        } catch (Exception e) {
            logParsingError(element, BindArray.class, e);
        }
    }
}
```

由于 env.getElementsAnnotatedWith 返回的是所有被注解了@BindView 的元素。所以有时候我们还需要一些额外的判断，比如检查这些 Element 是否是一个类。

javapoet (com.squareup:javapoet) 是一个根据指定参数，生成 java 文件的开源库，在处理器中按照参数创建出 JavaFile 之后，通过 Filer 利用 javaFile.writeTo(filer); 就可以生成需要的 java 文件。

在处理器中我们不能直接抛出一个异常，因为在 process() 中抛出一个异常会导致运行注解处理器的 JVM 崩溃。因此，注解处理器有一个 Messager 类，一般

通过 `messager.printMessage(Diagnostic.Kind.ERROR, StringMessage, element)` 即可正常输出错误信息。

在 Android 中使用自定义注解

这里写了一个 Demo，创建了一个 ViewAnnotation 的 Java Library。

gradle 配置

首先，在这个 Module 的 `build.gradle` 中 implementation 了对应的库

```
apply plugin: 'java-library'
sourceCompatibility = "7"
targetCompatibility = "7"
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.google.auto.service:auto-service:1.0-rc4'
    implementation 'com.squareup:javapoet:1.11.1'
}
```

定义注解

然后，我定义了两个注解 `DIActivity` 及 `DIView`

DIActivity 标明了要绑定 View 的 id 的 Activity

DIView 则用于指定控件对应的 id

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.CLASS)
public @interface DIActivity{
}

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.CLASS)
public @interface DIView {
    int value() default 0;
}
```

实现注解处理器

最后，我自定义了一个名为 DIProcessor 的类用于解析 Annotation 并生成对应 java 文件

```
@AutoService(Processor.class)
public class DIProcessor extends AbstractProcessor {
    private Elements elementUtils;
    @Override
    public Set<String> getSupportedAnnotationTypes() {
        // 规定需要处理的注解
        return Collections.singleton(DIActivity.class.getCanonicalName());
    }
    @Override
    public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment) {
        System.out.println("DIProcessor");
        Set<? extends Element> elements = roundEnvironment.getElementsAnnotatedWith(DIActivity.class);
```

```
        for (Element element : elements) {
            // 判断是否Class
            TypeElement typeElement = (TypeElement) element;
            List<? extends Element> members = elementUtils.getAllMembers(typeElement);
            MethodSpec.Builder bindViewMethodSpecBuilder = MethodSpec.methodBuilder("bindView")
                .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
                .returns(TypeName.VOID)
                .addParameter(ClassName.get(typeElement.asType()), "activity");
            for (Element item : members) {
                DIView diView = item.getAnnotation(DIView.class);
                if (diView == null){
                    continue;
                }
                bindViewMethodSpecBuilder.addStatement(String.format("activity.%s = (%s) activity.findViewById(%s)", item.getSimpleName(), ClassName.get(item.asType()).toString(), diView.value()));
            }
            TypeSpec typeSpec = TypeSpec.classBuilder("DI" + element.getSimpleName())
                .superclass(TypeName.get(typeElement.asType()))
                .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
                .addMethod(bindViewMethodSpecBuilder.build())
                .build();
            JavaFile javaFile = JavaFile.builder(getPackageName(typeElement), typeSpec).build();
            try {
                javaFile.writeTo(processEnv.getFiler());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return true;
    }

    private String getPackageName(TypeElement type) {
        return elementUtils.getPackageOf(type).getQualifiedName().toString();
    }

    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        elementUtils = processingEnv.getElementUtils();
    }
}
```

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

可以看到，在生成的 java 文件里面我创建了一个名为 bindView 的方法，它是 public void static 的，并且参数是@DIActivity 这个注解所标识的 Activity 的类名。之后，在 bindView 方法中添加了所有被 @DIView 标识的 View 的 findViewById 代码。

最后，将这个 bindView 方法放入了一个名为 DI+ 对应 Activity 名称的 final 类中，然后通过 JavaFile.builder 方法创建了对应的 JavaFile，然后调用 JavaFile::writeTo 方法创建对应的 Java 文件。

在项目中使用

当我们想要在项目中使用的时候，gradle 的配置要分为老版本和新版本的 gradle 来处理

老版本

首先需要在 Project 的 build.gradle 中添加如下的配置:

```
dependencies {  
    classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'  
}
```

之后需要在 app 的 build.gradle 中添加如下配置:

```
apply plugin: 'com.android.application'  
apply plugin: 'com.neenbedankt.android-apt'  
dependencies {  
    implementation project(':ViewAnnotation')  
    apt project(':ViewAnnotation')  
}
```

新版本

新版本下，使用老版本的方式会报错

```
android-apt plugin is incompatible with future version of  
Android Gradle plugin. use 'annotationProcessor'  
configuration instead.
```

也就是说新版本的 Gradle 已经全面使用 annotationProcessor 来替代了 apt

我们只需要按照如下的设置 app 的 build.gradle 即可

```
apply plugin: 'com.android.application'
dependencies {
    implementation project(':ViewAnnotation')
    annotationProcessor project(':ViewAnnotation')
}
```

在项目中使用

配置完 gradle 之后，我们便可以在项目中使用了

```
@DIActivity
public class MainActivity extends AppCompatActivity {
    @DIView(R.id.tv_text)
    TextView mTvText;
    @DIView(R.id.btn_change)
    Button mBtnChange;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        DIMainActivity.bindView(this);
        mTvText.setText("Test");
        mBtnChange.setOnClickListener(view->mTvText.setText("NotExpectError"));
    }
}
```

build 之后我们可以看到编译期自动为我们生成的代码

```
public final class DIMainActivity extends MainActivity {
    public static void bindView(MainActivity activity) {
        activity.mTvText = (android.widget.TextView) activity.findViewById(2131165
326);
        activity.mBtnChange = (android.widget.Button) activity.findViewById(213116
5218);
```

```
    }  
}
```

可以看到，和我们之前代码中的描述相同。

ButterKnife 的流程

最后简单描述一下 ButterKnife 的 `@BindView` 注解生成代码的实现流程。

1. `@BindView` 在编译时，根据 `xxxActivity` 生成了 `xxxActivity$$ViewBinder`
2. 根据 `Activity` 中调用的 `ButterKnife.bind(this);` 通过 `this` 的类名 `+ $$ViewBinder`，反射得到了 `viewBinder`，和编译处理器生产的 `java` 文件关联起来，并将其存在 `map` 中缓存，然后调用 `ViewBinder.bind()`。
3. 在 `ViewBinder` 的 `bind` 方法中通过 `id`，利用 `ButterKnife` 的 `butterknife.internal.Utils` 工具类中的封装方法，将 `findViewById()` 控件注入到 `Activity` 的参数中。

12. View 的工作机制源码分析

1、`ViewGroup` 首先根据自己的 `MeasureSpec` 和子 `View` 自己设定的宽高，来为每个子 `View` 生成 `MeasureSpec`

2、ViewGroup 循环调用每个子 View 的 measure 方法，同时把子 View 的 MeasureSpec 传递过去

3、子 View 根据父视图给它的 MeasureSpec 确定最终的测量大小

可以看出，决定子 View 测量宽高的因素有：父视图的 MeasureSpec、子 View 自身指定的大小

[View 的工作机制源码分析完整版地址](#)

13.Android 触摸事件分发机制源码分析

[Android 触摸事件分发机制源码分析完整版地址](#)

触摸事件分发流程：

1、ViewGroup 的事件分发

- 首先判断自身是否需要，需要则进行拦截，并调用自身的 onTouchEvent 方法进行使用并决定是否消费，子 View 对事件无感。
- 自己不需要则不进行拦截，事件分发给子 View（事件会分发给手指触摸区域的子 View，有重叠时传递给最上层 View）
- 子 View 不消费时，事件会回传，调用自身的 onTouchEvent 方法进行使用并决定是否消费；子 View 消费时事件不回传

伪代码：

```
public boolean dispatchTouchEvent(MotionEvent event) {  
    // 默认状态为未消费过
```

```
boolean result = false;

// 如果没有拦截

if (!onInterceptTouchEvent(event)) {

    // 则交给子 view

    result = child.dispatchTouchEvent(event);

}

// 如果事件没有被子 View 消费

if (!result) {

    // 则询问自身 onTouchEvent()

    result = onTouchEvent(event);

}

// 返回事件消费状态

return result;
}
```

2、View 的事件分发

dispatchTouchEvent 方法对 View 的监听器和方法进行分发，顺序如下：

onTouchListener --> onTouchEvent --> onLongClickListener -->
onClickListener

14.Android 按键事件分发机制源码分析

[Android 按键事件分发机制源码分析完整版地址](#)

分发流程：

1、Activity 的分发

- Activity 接收到事件，为 Menu 键直接消费掉。否则分发给 PhoneWindow，PhoneWindow 直接传递给它的 DecorView
- DecorView 对 Back 键进行判断是否消费，不消费则分发给 ViewGroup
- ViewGroup 不消费时，事件分发给 KeyEvent

2、ViewGroup 的分发

- ViewGroup 有焦点且大小确定时，首先自己处理
- ViewGroup 的子 View 有焦点且大小确定时，分发给子 View
- 层层向下分发，直到最底层 View

3、View 的分发

View 首先将事件传递给 onKeyListner 监听器，不消费时传递给 KeyEvent

4、KeyEvent 的分发

KeyEvent 会根据类型来分别决定回调 Activity 或 View 的 onKeyDown、onKeyUp、onKeyLongPress 方法

5、事件回传

当 Activity、View 的 onKeyDown、onKeyUp 也没有消费事件时，事件会进行回传

15. 深入解析 Handler 源码

大家应该都知道，Android 的消息机制是基于 Handler 实现的。还记得一年前的自己就看了几篇博客，知道了 Handler、Looper、MessageQueue 就自以为了解了 Handler 的原理。但其实看源码的过程中慢慢就会发现，Handler 的内容可不止这点，像同步屏障、Handler 的 native 层的阻塞唤醒机制等等这些知识以前就没有理解清楚。因此写下这篇文章，从头开始重塑对 Handler 的印象。

Handler 采用的是一种生产者-消费者模型，Handler 就是生产者，通过它可以生产需要执行的任务。而 Looper 则是消费者，不断从 MessageQueue 中取出 Message 对这些消息进行消费，下面我们看一下其具体的实现。

发送消息

post & sendMessage

首先我们都知道，Handler 对外主要有两种方式来实现在其所在 Looper 所在线程执行指定 Runnable——post 及 sendMessage，它们都有对应的 delay 方法。而不论是 post 还是 sendMessage，都会调用到 sendMessageDelayed 方法。比如下面是 post 方法的实现：

```
public final boolean post(Runnable r) {  
    return sendMessageDelayed(getPostMessage(r), 0);  
}
```

可以看到它其实调用的仍然是 `sendMessageDelayed` 方法，只是通过 `getPostMessage` 方法将这个 `Runnable` 包装成了一个 `Message` 对象。

```
private static Message getPostMessage(Runnable r) {
    Message m = Message.obtain();
    m.callback = r;
    return m;
}
```

这个包装出来的 `Message` 将 `callback` 设置为了对应的 `Runnable`。

而所有的 `sendMessage` 和 `post` 方法，实际上最后都通过

`sendMessageDelayed` 方法调用到了 `sendMessageAtTime` 方法：

```
public final boolean sendMessageDelayed(Message msg, long delayMillis) {
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}
```

在 `sendMessageAtTime` 中，它首先通过 `mQueue` 拿到了对应的 `MessageQueue` 对象，然后调用了 `enqueueMessage` 方法将 `Message` 发送至 `MessageQueue` 中。

```
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(
                this + " sendMessageAtTime() called with no mQueue");
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}
```

最后实际上是调用到了 MessageQueue 的 enqueueMessage 方法将这个消息传入了 MessageQueue。它将 Message 的 target 设置为了当前的 Handler，同时要注意看到，这里在 enqueueMessage 之前先判断了一下 mAsynchronous 是否为 true，若为 true 则将该 Message 的 Asynchronous 置为 true。

```
private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
    msg.target = this;
    if (mAsynchronous) {
        msg.setAsynchronous(true);
    }
    return queue.enqueueMessage(msg, uptimeMillis);
}
```

那这个 mAsynchronous 是什么时候被赋值的呢？点进去看可以发现它的赋值是在 Handler 的构造函数中进行的。也就是说创建的 Handler 时若将 async 置为 true 则该 Handler 发出的 Message 都会被设为 Async，也就是『异步消息』。

- public Handler(Callback callback, boolean async)
- public Handler(Looper looper, Callback callback, boolean async)

关于异步消息和同步消息是什么，我们放在后面讨论。

Handler 有很多种构造函数，但其他的构造函数最后仍然会调用到上述的两种构造函数，其 async 默认会被设置为 false。

让我们看看上述的两种构造函数：

```
public Handler(Callback callback, boolean async) {
    if (FIND_POTENTIAL_LEAKS) {
        final Class<? extends Handler> klass = getClass();
```

```

        if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocal
class()) &&
            (klass.getModifiers() & Modifier.STATIC) == 0) {
        Log.w(TAG, "The following Handler class should be static or leaks m
ight occur: " +
            klass.getCanonicalName());
    }
}

mLooper = Looper.myLooper();
if (mLooper == null) {
    throw new RuntimeException(
        "Can't create handler inside thread that has not called Looper.prep
are()");
}
mQueue = mLooper.mQueue;
mCallback = callback;
mAsynchronous = async;
}

```

可以看到这个构造函数主要是对 mLooper、mQueue、mCallback、mAsynchronous 进行赋值，其中 mLooper 是通过 Looper.myLooper 方法获取到的，另一种构造函数除了 Looper 是通过外部传入以外和这个构造函数的实现差不多。同时我们还能看出，mQueue 这个 MessageQueue 是 Looper 对象内部的一个成员变量。

消息入队

enqueueMessage

我们接着看看 Handler 发送了消息后 MessageQueue 的 enqueueMessage 方法：

```

boolean enqueueMessage(Message msg, long when) {
    if (msg.target == null) {
        throw new IllegalArgumentException("Message must have a target.");
    }
    if (msg.isInUse()) {

```

```
        throw new IllegalStateException(msg + " This message is already in use.");
    });

    synchronized (this) {
        if (mQuitting) {
            IllegalStateException e = new IllegalStateException(
                msg.target + " sending message to a Handler on a dead thread");
        }

        Log.w(TAG, e.getMessage(), e);
        msg.recycle();
        return false;
    }

    msg.markInUse();
    msg.when = when;
    Message p = mMessages;
    boolean needWake;
    if (p == null || when == 0 || when < p.when) {
        // New head, wake up the event queue if blocked.
        msg.next = p;
        mMessages = msg;
        needWake = mBlocked;
    } else {
        // Inserted within the middle of the queue. Usually we don't have to wake
        // up the event queue unless there is a barrier at the head of the queue
        // and the message is the earliest asynchronous message in the queue.
        needWake = mBlocked && p.target == null && msg.isAsynchronous();
        Message prev;
        for (;;) {
            prev = p;
            p = p.next;
            if (p == null || when < p.when) {
                break;
            }
            if (needWake && p.isAsynchronous()) {
                needWake = false;
            }
        }
        msg.next = p; // invariant: p == prev.next
        prev.next = msg;
    }
    // We can assume mPtr != 0 because mQuitting is false.
}
```

```
    if (needWake) {
        nativeWake(mPtr);
    }
}
return true;
}
```

可以看到，`MessageQueue` 实际上里面维护了一个 `Message` 构成的链表，每次插入数据都会按时间顺序进行插入，也就是说 `MessageQueue` 中的 `Message` 都是按照时间排好序的，这样的话就使得循环取出 `Message` 的时候只需要一个个地从前往后拿即可，这样 `Message` 都可以按时间先后顺序被消费。

最后在需要唤醒的情况下会调用 `nativeWake` 这个 `native` 方法用于进行唤醒，这些和唤醒机制有关的代码我们后面再进行讨论，先暂时放在一边。

消息循环

那么我们看看 `Looper.myLooper` 方法是如何获取到 `Looper` 对象的呢？

```
public static @Nullable Looper myLooper() {
    return sThreadLocal.get();
}
```

可以看出来，这个 `Looper` 对象是通过 `sThreadLocal.get` 方法获取到的，也就是说这个 `Looper` 是一个线程独有的变量，每个线程具有一个不同的 `Looper`。

那么这个 `Looper` 对象是何时创建又何时放入这个 `ThreadLocal` 中的呢？

我们通过跟踪可以发现它实际上是通过 `Looper.prepare` 方法放入 `ThreadLocal` 中的：

```
private static void prepare(boolean quitAllowed) {
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    sThreadLocal.set(new Looper(quitAllowed));
}
```

那按道理来说我们在应用程序中并没有调用 `Looper.prepare` 方法，为何还能通过主线程的 `Handler` 发送 `Message` 到主线程呢？

其实这个 `Looper.prepare` 方法在主线程创建时就已经被创建并调用了 `prepare` 方法进行设置，具体我们可以看到 `ActivityThread` 类的 `main` 函数：

```
public static void main(String[] args) {
    // ...
    Looper.prepareMainLooper();
    // ...
    Looper.loop();
    // ...
}
```

这个 `main` 函数其实就是我们进程的入口，可以看出来它首先调用了 `Looper.prepareMainLooper` 创建了主线程的 `Looper` 并传入 `ThreadLocal`，自此我们就可以在主线程发送消息了。为什么要这样设计呢？因为其实我们的 `View` 绘制事件等都是通过主线程的 `Handler` 来进行调度的。

我们接着看到 `Looper.loop` 方法：

```
public static void loop() {
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
    }
    final MessageQueue queue = me.mQueue;
    // Make sure the identity of this thread is that of the Local process,
```

```
// and keep track of what that identity token actually is.  
Binder.clearCallingIdentity();  
final long ident = Binder.clearCallingIdentity();  
for (;;) {  
    Message msg = queue.next(); // might block  
    if (msg == null) {  
        // No message indicates that the message queue is quitting.  
        return;  
    }  
    // ...  
    final long start = (slowDispatchThresholdMs == 0) ? 0 : SystemClock.uptimeMillis();  
    final long end;  
    try {  
        msg.target.dispatchMessage(msg);  
        end = (slowDispatchThresholdMs == 0) ? 0 : SystemClock.uptimeMillis();  
    }  
    } finally {  
        // ...  
    }  
    // ...  
    msg.recycleUnchecked();  
}  
}
```

这里其实是一个死循环，它的主要作用是遍历 MessageQueue，获取到 Looper 及 MessageQueue 后，不断通过 MessageQueue 的 next 方法获取到消息列表中的下一个 Message，之后调用了 Message 的 target 的 dispatchMessage 方法对 Message 进行消费，最后对 Message 进行了回收。

通过上面的代码可以看出，Looper 主要的作用是遍历 MessageQueue，每找到一个 Message 都会调用其 target 的 dispatchMessage 对该消息进行消费，这里的 target 也就是我们之前发出该 Message 的 Handler。

消息遍历

我们接着看到消息的遍历过程，它不断地从 MessageQueue 中调用 next 方法拿到消息，并对其进行消费，那我们具体看看 next 的过程：

```
Message next() {
    final long ptr = mPtr;
    if (ptr == 0) {
        return null;
    }
    int pendingIdleHandlerCount = -1;
    int nextPollTimeoutMillis = 0;
    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
        }
        // 1
        nativePollOnce(ptr, nextPollTimeoutMillis);
        synchronized (this) {
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            // 2
            if (msg != null && msg.target == null) {
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
            if (msg != null) {
                // 3
                if (now < msg.when) {
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALUE);
                } else {
                    // Got a message.
                    mBlocked = false;
                    if (prevMsg != null) {
                        prevMsg.next = msg.next;
                    } else {
                        mMessages = msg.next;
                    }
                    msg.next = null;
                    if (DEBUG) Log.v(TAG, "Returning message: " + msg);
                }
            }
        }
    }
}
```

```

        msg.markInUse();
        return msg;
    }
} else {
    // No more messages.
    nextPollTimeoutMillis = -1;
}
if (mQuitting) {
    dispose();
    return null;
}
// 4
if (pendingIdleHandlerCount < 0
    && (mMessages == null || now < mMessages.when)) {
    pendingIdleHandlerCount = mIdleHandlers.size();
}
if (pendingIdleHandlerCount <= 0) {
    // No idle handlers to run. Loop and wait some more.
    mBlocked = true;
    continue;
}
if (mPendingIdleHandlers == null) {
    mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
}
mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
}
for (int i = 0; i < pendingIdleHandlerCount; i++) {
    final IdleHandler idler = mPendingIdleHandlers[i];
    mPendingIdleHandlers[i] = null; // release the reference to the idler
    boolean keep = false;
    try {
        keep = idler.queueIdle();
    } catch (Throwable t) {
        Log.wtf(TAG, "IdleHandler threw exception", t);
    }
    if (!keep) {
        synchronized (this) {
            mIdleHandlers.remove(idler);
        }
    }
}
pendingIdleHandlerCount = 0;

```

```
    nextPollTimeoutMillis = 0;  
}  
}
```

上面的代码比较长，我们一步一步来进行分析

首先在 `next` 方法内，它在不断地进行着循环，在 1 处它先调用了一次 `nativePollOnce` 这个 `native` 方法，它与 `Handler` 的阻塞唤醒机制有关，我们后面再进行介绍。

之后，在 2 处，它进行了一个非常特殊的处理。这里判断当前的消息是否是 `target` 为 `null` 的消息，若 `target` 为 `null`，则它会不断地向下取 `Message`，直到遇到一个异步的消息。到这里可能会有读者觉得很奇怪了，明明在 `enqueueMessage` 中避免了 `Message` 的 `target` 为 `null`，为什么这里还会存在 `target` 为 `null` 的消息呢？其实这与 `Handler` 的同步屏障机制有关，我们稍后介绍

之后便在注释 3 处判断判断当前消息是否到了应该发送的时间，若到了应该发送的时间，就会将该消息取出并返回，否则仅仅是将 `nextPollTimeoutMillis` 置为了剩余的时间（这里为了防止 `int` 越界做了防越界处理）

之后在注释 4 处，第一次循环的前提下，若 `MessageQueue` 为空或者消息未来才会执行，则会尝试去执行一些 `idleHandler`，并在执行后将 `pendingIdleHandlerCount` 置为 0 避免下次再次执行。

若这一次拿到的消息不是现在该执行的，那么会再次调用到 `nativePollOnce`，并且此次的 `nextPollTimeoutMillis` 不再为 0 了，这与我们后面会提到的阻塞唤醒机制有关。

消息的处理

消息的处理是通过 Handler 的 `dispatchMessage` 实现的：

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}
```

它优先调用了 Message 的 `callback`, 若没有 `callback` 则会调用 Handler 中 `Callback` 的 `handleMessage` 方法, 若其仍没定义则最终会调用到 Handler 自身所实现的 `handleMessage` 方法。

因此我们在使用的时候可以根据自己的需求来重写上面三者其中一个。

同步屏障机制

Handler 中存在着一种叫做同步屏障的机制, 它可以实现异步消息优先执行的功能, 让我们看看它是如何实现的。

加入同步屏障

在 Handler 中还存在了一种特殊的消息, 它的 `target` 为 `null`, 并不会被消费, 仅仅是作为一个标识处于 `MessageQueue` 中。它就是 `SyncBarrier` (同步屏障)

这种特殊的消息。我们可以通过 `MessageQueue::postSyncBarrier` 方法将其加入消息队列。

```
private int postSyncBarrier(long when) {
    // Enqueue a new sync barrier token.
    // We don't need to wake the queue because the purpose of a barrier is to stall it.
    synchronized (this) {
        final int token = mNextBarrierToken++;
        final Message msg = Message.obtain();
        msg.markInUse();
        msg.when = when;
        msg.arg1 = token;
        Message prev = null;
        Message p = mMessages;
        if (when != 0) {
            while (p != null && p.when <= when) {
                prev = p;
                p = p.next;
            }
        }
        if (prev != null) { // invariant: p == prev.next
            msg.next = p;
            prev.next = msg;
        } else {
            msg.next = p;
            mMessages = msg;
        }
        return token;
    }
}
```

可以看到，这里并没有什么特殊的，只是将一个 `target` 为 `null` 的消息加入了消息队列中，但我们在前面的 `enqueueMessage` 方法中也看到了，普通的 `enqueue` 操作是没有办法在消息队列中放入这样一个 `target` 为 `null` 的消息的。因此这种同步屏障只能通过这个方法发出。

移除同步屏障

我们可以通过 `removeSyncBarrier` 方法来移除消息屏障。

```
public void removeSyncBarrier(int token) {
    // Remove a sync barrier token from the queue.
    // If the queue is no longer stalled by a barrier then wake it.
    synchronized (this) {
        Message prev = null;
        Message p = mMessages;
        // 找到 target 为 null 且 token 相同的消息
        while (p != null && (p.target != null || p.arg1 != token)) {
            prev = p;
            p = p.next;
        }
        if (p == null) {
            throw new IllegalStateException("The specified message queue synchronization "
                + "barrier token has not been posted or has already been removed.");
        }
        final boolean needWake;
        if (prev != null) {
            prev.next = p.next;
            needWake = false;
        } else {
            mMessages = p.next;
            needWake = mMessages == null || mMessages.target != null;
        }
        p.recycleUnchecked();
        // If the loop is quitting then it is already awake.
        // We can assume mPtr != 0 when mQuitting is false.
        if (needWake && !mQuitting) {
            nativeWake(mPtr);
        }
    }
}
```

这里主要是将同步屏障从 `MessageQueue` 中移除，一般执行完了异步消息后就会通过该方法将同步屏障移除。

最后若需要唤醒，调用了 `nativeWake` 方法进行唤醒。

同步屏障的作用

而看了前面 `MessageQueue::next` 的代码我们知道，当 `MessageQueue` 中遇到了一个同步屏障，则它会不断地忽略后面的同步消息直到遇到一个异步的消息，这样设计的目的其实是为了使得当队列中遇到同步屏障时，则会使得异步的消息优先执行，这样就可以使得一些消息优先执行。比如 `View` 的绘制过程中的 `TraversalRunnable` 消息就是异步消息，在放入队列之前先放入了一个消息屏障，从而使得界面绘制的消息会比其他消息优先执行，避免了因为 `MessageQueue` 中消息太多导致绘制消息被阻塞导致画面卡顿，当绘制完成后，就会将消息屏障移除。

阻塞唤醒机制

从前面可以看出来 `Handler` 中其实还存在着一种阻塞唤醒机制，我们都知道不断地进行循环是非常消耗资源的，有时我们 `MessageQueue` 中的消息都不是当下就需要执行的，而是要过一段时间，此时如果 `Looper` 仍然不断进行循环肯定是一种对于资源的浪费。因此 `Handler` 设计了这样一种阻塞唤醒机制使得在当下没有需要执行的消息时，就将 `Looper` 的 `loop` 过程阻塞，直到下一个任务的执行时间到达或者一些特殊情况下再将其唤醒，从而避免了上述的资源浪费。

epoll

这个阻塞唤醒机制是基于 Linux 的 I/O 多路复用机制 `epoll` 实现的，它可以同时监控多个文件描述符，当某个文件描述符就绪时，会通知对应程序进行读/写操作。

epoll 主要有三个方法，分别是 `epoll_create`、`epoll_ctl`、`epoll_wait`。

epoll_create

```
int epoll_create(int size)
```

其功能主要是创建一个 epoll 句柄并返回，传入的 `size` 代表监听的描述符个数（仅仅是初次分配的 `fd` 个数）

epoll_ctl

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

其功能是对 epoll 事件进行注册，会对该 `fd` 执行指定的 `op` 操作，参数含义如下：

- `epfd`: epoll 的句柄值（也就是 `epoll_create` 的返回值）
- `op`: 对 `fd` 执行的操作
 - `EPOLL_CTL_ADD`: 注册 `fd` 到 `epfd`
 - `EPOLL_CTL_DEL`: 从 `epfd` 中删除 `fd`
 - `EPOLL_CTL_MOD`: 修改已注册的 `fd` 的监听事件
- `fd`: 需要监听的文件描述符
- `epoll_event`: 需要监听的事件

`epoll_event` 是一个结构体，里面的 `events` 代表了对应文件操作符的操作，而 `data` 代表了用户可用的数据。

其中 `events` 可取下面几个值：

- **EPOLLIN** : 表示对应的文件描述符可以读(包括对端 SOCKET 正常关闭);
- **EPOLLOUT**: 表示对应的文件描述符可以写;
- **EPOLLPRI**: 表示对应的文件描述符有紧急的数据可读 (这里应该表示有带外部数据来) ;
- **EPOLLERR**: 表示对应的文件描述符发生错误;
- **EPOLLHUP**: 表示对应的文件描述符被挂断;
- **EPOLLET**: 将 EPOLL 设为边缘触发(Edge Triggered)模式, 这是相对于水平触发(Level Triggered)来说的。
- **EPOLLONESHOT**: 只监听一次事件, 当监听完这次事件之后, 如果还需要继续监听这个 socket 的话, 需要再次把这个 socket 加入到 EPOLL 队列里

epoll_wait

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

其功能是等待事件的上报, 参数含义如下:

- **epfd**: epoll 的句柄值
- **events**: 从内核中得到的事件集合
- **maxevents**: events 数量, 不能超过 create 时的 size
- **timeout**: 超时时间

当调用了该方法后, 会进入阻塞状态, 等待 epfd 上的 IO 事件, 若 epfd 监听的某个文件描述符发生前面指定的 event 时, 就会进行回调, 从而使得 epoll 被唤醒并返回需要处理的事件个数。若超过了设定的超时时间, 同样也会被唤醒并返回 0 避免一直阻塞。

而 Handler 的阻塞唤醒机制就是基于上面的 epoll 的阻塞特性，我们来看看它的具体实现。

native 初始化

在 Java 中的 MessageQueue 创建时会调用到 nativeInit 方法，在 native 层会创建 NativeMessageQueue 并返回其地址，之后都是通过这个地址来与该 NativeMessageQueue 进行通信（也就是 MessageQueue 中的 mPtr，类似 MMKV 的做法），而在 NativeMessageQueue 创建时又会创建 Native 层下的 Looper，我们看到 Native 下的 Looper 的构造函数：

```
Looper::Looper(bool allowNonCallbacks) :  
    mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),  
    mPolling(false), mEpollFd(-1), mEpollRebuildRequired(false),  
    mNextRequestSeq(0), mResponseIndex(0), mNextMessageUptime(LLONG_MAX) {  
    mWakeEventFd = eventfd(0, EFD_NONBLOCK); // 构造唤醒事件的 fd  
    AutoMutex _l(mLock);  
    rebuildEpollLocked();  
}
```

可以看到，它调用了 rebuildEpollLocked 方法对 epoll 进行初始化，让我们看看其实现

```
void Looper::rebuildEpollLocked() {  
    if (mEpollFd >= 0) {  
        close(mEpollFd);  
    }  
    mEpollFd = epoll_create(EPOLL_SIZE_HINT);  
    struct epoll_event eventItem;  
    memset(& eventItem, 0, sizeof(epoll_event));  
    eventItem.events = EPOLLIN;  
    eventItem.data.fd = mWakeEventFd;  
    int result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeEventFd, & eventItem);  
    for (size_t i = 0; i < mRequests.size(); i++) {  
        const Request& request = mRequests.valueAt(i);  
    }  
}
```

```
    struct epoll_event eventItem;
    request.initEventItem(&eventItem);
    int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, request.fd, & even
tItem);
}
}
```

可以看到，这里首先关闭了旧的 epoll 描述符，之后又调用了 epoll_create 创建了新的 epoll 描述符，然后进行了一些初始化后，将 mWakeEventFd 及 mRequests 中的 fd 都注册到了 epoll 的描述符中，注册的事件都是 EPOLLIN。

这就意味着当这些文件描述符其中一个发生了 IO 时，就会通知 epoll_wait 使其唤醒，那么我们猜测 Handler 的阻塞就是通过 epoll_wait 实现的。

同时可以发现，Native 层也是存在 MessageQueue 及 Looper 的，也就是说 Native 层实际上也是有一套消息机制的，这些我们到后面再进行介绍。

native 阻塞实现

我们看看阻塞，它的实现就在我们之前看到的 MessageQueue::next 中，当发现要返回的消息将来才会执行，则会计算出当下距离其将要执行的时间还差多少毫秒，并调用 nativePollOnce 方法将返回的过程阻塞到指定的时间。

nativePollOnce 很显然是一個 native 方法，它最后调用到了 Looper 这个 native 层类的 pollOnce 方法。

```
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** out
Data) {
    int result = 0;
```

```

    for (;;) {
        while (mResponseIndex < mResponses.size()) {
            const Response& response = mResponses.itemAt(mResponseIndex++);
            int ident = response.request.ident;
            if (ident >= 0) {
                int fd = response.request.fd;
                int events = response.events;
                void* data = response.request.data;
                if (outFd != NULL) *outFd = fd;
                if (outEvents != NULL) *outEvents = events;
                if (outData != NULL) *outData = data;
                return ident;
            }
        }
        if (result != 0) {
            if (outFd != NULL) *outFd = 0;
            if (outEvents != NULL) *outEvents = 0;
            if (outData != NULL) *outData = NULL;
            return result;
        }
        result = pollInner(timeoutMillis);
    }
}

```

前面主要是一些对 Native 层消息机制的处理，我们先暂时不关心，这里最后调用到了 `pollInner` 方法：

```

int Looper::pollInner(int timeoutMillis) {
    // ...
    int result = POLL_WAKE;
    mResponses.clear();
    mResponseIndex = 0;
    mPolling = true;
    struct epoll_event eventItems[EPOLL_MAX_EVENTS];
    // 1
    int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS, timeoutMillis);
    // ...
    return result;
}

```

可以发现，这里在 1 处调用了 `epoll_wait` 方法，并传入了我们之前在 `nativePollOnce` 方法传入的当前时间距下个任务执行时间的差值。这就是我们的阻塞功能的核心实现了，调用该方法后，会一直阻塞，直到到达我们设定的时间或之前我们在 `epoll` 的 `fd` 中注册的几个 `fd` 发生了 IO。其实到了这里我们就可以猜到，`nativeWake` 方法就是通过对注册的 `mWakeEventFd` 进行操作从而实现的唤醒。

后面主要是一些对 Native 层消息机制的处理，这篇文章暂时不关注，它的逻辑和 Java 层是基本一致的。

native 唤醒

`nativeWake` 方法最后通过 `NativeMessageQueue` 的 `wake` 方法调用到了 Native 下 `Looper` 的 `wake` 方法：

```
void Looper::wake() {
    uint64_t inc = 1;
    ssize_t nWrite = TEMP_FAILURE_RETRY(write(mWakeEventFd, &inc, sizeof(uint64_t)));
    if (nWrite != sizeof(uint64_t)) {
        if (errno != EAGAIN) {
            ALOGW("Could not write wake signal, errno=%d", errno);
        }
    }
}
```

这里其实就是调用了 `write` 方法，对 `mWakeEventFd` 中写入了 1，从而使得监听该 `fd` 的 `pollOnce` 方法被唤醒，从而使得 Java 中的 `next` 方法继续执行。

那我们再回去看看，在什么情况下，Java 层会调用 `nativeWake` 方法进行唤醒呢？

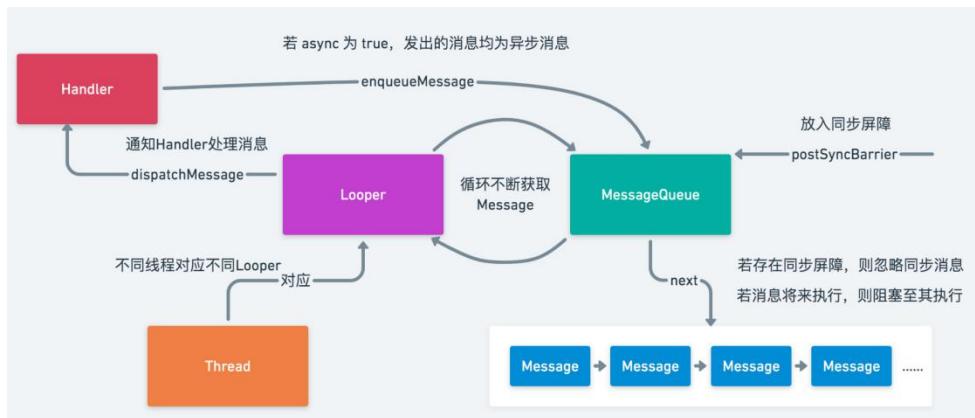
MessageQueue 类中调用 nativeWake 方法主要有下列几个时机：

- 调用 MessageQueue 的 quit 方法进行退出时，会进行唤醒
- 消息入队时，若插入的消息在链表最前端（最早将执行）或者有同步屏障时插入的是最前端的异步消息（最早被执行的异步消息）
- 移除同步屏障时，若消息列表为空或者同步屏障后面不是异步消息时

可以发现，主要是在可能不再需要阻塞的情况下进行唤醒。（比如加入了一个更早的任务，那继续阻塞显然会影响这个任务的执行）

总结

Android 的消息机制在 Java 层及 Native 层均是由 Handler、Looper、MessageQueue 三者构成



- Handler: 事件的发送及处理者，在构造方法中可以设置其 async，默认为 false。若 async 为 true 则该 Handler 发送的 Message 均为异步消息，有同步屏障的情况下会被优先处理。

- **Looper**: 一个用于遍历 `MessageQueue` 的类，每个线程有一个独有的 `Looper`，它会在所处的线程开启一个死循环，不断从 `MessageQueue` 中拿出消息，并将其发送给 `target` 进行处理
- **MessageQueue**: 用于存储 `Message`，内部维护了 `Message` 的链表，每次拿取 `Message` 时，若该 `Message` 离真正执行还需要一段时间，会通过 `nativePollOnce` 进入阻塞状态，避免资源的浪费。若存在消息屏障，则会忽略同步消息优先拿取异步消息，从而实现异步消息的优先消费。

相关问题

下面还有一些与 `Handler` 相关的常见问题，可以结合前面的内容得到答案。

问题 1

`Looper` 是在主线程创建，同时其 `loop` 方法也是在主线程执行，为什么这样一个死循环却不会阻塞主线程呢？

我们看到 `ActivityThread` 中，它实际上是一个 `Handler`，我们在使用的过程中的很多事件（如 `Activity`、`Service` 的各种生命周期）都在这里的各种 `Case` 中，也就是说我们平时说的主线程其实就是依靠这个 `Looper` 的 `loop` 方法来处理各种消息，从而实现如 `Activity` 的声明周期的回调等等的处理，从而回调给我们使用者。

```
public void handleMessage(Message msg) {
    if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
    switch (msg.what) {
```

```
        case XXX:
            // ...
        }
        Object obj = msg.obj;
        if (obj instanceof SomeArgs) {
            ((SomeArgs) obj).recycle();
        }
        if (DEBUG_MESSAGES) Slog.v(TAG, "<<< done: " + codeToString(msg.what));
    }
}
```

因此不能说主线程不会阻塞，因为主线程本身就是阻塞的，其中所有事件都由主线程进行处理，从而使得我们能在这个循环的过程中作出自己的各种处理（如 View 的绘制等）。

而这个问题的意思应该是为何这样一个死循环不会使得界面卡顿，这有两个原因：

1. 界面的绘制本身就是这个循环内的一个事件
2. 界面的绘制是通过了同步屏障保护下发送的异步消息，会被主线程优先处理，因此使得界面绘制拥有了最高的优先级，不会因为 Handler 中事件太多而造成卡顿。

问题 2

Handler 的内存泄漏是怎么回事？如何产生的呢？

首先，造成 Handler 的内存泄漏往往是因为如下的这种代码：

```
public class XXXActivity extends BaseActivity {
    // ...
    private Handler mHandler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            // 一些处理
        }
    }
}
```

```
};  
// ...  
}
```

那这样为什么会造成内存泄漏呢？

我们都知道，匿名内部类会持有外部类的引用，也就是说这里的 Handler 会持有其外部类 XXXActivity 的引用。而我们可以回忆一下 sendMessage 的过程中，它会将 Message 的 target 设置为 Handler，也就是说明这个 Message 持有了 mHandler 的引用。那么我们假设通过 mHandler 发送了一个 2 分钟后的延时消息，在两分钟还没到的时候，我们关闭了界面。按道理来说此时 Activity 可以被 GC 回收，但由于此时 Message 还处于 MessageQueue 中，MessageQueue 这个对象持有了 Message 的引用，Message 又持有了我们的 Handler 的引用，同时由于 Handler 又持有了其外部类 XXXActivity 的引用。这就导致此时 XXXActivity 仍然是可达的，因此导致 XXXActivity 无法被 GC 回收，这就造成了内存泄漏。

因此我们使用 Handler 最好将其定义为 static 的，避免其持有外部类的引用导致类似的内存泄漏问题。如果此时还需要用到 XXXActivity 的一些信息，可以通过 WeakReference 来使其持有 Activity 的弱引用，从而可以访问其中的某些信息，又避免了内存泄漏问题。

16. 深入解析 Binder 源码

简介

Binder 在 Android 中堪称武林秘籍中的"易筋经"，无论是菜鸟还是老鸟都对之神往。Binder 架构是进程间相互通信的最常用手段，四大组件的基本功能都是依赖着 Binder 才能够实现的。

为了开发者能够使用 java 与 cpp 进行 binder 通信，binder 的设计贯穿了 framework、native 和 kernel 层，开发者可以轻松的在上层使用 binder 向其它进程发起数据通信。

这是我第三次系统性的阅读 binder 代码了，之前的两次可以去 [Binder_SourceCode](#) 和 [Android 面试考 Binder，看这一篇就够了](#) 中阅读。

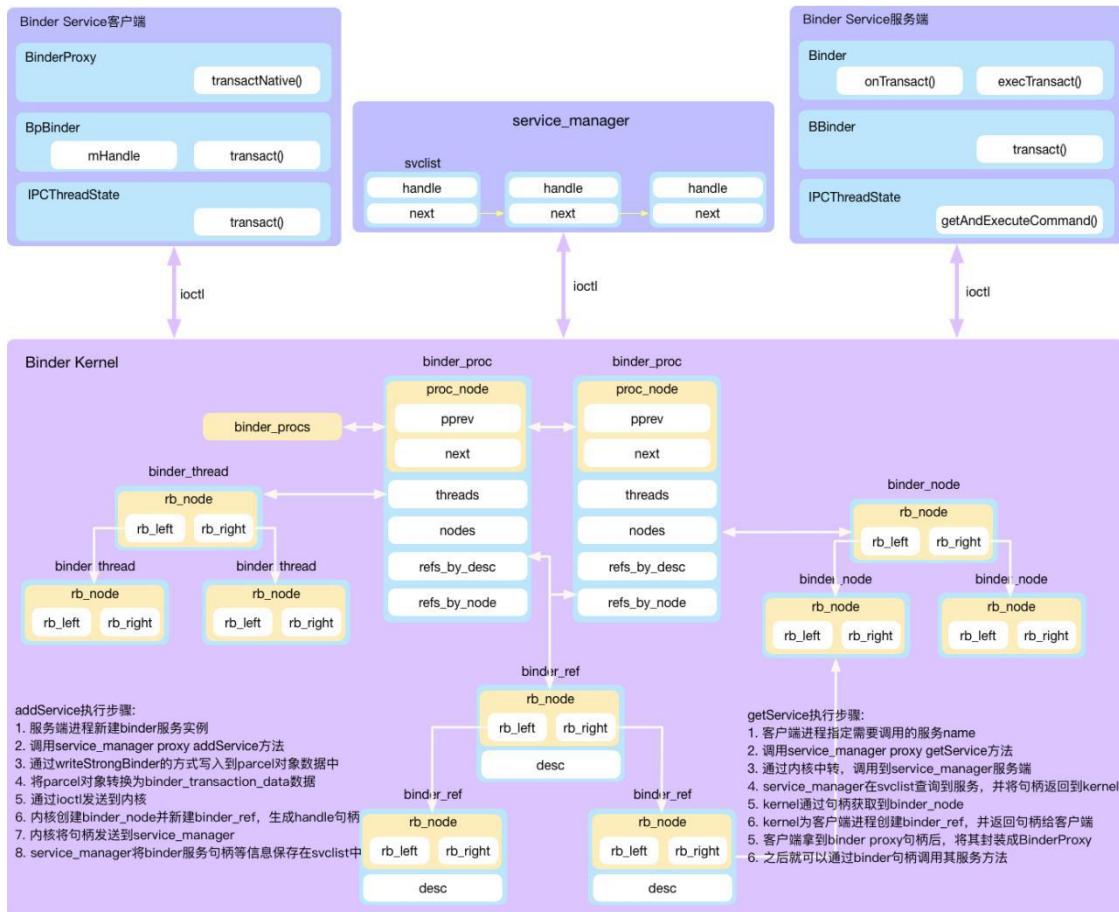
这一次的源码分析，除了分析 binder 数据传输的主干，我会单独提炼出我平时的一些困惑，例如：

- 1、binder kernel 是如何组织数据结构的，如其中的 `binder_proc->refs_by_desc`, `refs_by_node` 这些重要的红黑树
- 2、binder 涉及到的基本并发设计模式，例如典型的生产者-消费者模型的运用
- 3、与 binder 客户端紧密联系的 handle 句柄是怎样在 kernel 中查询到服务进程的
- 4、linux 基本数据结构及 `ioctl`, `mmap` 这些基础系统调用的作用及原理
- 5、死亡回调究竟是依赖着什么实现的...

等等问题，都会在这篇文章中一一分析

整体图

设计架构



以上图片大体列出了实名 Binder 的设计架构，几点需要注意的点：

- 1、`BinderProxy` 为客户端在 java 层的表示，它的成员变量 `mObject` 是和 native 层的表示 `BpBinder` 的 `mObjects` 一一对应的，通过调用 `BinderProxy` 的 `tractNative` 方法，java 层可以调用到 native 层 `BpBinder`，随后调用到进行 IPC 通信的控制器 `IPCThreadState.transact()` 向内核发送请求

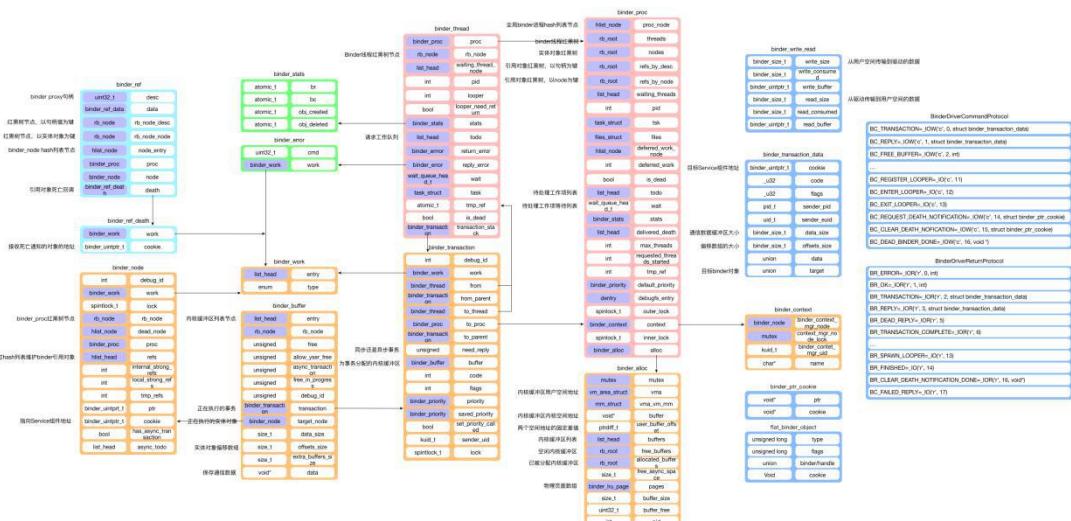
2、客户端持有的 `mHandle` 字段是与内核通信的关键，用以查询到对应的 binder 实体对象与 binder 服务所在进程。通过 `mHandler` 可以查询到 `binder->ref`，再通过 `binder_ref->binder_node->binder_proc` 的链路可以拿到 binder 服务端进程

3、`service_manager` 本质上其实是维护了一个 `handle` 链表，便于客户端通过 `binder` 服务 `name` 进行查询，随后可以获取到 `binder` 代理 `handle`，最终以封装成 `binder` 代理对象

4、内核除 IPC 通信以外的核心功能在于，提供了目前查询、命令分发、请求路由等服务

5、匿名 binder 服务与实名 binder 服务的区别在于，匿名 binder 服务是直接通过 binder 调用在进程之间传输的，而实名 binder 服务需要通过 service_manager 进行注册、随取随用

数据结构鸟瞰



实体型结构：

- binder_proc: binder 进程
- binder_thread: binder 线程
- binder_node: binder 实体对象
- binder_ref: binder 引用对象
- binder_work: binder 事务，是生产与消费者模式队列中的基本对象
- binder_transaction: binder 调用
- binder_buffer: binder 数据传输空间
- binder_ref_death: binder 服务端死亡通知

数据型结构：

- binder_write_read: 用户空间与内核空间交互的数据
- binder_transaction_data: binder_write_read 数据的主体
- flat_binder_objet: binder 对象的封装，便于通信传输

Binder 设计基础

ioctl(): 内核/用户空间调用

ioctl 在 linux 中的使用很常见，也是 binder 驱动能够实现用户空间、内核空间之间通信的基础之一。简而言之，它是设备驱动中对设备的 I/O 通道进行管理的函数，每次调用需要指定三个参数: fd, command 与需要传入到内核的数据。其中 cmd 至关重要，用户将针对设备的命令操作进行分流，素有没有在 binder 驱动中的 ioctl 实现中，看到的是常常 switch-case 条件判断语句

与 binder 内核通信的 ioctl 传入的 fd 是在 IPCThreadState 初始化时，通过调用 open 打开 /dev/binder 文件后获取到的 fd，该打开操作只会进行一次，随后的 binder 调用本质上都是通过 ioctl 向该 fd 所在设备进行通信的

frameworks/native/libs/binder/IPCThreadState.cpp

```
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()),
      mMyThreadId(gettid()),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0)
{
    ...
}
```

frameworks/native/libs/binder/ProcessState.cpp

```
static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR | O_CLOEXEC);
    ...
    return fd;
}
```

这里需要的 open 方法的调用也是需要重点关注下，有了打开也就有了关闭，作为以万物皆文件设计理念著称的 Linux，当进程死亡后，会关闭所有该进程打开过的文件。binder 驱动正是借助于此，当文件 fd 关闭后，驱动会被回调通知，然后产生死亡回调，通知监听过该回调的进程。

mmap(): 内核/用户空间内存映射

内存映射的作用简而言之是将用户控件的一段内存区域映射到内核空间，映射之后，用户再对这段内存进行的读写操作，都会和内核空间的对应内存区域保持一

致，使用 mmap 进行普通用户用户控件与内核空间的映射，是实现进程间通信的物理基础

这里特别提一下 copy_to_user 与 copy_from_user 这两个方法，在 binder 内核中使用的十分频繁，经典用法：

```
static int binder_ioctl_write_read(struct file *filp,
                                    unsigned int cmd, unsigned long arg,
                                    struct binder_thread *thread)
{
    ...
    void __user *ubuf = (void __user *)arg;
    struct binder_write_read bwr;
    ...
    // 从用户空间拷贝数据到内核空间
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
        ret = -EFAULT;
        goto out;
    }
    ...
    // 从内核空间拷贝数据到用户空间
    copy_to_user(ubuf, &bwr, sizeof(bwr))
}
```

有了以上的基础，binder 驱动遍有了"建筑地基"，可以 IPC 通信过程中传输数据

Binder 中的 ONEWAY 与非 ONEWAY 调用

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::transact(int32_t handle,
                                    uint32_t code, const Parcel& data,
                                    Parcel* reply, uint32_t flags)
{
    ...
    if (((flags & TF_ONE_WAY) == 0) {
        ...
    }
```

```

    if (reply) {
        err = waitForResponse(reply);
    } else {
        Parcel fakeReply;
        err = waitForResponse(&fakeReply);
    }
    ...
}
} else {
    err = waitForResponse(NULL, NULL);
}
return err;
}

```

ONE_WAY这个字段的设置,最终提现在transact阶段中,调用waitForResponse方法是否传入了reply实体参数

```

status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;
    while (1) {
        // 调用talkWithDriver与binder driver进行通信
        if ((err= talkWithDriver()) < NO_ERROR) break;
        ...
        cmd = (uint32_t)mIn.readInt32();
        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;
        ...
        case BR_REPLY:
            ...
            goto finish;
        ...
    }
finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }
}
return err;

```

```
}
```

waitForResponse 本质上是一个无限循环，通过 talkWithDriver 与 binder driver 进行通信，根据返回的 command 进行不同的操作

可以看到，如果 reply 字段实体对象不为空的话，在获取到 binder driver 相应成功的命令后，就会跳出循环，进行到 finish

Binder 中的生产者与消费者

Binder 驱动模型中，驱动程序担任着生产者的角色，而 binder 客户端与服务端进程、线程担任着消费者的角色。实现具体中，消费队列体现为在 binder_proc 与 binder_thread 的 todo 中，消费者从这两个队列中获取需要处理的事务，binder

角色	职责	binder 驱动体现
生产者	生产事务	驱动
消费者	消费事务	binder 进程、线程
事务队列	用于存储生产事务，方便消费者取出	binder_proc->todo, binder_thread->todo
事务	用于处理的基本单位	binder_work

驱动中的事务有体现为 binder_work 的实例：

binder_work 的事务消费都是在 binder_thread_read 中进行的，以下对 binder_work 类别进行总结：

Type	场景	涉及队列
BINDER_WORK_TRANSACTION	binder 驱动 对服务端进程进行请求	thread->todo, proc->todo
BINDER_WORK_RETURN_ERROR	binder 驱动 对客户端进程发出错误回复	thread->todo
BINDER_WORK_TRANSACTION_COMPLETE	binder 驱动 对客户端进程的首次回应	thread->todo
BINDER_WORK_NODE	binder 驱动 对实体引用相关操作	thread->todo, proc->todo
BINDER_WORK_DEAD_BINDER	binder 实体 死亡相关操作	thread->todo, proc->todo

这里 Binder 客户端与服务端虽然都是消费者，但是客户端作为调用者，发起调用的只是普通进程，获取到 Binder 内核事务队列的机会只是一次性的，而作为服务端，却是在循环的获取事务队列中的数据，并分配空闲的 Binder 线程进行处理。这里重点关注一下服务端的循环消费。

我们知道，在服务端注册服务时，有两个方法是十分重要的，一个是 ServiceManager.publishService，另外一个是 IPCThreadState.joinThreadPool 方法，这里 joinThreadPool 方法的主要功能就是循环的去获取内核的 binder_work 处理反馈：

frameworks/native/libs/binder/IPCThreadState.cpp

```

void IPCThreadState::joinThreadPool(bool isMain)
{
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
    set_sched_policy(mMyThreadId, SP_FOREGROUND);
    status_t result;
    // 在这里进入循环，循环处理该进程服务的请求
    do {
        processPendingDerefs();
        result = getAndExecuteCommand();
    } while (result != -ECONNREFUSED && result != -EBADF);
    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
status_t IPCThreadState::getAndExecuteCommand()
{
    status_t result;
    int32_t cmd;
    // 请求driver，查看是否有需要处理的事务
    result = talkWithDriver();
    if (result >= NO_ERROR) {
        ...
        result = executeCommand(cmd);
    }
    return result;
}
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    ...
    // talkWithDriver 的核心在于调用 ioctl 与内核进行通信
    // 最终会在内核中调用 binder_thread_read 方法
    // 如果没有获取到需要处理的事务，线程会处于阻塞状态
    if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
        err = NO_ERROR;
    else
        err = -errno;
    ...
    // 通信返回后，说明获取到了需要处理的事务，然后会 bwr 进行处理
}

```

梳理一下思路：

- 内核在生产者消费者之间充当了调和的角色，相当于一个 controller，用来协调两者之间的关系。

- 生产与消费的场景实质上都发生在内核，服务端线程获取到消费事务后才在自己用户空间中进行处理操作
- 进程想要发布服务除了使用 `ServiceManager.addService` 外，务必要调用 `joinThreadPool`，否则无法处理其它进程的请求事务
- 每个调用 `ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)` 的线程，都会在调用的过程中涉及 `binder_thread_write` 与 `binder_thread_read` 这两个方法，其中前者主要做生产者的操作，后者主要做消费者的操作。`binder_thread_read` 主要使用到了 `proc->todo` 与 `thread->todo` 这两个队列，当这两个队列不为空时才会唤起线程，否则会产生阻塞

值得注意的时候，`binder` 的设计很大程度上依赖着 `command` 的顺序，也就意味着生产与消费是一一对应的，否则的话会导致消费顺序错乱的异常，引导意想不到的问题，例如 `ONE_WAY` 的 `binder` 调用居然也会卡死

早起的 `binder` 版本，为了保持数据的一致性，使用了 `global lock` 的方式，很大程度上限制了 `binder` 内核的并发性。最新的 `binder` 驱动中，通过引入三把锁，分离的早期的 `global lock`，大大提高了性能

题外话，我们可能都比较熟悉经典的生产者与消费者在 JAVA 的实现，核心思路就是如果事务队列为空，那么消费者则调用 `wait()` 进入睡眠状态，不占用进程时间片资源，而生产者在插入事务到队列后会调用 `notify()` 来唤起消费者。

在 `binder` 内核中，典型的消费者休眠逻辑：

`msm-3.18/drivers/staging/android/binder.c`

```
static int binder_wait_for_work(struct binder_thread *thread,
                                bool do_proc_work)
{
```

```
...
for (;;) {
    // 调用该方法使进程在等待队列上睡眠
    prepare_to_wait(&thread->wait, &wait, TASK_INTERRUPTIBLE);
    // 当进程被唤醒后, 检查 todo 队列是否不为空, 如果是, 则唤醒进程
    if (binder_has_work_ilocked(thread, do_proc_work))
        break;
    ...
    // 否则的话, 将该线程的调度权交给调度器, 再次进入睡眠
    schedule();
    ...
}
```

典型的生产者逻辑:

```
...
// 将事务插入到 todo 队列中
binder_enqueue_work_ilocked(&t->work, &target_thread->todo);
...
// 随后唤醒等待队列
wake_up_interruptible_sync(&target_thread->wait);
...
```

Binder 代理对象的 handle 句柄

大家在谈及 binder 的设计基础时, 往往说的最多的还是内存映射。的确, 没有内存映射的基础设施, 使用 binder 进行 IPC 通信就是天方夜谭, 然而当我们从程序设计的角度看待这个问题时, 就能发现 binder 内核与 binder 客户端常用到的代理对象 handle 句柄却是贯穿着 IPC 通信的核心所在。

binder 代理对象在客户端的抽象 BinderProxy, 它通信的核心其实就在于调用 transactNative 方法, 通过 JNI 调用到 native 层的 binder 代理对象 BpBinder 来

实现与内核的通信。BinderProxy 从设计的层次来看只是一个通过 JNI 调用到 native 的封装。

frameworks/native/libs/binder/BpBinder.cpp

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive) {
        // 与内核通信
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

这里 BpBinder 实际上也是委托 IPCThreadState 来与内核进行通信的，调用 transact 方法的关键是传入 mHandle 作为 Binder 服务的标志，类似与 HashMap 中的键，以帮助内核查找到对应的实体对象

这里的 mHandle 是在 BpBinder 创建时传入的，而无论是匿名 Binder 还是实名 Binder 服务，对应的 BpBinder 的创建操作是在跨进程传输时，通过调用 Parcel.readStrongBinder 进行的

这里深入到 IPCThreadState->transact 方法中：

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::transact(int32_t handle,
    uint32_t code, const Parcel& data,
```

```
        Parcel* reply, uint32_t flags)
{
    ...
    if (err == NO_ERROR) {
        // 将handle 写入到binder_transaction_data 中
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, N
ULL);
    }
    ...
    if ((flags & TF_ONE_WAY) == 0) {
        ...
        // 最终还是通过调用waitForResponse 来实现通信
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        ...
    } else {
        err = waitForResponse(NULL, NULL);
    }
    ...
}
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlag
s,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;
    tr.target.handle = handle;
    ...
    mOut.writeInt32(cmd);
    // 特殊留意一下, binder_transaction_data 最终被包装到Parcel 类型的mOut 中
    mOut.write(&tr, sizeof(tr));
    return NO_ERROR;
}
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResul
t)
{
    uint32_t cmd;
    int32_t err;
    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        ...
    }
}
```

```

    }
}

status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    binder_write_read bwr;
    ...
    bwr.write_size = outAvail;
    // 获取Parcel的数据，封装到binder_write_read中
    bwr.write_buffer = (uintptr_t)mOut.data();
    ...
    // 最终将binder_write_read发送到内核，其中handle数据就在其中
    if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
        err = NO_ERROR;
    else
        err = -errno;
    ...
}

```

handle 在内核中的表现名称为 desc，是 binder_ref 的成员，当 binder 请求调用到驱动时，典型的获取实体对象代码如下：

msm-3.18/drivers/staging/android/binder.c

```

static void binder_transaction(struct binder_proc *proc,
                                struct binder_thread *thread,
                                struct binder_transaction_data *tr, int reply,
                                binder_size_t extra_buffers_size)
{
    ...
    // 通过使用binder_proc->refs_by_desc 红黑树与目前binder代理对比，获取都对应的
    // binder_ref
    ref = binder_get_ref_olocked(proc, tr->target.handle,
                                 true);
    if (ref) {
        // 随后获取到binder_ref 对应的binder_node，核心获取操作是调用ref->node
        target_node = binder_get_node_refs_for_txn(
            ref->node, &target_proc,
            &return_error);
    }
    ...
}
static struct binder_node *binder_get_node_refs_for_txn(

```

```

        struct binder_node *node,
        struct binder_proc **proc,
        uint32_t *error)
{
    struct binder_node *target_node = NULL;
    binder_node_inner_lock(node);
    if (node->proc) {
        target_node = node;
        ...
        // 通过node 获取target_proc
        *proc = node->proc;
    } else
        *error = BR_DEAD_REPLY;
    binder_node_inner_unlock(node);
    return target_node;
}

```

通过以上代码，我们可以大致分析出发送 binder 调用时，在发送线程通过 IPCThreadState 传入 binder 代理的 handle 到内核后，会通过记录在 binder_proc 的 refs_by_desc 红黑树获取到 binder_ref，随后再通过 binder_ref->binder_node->binder_proc 的调用链获取到实体对象运行在的服务端进程。随后可以将 binder_work 入队到 binder_proc 或 binder_thread 的 todo 队列中，完成剩余的操作

当 binder 调用发送完毕后，如果该调用是非 ONE_WAY 调用，那么在服务进程处理完成后，如果调用回之前的客户端进程，返回处理后的信息呢？

msm-3.18/drivers/staging/android/binder.C

```

static void binder_transaction(struct binder_proc *proc,
                             struct binder_thread *thread,
                             struct binder_transaction_data *tr, int reply,
                             binder_size_t extra_buffers_size)
{
    ...
    // 获取发起调用时，被保存的stack 信息
    in_reply_to = thread->transaction_stack;
}

```

```

// 从 stack 中获取发起调用的线程，也就是返回操作的目标线程
target_thread = binder_get_txn_from_and_acq_inner(in_reply_to);
...
}

```

从上面的代码我们可以看出，当服务端处理完毕后，会返回信息调用到 binder 内核，之所以返回的调用不需要指定 handle，是因为使用了被记录的 from 信息，可以轻松获取到返回目标线程

Binder 内核中的红黑树

binder 驱动中创建了很多方便索引、查询的数据结构，其中最重要的莫过于红黑树。这里总结了具代表性的几个结构，有人常常会发问，为什么 binder_ref 需要两颗树来索引，其中答案很简单，归根结底还是每棵树使用的场景不同，使用不同的键来查询效率不同罢了

红黑树位置	意义	节点位置	使用场景
binder_proc->refs_by_desc	进程中所有使用的代理引用，以 desc 为键	binder_ref->rb_node_node	方便以 handle 进行查询，多用于发送 binder 调用
binder_proc->refs_by_node	进程中所有使用的代理引用，以 node 为键	binder_ref->	方便以 node 查询，多用于增加 node 引用
binder_proc->nodes	进程中所有实	binder_node	方便已服务端

红黑树位置	意义	节点位置	使用场景
	体服务 节点		服务地址查询, 多用于 binder
binder_proc->threads	进程中 所有 binder 线程	binder_thread	对象传输 方便获取 binder 线程, 以 PID 进行 搜索, 使 用范围 很广

辅助功能：实名服务的注册与获取

这里我们以 system_server 注册 AMS 服务为示例来分析 service_manager 是如何完成服务的注册与获取的:

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
public void setSystemProcess() {
    ...
    ServiceManager.addService(Context.ACTIVITY_SERVICE, this, true);
    ...
}
```

可以看到, addService 必需的前两个参数为服务 name 与服务的实体对象, 其中 name 会作为 service_manager 提供查询服务的键值, 而实体对象的地址引用会被封装为 binder_node 保存在 binder kernel 中

frameworks/base/core/java/android/os/ServiceManager.java

```
public static void addService(String name, IBinder service) {
    try {
        getServiceManager().addService(name, service, false);
    } catch (RemoteException e) {
        Log.e(TAG, "error in addService", e);
    }
}
```

实名服务是通过 service_manager 进行查询的，那么 service_manager 的引用该从哪里获取的，这其实是一个先有鸡还是先有蛋的问题，binder 的设计者给出了答案，是先有了"鸡"才有了"蛋"。

通过使用固定不变的 handle 值 0 来表示 service_manager 的引用句柄，以此就可以直接与 binder 内核通信，查询并获取到 service_manager 的引用对象，直接调用 addService，这里的代码就不细分析了，后面 getService 的流程与此类似

frameworks/native/libs/binder/IServiceManager.cpp

```
virtual status_t addService(const String16& name, const sp<IBinder>& service,
                            bool allowIsolated)
{
    Parcel data, reply;
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);
    data.writeStrongBinder(service);
    data.writeInt32(allowIsolated ? 1 : 0);
    // 获取 BpBinder 对象
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
    return err == NO_ERROR ? reply.readExceptionCode() : err;
}
```

通过调用 remote() 方法获取到 BpBinder，时候调用 transact 方法以发起类型为 ADD_SERVICE_TRANSACTION 的 binder 调用

frameworks/native/libs/binder/BpBinder.cpp

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    if (mAlive) {
        // 还是通过IPCThreadState 进行调用，这里的mHandle 句柄为0
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

往下 iPCThreadState 是如何发起调用到内核，内核是如何查找到 service_manager 服务进程并将进程唤醒、进行服务方法调用的，这些细节就不在罗列了，之前的 ioctl 介绍、生产者消费者模型等基础知识已经解释过了。这里直奔 service_manager 进程本身：

frameworks/native/cmds/servicemanager/service_manager.c

```
int svcmgr_handler(struct binder_state *bs,
                    struct binder_transaction_data *txn,
                    struct binder_io *msg,
                    struct binder_io *reply)
{
    uint16_t *s;
    ...
    switch(txn->code) {
        case SVC_MGR_ADD_SERVICE:
            s = bio_get_string16(msg, &len);
            if (s == NULL) {
                return -1;
            }
            handle = bio_get_ref(msg);
            allow_isolated = bio_get_uint32(msg) ? 1 : 0;
            // 进行服务添加
    }
}
```

```

        if (do_add_service(bs, s, len, handle, txn->sender_euid,
                           allow_isolated, txn->sender_pid))
            return -1;
        break;
    }
    ...
}

```

svcmgr_handler 函数是在 service_manager 进程启动后，在循环体中不断被调用到的服务函数。SVC_MGR_ADD_SERVICE 为服务添加的对应 code，最终会在 svclist 中插入对应的 svcinfo 子项

frameworks/native/cmds/servicemanager/service_manager.c

```

int svcmgr_handler(struct binder_state *bs,
                    struct binder_transaction_data *txn,
                    struct binder_io *msg,
                    struct binder_io *reply)
{
    uint16_t *s;
    ...
    switch(txn->code) {
e SVC_MGR_CHECK_SERVICE:
    s = bio_get_string16(msg, &len);
    if (s == NULL) {
        return -1;
    }
    服务查询
    handle = do_find_service(s, len, txn->sender_euid, txn->sender_pid);
    if (!handle)
        break;
    bio_put_ref(reply, handle);
    return 0;
}
...
}

```

getService 的调用流程和 addService 的调用流程大致类似，主要的区别有两点：

1. 调用的进程不同，`addService` 在服务端进行调用，而 `getService` 在客户端中进行调用
2. 在 `service_manager` 中的执行命令不同，前者会在列表中加入子项，后者查询列表中的子项

核心功能：跨进程数据传输

这里以 `AMS.getRecentTasks` 方法为实例分析 binder 跨进程数据传输，这个方法常常被 `SystemUI` 用于最近任务卡片的获取，它的调用需要向 `AMS` 传递三个基本数据参数，`AMS` 处理完成后回返回一个列表。因为调用是非 `ONE_WAY` 的，所以该数据传输是双工的

`frameworks/base/core/java/android/app/ActivityManagerNative.java`

```
public ParceledListSlice<ActivityManager.RecentTaskInfo> getRecentTasks(int ma
xNum, int flags, int userId) throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeInt(maxNum);
    data.writeInt(flags);
    data.writeInt(userId);
    // 这里的mRemote 其实是 BinderProxy 的实例，通过 transact 调用与 binder 内核进行
    // 通信
    mRemote.transact(GET_RECENT_TASKS_TRANSACTION, data, reply, 0);
    reply.readException();
    final ParceledListSlice<ActivityManager.RecentTaskInfo> list = ParceledLis
tSlice.CREATOR
        .createFromParcel(reply);
    data.recycle();
    reply.recycle();
    return list;
}
```

这里的调用流程和上述类似，大概就是
BinderProxy->BpBinder->IPCThreadState，这里特别留意一下传输数据的封装，
直接看 IPCThreadState 的相关代码：

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::transact(int32_t handle,
                                   uint32_t code, const Parcel& data,
                                   Parcel* reply, uint32_t flags)
{
    ...
    if (err == NO_ERROR) {
        LOG_ONEWAY("">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
                   (flags & TF_ONE WAY) == 0 ? "READ REPLY" : "ONE WAY");
        // 将数据进行封装
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, N
ULL);
    }
    ...

    if ((flags & TF_ONE WAY) == 0) {
        // 非ONE WAY通信，传入 reply 参数，意味着服务端回返回数据
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
    }
    ...
}

status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
                                              int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;
    ...
    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
```

```

        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(binder_size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    }
    ...
    // 将Parcel数据封装成binder_transaction_data，其中包装了命令与数据内容
    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));
    return NO_ERROR;
}

```

注意一下，需要发送给 binder 驱动的数据，此时已经被包装到 mOut 成员变量中

```

status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    uint32_t cmd;
    int32_t err;

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        ...
    }
    ...
}

status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    binder_write_read bwr;
    ...
    bwr.write_size = outAvail;
    // 获取mOut中的数据
    bwr.write_buffer = (uintptr_t)mOut.data();
    ...
    // 调用ioctl和内核通信，通信数据为binder_write_read
    if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
        err = NO_ERROR;
    ...
}

```

先调用 talkWithDriver 与内核通信

msm-3.18/drivers/staging/android/binder.c

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    // proc 引用保存在file 的private_data 引用中
    struct binder_proc *proc = filp->private_data;
    ...
    switch (cmd) {
        // command 为BINDER_WRITE_READ, 进入该代码块
        case BINDER_WRITE_READ:
            ret = binder_ioctl_write_read(filp, cmd, arg, thread);
            if (ret)
                goto err;
            break;
        ...
    }

    static int binder_ioctl_write_read(struct file *filp,
                                      unsigned int cmd, unsigned long arg,
                                      struct binder_thread *thread)
    {
        struct binder_write_read bwr;
        // 从用户空间中拷贝数据到内核空间
        if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
            ret = -EFAULT;
            goto out;
        }
        ...
        if (bwr.write_size > 0) {
            // write_size 大于0 时, 调用该方法进行事务生产
            ret = binder_thread_write(proc, thread,
                                      bwr.write_buffer,
                                      bwr.write_size,
                                      &bwr.write_consumed);
            ...
        }
        if (bwr.read_size > 0) {
            // read_size 大于0, 调用该方法进行事务消费
            ret = binder_thread_read(proc, thread, bwr.read_buffer,
                                      bwr.read_size,
                                      &bwr.read_consumed,
        }
    }
}
```

```
        filp->f_flags & O_NONBLOCK);

        ...
    }

    ...

// 将内核空间处理过的 bwr 拷贝到用户空间
if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
    ret = -EFAULT;
    goto out;
}

...

}

static int binder_thread_write(struct binder_proc *proc,
                               struct binder_thread *thread,
                               binder_uintptr_t binder_buffer, size_t size,
                               binder_size_t *consumed)
{
    switch (cmd) {
        ...
        case BC_TRANSACTION:
        case BC_REPLY: {
            struct binder_transaction_data tr;

            // 将封装好的数据拷贝到内核空间
            if (copy_from_user(&tr, ptr, sizeof(tr)))
                return -EFAULT;
            ptr += sizeof(tr);
            binder_transaction(proc, thread, &tr,
                               cmd == BC_REPLY, 0);
            break;
        }
        ...
    }
}

static void binder_transaction(struct *proc,
                               struct binder_thread *thread,
                               struct binder_transaction_data *tr, int reply,
                               binder_size_t extra_buffers_size)
{
    struct binder_proc *target_proc = NULL;
    struct binder_thread *target_thread = NULL;
    ...

    if (tr->target.handle) {
        struct binder_ref *ref;
```

```

        binder_proc_lock(proc);
        // 使用handle 获取到binder_ref
        ref = binder_get_ref_olocked(proc, tr->target.handle, true);
        if (ref) {
            // 通过binder_ref 获取binder_node 与target_proc
            target_node = binder_get_node_refs_for_txn(
                ref->node, &target_proc, &return_error);
        }
        binder_proc_unlock(proc);
    }
    ...
    // 注意这里binder_work 的类型
    t->work.type = BINDER_WORK_TRANSACTION;
    ...
    // 如果此次通信为非ONE_WAY 则需要向目标进程发起请求并等待
    if (!(t->flags & TF_ONE_WAY)) {
        binder_inner_proc_lock(proc);
        t->need_reply = 1;
        t->from_parent = thread->transaction_stack;
        // 将此次事务保存在线程的transaction_stack 中
        thread->transaction_stack = t;
        binder_inner_proc_unlock(proc);
        if (!binder_proc_transaction(t, target_proc, target_thread)) {
            ...
            goto err_dead_proc_or_thread;
        }
    }
}

static bool binder_proc_transaction(struct binder_transaction *t,
                                    struct binder_proc *proc,
                                    struct binder_thread *thread)
{
    ...
    // 在这里将binder_work 入队，这样目标进程就可以解除阻塞，开始进行事务处理
    binder_enqueue_work_ilocked(&t->work, target_list);
}

```

这里梳理一下发送 binder 请求的流程：

- 调用 BinderProxy JNI 方法到 native，封装需要传输的数据

- 调用 `BpBinder.transact()`方法，继续向下委托
- 调用 `IPCThreadState.talkWithDriver`，继续封装数据
- 最终调用 `ioctl` 方法与内核进行通信
- 内核空间中，调用 `binder_ioctl`，分流命令
- 通过 `handle` 找到 `binder_ref`，最终找到 `binder_proc`
- 将类型为 `BINDER_WORK_TRANSACTION` 的 `binder_work` 入队到 `binder_proc->todo` 队列中

插入事务到 `todo` 队列中之后，接下来的分析就到了目前服务所在进程的空间下了：

```

static int binder_thread_read(struct binder_proc *proc,
                             struct binder_thread *thread,
                             binder_uintptr_t binder_buffer, size_t size,
                             binder_size_t *consumed, int non_block)
{
    ...
    while (1) {
        uint32_t cmd;
        ...
        w = binder_dequeue_work_head_ilocked(list);
        case BINDER_WORK_TRANSACTION: {
            binder_inner_proc_unlock(proc);
            // 通过 container_of 宏获取到 binder_transaction 实例
            t = container_of(w, struct binder_transaction, work);
            } break;

        ...
        if (t->buffer->target_node) {
            // 获取到目标进程服务相关信息
            struct binder_node *target_node = t->buffer->target_node;
            struct binder_priority node_prio;

            tr.target.ptr = target_node->ptr;
            tr.cookie = target_node->cookie;
            ...
        }
    }
}

```

```

        // 留意一下这里返回给用户空间所在进程的命令
        cmd = BR_TRANSACTION;
    }

    ...

    // 将目标进程服务地址与所需处理的数据从内核空间传到用户空间
    if (put_user(cmd, (uint32_t *)__user *)ptr)) {
        ...
        return -EFAULT;
    }

    ptr += sizeof(uint32_t);
    if (copy_to_user(ptr, &tr, sizeof(tr))) {
        ...
        return -EFAULT;
    }
}

```

上述代码运行在内核空间中，当执行完毕后回返回到用户空间，这里直接分析
IPCThreadState.executeCommand 方法：

frameworks/native/libs/binder/IPCThreadState.cpp

```

status_t IPCThreadState::executeCommand(int32_t cmd)
{
    switch ((uint32_t)cmd) {
        ...
        case BR_TRANSACTION:
        {
            ...
            // 获取到目标服务所在地址
            if (tr.target.ptr) {
                // 调用transact 方法，执行服务端方法调用
                error = reinterpret_cast<BBinder*>(tr.cookie)->transact(tr.code, bu
ffer, &reply, tr.flags);
            }
            ...
        }
        ...
    }
}

```

tr.cookie 会被转换为 BBinder 对象，它是服务端的实体对象

frameworks/native/libs/binder/Binder.cpp

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
        ...
        default:
            // 调用该方法进行进一步的事务分发
            err = onTransact(code, data, reply, flags);
            break;
    }
    ...

    return err;
}
```

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
@Override
public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws RemoteException {
    switch (code) {
        ...
        case GET_RECENT_TASKS_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            int maxNum = data.readInt();
            int fl = data.readInt();
            int userId = data.readInt();
            // 最终执行服务方法
            ParceledListSlice<ActivityManager.RecentTaskInfo> list = getRecentTasks(maxNum,
                fl, userId);
            reply.writeNoException();
            // 将结果列表写入到Parcel实例reply中
            list.writeToParcel(reply, Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
        }
        return true;
    }
    ...
}
```

```
    }
}
```

到这里，服务端的对应方法就执行完毕了，但是对于非 ONE_WAY 的方法，需要返回处理结果给发出该请求的进程

frameworks/native/libs/binder/IPCThreadState.cpp

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    switch ((uint32_t)cmd) {
        ...
        case BR_TRANSACTION:
        {
            ...
            if ((tr.flags & TF_ONE WAY) == 0) {
                // 调用该方法返回结果，需要经过binder驱动的中转
                sendReply(reply, 0);
            }
            ...
        }
        ...
    }
}

status_t IPCThreadState::sendReply(const Parcel& reply, uint32_t flags)
{
    status_t err;
    status_t statusBuffer;
    // 写入返回的数据到mOut成员中，command命令为BC_REPLY
    err = writeTransactionData(BC_REPLY, flags, -1, 0, reply, &statusBuffer);
    if (err < NO_ERROR) return err;

    // 调用该方法与binder通信，注意这个方法是ONE WAY的
    ForResponse(NULL, NULL);
}
```

返回的流程与发出请求的流程大致相同，主要区别在于 cmd 命令不同，这里直接看 binder_transaction 的相关实现：

```
static void binder_transaction(struct *proc,
```

```

        struct binder_thread *thread,
        struct binder_transaction_data *tr, int reply,
        binder_size_t extra_buffers_size)
{
    ...
    if (reply) {
        // 通过获取到与binder线程关联的事务获取到发起请求的线程
        in_reply_to = thread->transaction_stack;
        ...
        target_thread = binder_get_txn_from_and_acq_inner(in_reply_to);
    }
    ...
    // 之后的操作同样是将binder_work插入到todo队列中，唤醒请求端进程进行处理
}

```

辅助功能：匿名服务的跨进程传输与回调

这一小节的原理与内容其实与上一节高度相似，这里单独分析的原因是 Android 跨进程通信中，对于匿名服务的使用十分广泛，但是这种写法往往又十分另新手困惑。例如常见的四大组件，无不与匿名服务紧密关联，这里以动态广播注册要讲解匿名服务的传输与回调

```

public Intent registerReceiver(IApplicationThread caller, String packageName,
                                IIntentReceiver receiver,
                                IntentFilter filter, String perm, int userId) throws
RemoteException
{
    // 这里IApplicationThread caller 其实也是匿名服务，这里先忽略
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    ...
    // receiver为一个动态创建的匿名服务IIntentReceiver实例
    data.writeStrongBinder(receiver != null ? receiver.asBinder() : null);
    ...
    mRemote.transact(REGISTER_RECEIVER_TRANSACTION, data, reply, 0);
    reply.readException();
    ...
    reply.recycle();
    data.recycle();
    return intent;
}

```

receiver 实例将回在 writeStrongBinder 中被封装，随后传输到 binder 驱动，驱动在中转期间会取出 binder 对象并进行实体对象、引用对象的创建，最终会将引用对象传递到 binder 服务端，服务端只需要将它进行保存，就可以随时进行调用以实现回调

frameworks/native/libs/binder/Parcel.cpp

```
status_t Parcel::writeStrongBinder(const sp<IBinder>& val)
{
    return flatten_binder(ProcessState::self(), val, this);
}

status_t flatten_binder(const sp<ProcessState>& /*proc*/,
    const sp<IBinder>& binder, Parcel* out)
{
    // 封装的对象
    flat_binder_object obj;

    obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
    if (binder != NULL) {
        IBinder *local = binder->localBinder();
        if (!local) {
            ...
        } else {
            // 当前为本地服务对象，将会记录地址并将类型设置为BINDER_TYPE_BINDER
            obj.type = BINDER_TYPE_BINDER;
            obj.binder = reinterpret_cast<uintptr_t>(local->getWeakRefs());
            obj.cookie = reinterpret_cast<uintptr_t>(local);
        }
    }

    return finish_flatten_binder(binder, obj, out);
}
```

中间的分析和之前一致，不再冗余了，直接到 binder_transaction 函数中

msm-3.18/drivers/staging/android/binder.c

```

static void binder_transaction(struct proc,
                             struct binder_thread *thread,
                             struct binder_transaction_data *tr, int reply,
                             binder_size_t extra_buffers_size)
{
    ...
    // 传输过程中会涉及 binder 对象的检查，如果存在则需要进行解析
    hdr = (struct binder_object_header *) (t->buffer->data + *offp);
    off_min = *offp + object_size;
    switch (hdr->type) {
        case BINDER_TYPE_HANDLE:
        case BINDER_TYPE_WEAK_HANDLE: {
            struct flat_binder_object *fp;

            fp = to_flat_binder_object(hdr);
            ret = binder_translate_handle(fp, t, thread);
            if (ret < 0) {
                return_error = BR_FAILED_REPLY;
                return_error_param = ret;
                return_error_line = __LINE__;
                goto err_translate_failed;
            }
        } break;
    }
    ...
}

static int binder_translate_binder(struct flat_binder_object *fp,
                                   struct binder_transaction *t,
                                   struct binder_thread *thread)
{
    // 如果实体对象之前并未创建，则会进行创建操作
    node = binder_get_node(proc, fp->binder);
    if (!node) {
        node = binder_new_node(proc, fp);
        if (!node)
            return -ENOMEM;
    }
    ...
    // 接下来的操作将会增加实体对象的引用
}

```

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
@Override  
public boolean onTransact(int code, Parcel data, Parcel reply, int flags) throws  
RemoteException {  
    switch (code) {  
        ...  
        case REGISTER_RECEIVER_TRANSACTION:  
        {  
            data.enforceInterface(IActivityManager.descriptor);  
            ...  
            // 调用到AMS 接收端，回调用readStrongBinder 进行binder 对象的解析  
            b = data.readStrongBinder();  
            IIntentReceiver rec  
                = b != null ? IIntentReceiver.Stub.asInterface(b) : null;  
            ...  
            Intent intent = registerReceiver(app, packageName, rec, filter, per  
m, userId);  
            ...  
            return true;  
        }  
        ...  
    }  
}
```

readStrongBinder 本质上是调用了 Parcel.java 中的 JNI 方法

frameworks/base/core/jni/android_os_Parcel.cpp

```
static jobject android_os_Parcel_readStrongBinder(JNIEnv* env, jclass clazz, j  
long nativePtr)  
{  
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);  
    if (parcel != NULL) {  
        // 通过该核心方法，从内核中读取出binder 对象  
        return javaObjectForIBinder(env, parcel->readStrongBinder());  
    }  
    return NULL;  
}
```

frameworks/native/libs/binder/Parcel.cpp

```
sp<IBinder> Parcel::readStrongBinder() const  
{
```

```

    sp<IBinder> val;
    readStrongBinder(&val);
    return val;
}

status_t Parcel::readStrongBinder(sp<IBinder>* val) const
{
    return unflatten_binder(ProcessState::self(), *this, val);
}

status_t unflatten_binder(const sp<ProcessState>& proc,
    const Parcel& in, sp<IBinder>* out)
{
    const flat_binder_object* flat = in.readObject(false);

    if (flat) {
        switch (flat->type) {
            case BINDER_TYPE_BINDER:
                ...
            case BINDER_TYPE_HANDLE:
                // 因为这里处于进行读取端的进程, 所以这时类型为HANDLE
                // 同理通过传入过来的句柄来创建BpBinder 对象
                // out 也是将回返回给 javaObjectForIBinder 进一步处理的引用对象
                *out = proc->getStrongProxyForHandle(flat->handle);
                return finish_unflatten_binder(
                    static_cast<BpBinder*>(out->get()), *flat, in);
        }
    }
    return BAD_TYPE;
}

```

最终会在 native 中创建 java 层代理对象

frameworks/base/core/jni/android_util_Binder.cpp

```

jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
    ...
    object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mC
onstructor);
    ...
}

```

返回到 java 层，会再调用 `IIntentReceiver.Stub.asInterface(b)` 方法对 `BinderProxy` 进行深一步封装，最终 AMS 会对 `IBinder` 对象进行存储并作为主要数据结构的键，方便客户端进行索引。当需要回调到客户端进程时，可以随时使用传入的代理对象进行实现

辅助功能：死亡回调的注册与获取

死亡回调是 binder 架构中极其重要的一个功能，它的意义在于，当服务端进行死亡后，可以告知各个客户端进程、进行一些反注册的注销操作。死亡回调的核心实现原理在于，当服务端进程死亡后，内核会对它之前打开过的`/dev/binder` 文件进行关闭，随后 binder 内核会收到调用，并对注册了死亡监听的客户端进程们进行进一步的回调

以下从 java 层死亡回调的注册开始分析：

`frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java`

```
try {
    AppDeathRecipient adr = new AppDeathRecipient(
        app, pid, thread);
    // 获取 BinderProxy 进行死亡回调的注册
    thread.asBinder().linkToDeath(adr, 0);
    app.deathRecipient = adr;
} catch (RemoteException e) {
    ...
    // 注册异常
}
```

`BinderProxy` 的 `linkToDeath` 与 `transact` 一样，委托给了 native 层进行实现，最终会调用到 `BpBinder`

frameworks/native/libs/binder/BpBinder.cpp

```
status_t BpBinder::linkToDeath(
    const sp<DeathRecipient>& recipient, void* cookie, uint32_t flags)
{
    Obituary ob;
    ob.recipient = recipient;
    ob.cookie = cookie;
    ob.flags = flags;

    {
        AutoMutex _l(mLock);

        if (!mObitsSent) {
            if (!mObituaries) {
                // 进程可以注册单个binder 服务的多个死亡回调
                mObituaries = new Vector<Obituary>;
                if (!mObituaries) {
                    return NO_MEMORY;
                }
                getWeakRefs()->incWeak(this);
                IPCThreadState* self = IPCThreadState::self();
                // 调用该方法进行注册
                self->requestDeathNotification(mHandle, this);
                // 执行与binder 驱动的通信
                self->flushCommands();
            }
            ssize_t res = mObituaries->add(ob);
            return res >= (ssize_t)NO_ERROR ? (status_t)NO_ERROR : res;
        }
    }

    return DEAD_OBJECT;
}
```

msm-3.18/drivers/staging/android/binder.c

```
static int binder_thread_write(struct binder_proc *proc,
    struct binder_thread *thread,
    binder_uintptr_t binder_buffer, size_t size,
    binder_size_t *consumed)
{
    ...
}
```

```

switch (cmd) {
    case BC_REQUEST_DEATH_NOTIFICATION:
    case BC_CLEAR_DEATH_NOTIFICATION: {
        // 在内核中创建对应的binder_ref_death
        death = kzalloc(sizeof(*death), GFP_KERNEL);
        ...
        // 引用到binder_ref 中进行保存
        ref->death = death;
    }
}
}

```

至此，注册操作大致完成，接下来继续看死亡回调是如何从内核调用到客户端的

当/dev/binder 文件被关闭后，会调用实现注册的 binder_release 方法

msm-3.18/drivers/staging/android/binder.c

```

static int binder_release(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc = filp->private_data;

    // 进行进程相关资源销毁，之后会调用到实体对象的资源释放
    binder_defer_work(proc, BINDER_DEFERRED_RELEASE);

    return 0;
}

static int binder_node_release(struct binder_node *node, int refs)
{
    ...
    // 遍历每个实体对象的引用对象
    hlist_for_each_entry(ref, &node->refs, node_entry) {
        refs++;

        binder_inner_proc_lock(ref->proc);
        if (!ref->death) {
            binder_inner_proc_unlock(ref->proc);
            continue;
        }

        death++;
    }
}

```

```

    BUG_ON(!list_empty(&ref->death->work.entry));
    // 将binder_work类型置为BINDER_WORK_DEAD_BINDER
    ref->death->work.type = BINDER_WORK_DEAD_BINDER;
    // 将binder_work放入到binder_ref所在进程的todo队列中
    binder_enqueue_work_ilocked(&ref->death->work,
                                &ref->proc->todo);
    binder_wakeup_proc_ilocked(ref->proc);
    binder_inner_proc_unlock(ref->proc);
}
...
}

```

读取的操作与之前的数据传输操作是类似的，都是在 `binder_thread_read` 中进行的

```

static int binder_thread_read(struct binder_proc *proc,
                             struct binder_thread *thread,
                             binder_uintptr_t binder_buffer, size_t size,
                             binder_size_t *consumed, int non_block)
{
    ...
    switch (w->type) {
        case BINDER_WORK_DEAD_BINDER:
        case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
        case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
            ...
            // 获取到binder_ref_death对象
            death = container_of(w, struct binder_ref_death, work);
            ...
            // 之后回到用户空间，进行回调操作
        }
    }
    ...
}

```

可以看到binder的死亡调用的代码复杂度其实比核心功能数据传输还是高一些，但是这两者的设计原理其实大体一致，一旦掌握了核心套路，代码的流转就很好把握

附：内核基础知识

链表: list_head

msm-3.18/include/linux/types.h

```
struct list_head {
    struct list_head *next, *prev;
};
```

双向链表是最常见的数据结构之一，在 binder 驱动中用以实现 todo 队列

方法名	作用
INIT_LIST_HEAD	初始化链表
list_add	插入节点
list_del	删除节点
list_replace	替换节点
list_empty	链表是否为空

通过判断 todo 链表队列是否为空，来决定是否在 binder_thread_read 中唤醒进程：

```
static bool binder_has_work_ilocked(struct binder_thread *thread,
                                    bool do_proc_work)
{
    return !binder_worklist_empty_ilocked(&thread->todo) ||
           thread->looper_need_return ||
           (do_proc_work &&
            !binder_worklist_empty_ilocked(&thread->proc->todo));
}
```

散列表: hlist_head

msm-3.18/include/linux/types.h

```

    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev;
};

```

与 list_head 双指针表头双链表的设计相比, hlist 为单指针表头双链表, hlist_head 内部仅有一个指针指向第一个节点, 该结构适于用在 HASH 表中, 在链表泛滥的 HASH 表中, 相比 list_head 节省了许多空间损耗

典型遍历过程:

```

hlist_for_each_entry(itr, &binder_procs, proc_node) {
    if (itr->pid == pid) {
        seq_puts(m, "binder proc state:\n");
        print_binder_proc(m, itr, 1);
    }
}

```

红黑树: rb_root

msm-3.18/include/linux/rbtree.h

```

struct rb_node {
    unsigned long __rb_parent_color; // 父节点的颜色
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));

struct rb_root {
    struct rb_node *rb_node;
};

```

红黑树的特性:

1. 每个节点要么是黑色, 要么是红色
2. 根节点是黑色
3. 每个为空的叶子节点是黑色

4. 如果一个节点为红色，那么它的子节点必须是黑色
5. 从一个节点到该节点的子孙节点的所有路径上，包含相同数目的黑节点

由以上特性可以证明出，红黑树的高度至多为: $2\log(n+1)$

依我的理解，红黑树的查找效率虽然没有 AVL 树高，但是其插入、删除效率更高，是 AVL 树于 BST 实现的折中数据结构

核心方法：

方法名	作用	时间复杂度
rb_insert_color	插入节点	O(1)
rb_erase	删除节点	O(1)
rb_entry	获取持有节点的实例	O(1)
rb_first	获取 root 节点	O(1)
rb_last	获取下一个节点	O(1)
rb_next	获取上一个节点	O(1)

典型遍历过程：

```

for (n = rb_first(&proc->refs_by_desc); n != NULL; n = rb_next(n)) {
    ref = rb_entry(n, struct binder_ref, rb_node_desc);
    if (ref->data.desc > new_ref->data.desc)
        break;
    new_ref->data.desc = ref->data.desc + 1;
}

```

典型查找过程：

```

p = &proc->refs_by_desc.rb_node; // 获取到红黑树root 节点

```

```
while (*p) {
    parent = *p;
    // 原理是使用了container_of宏，通过红黑树节点，获取持有该节点的binder_ref实例
    ref = rb_entry(parent, struct binder_ref, rb_node_desc);

    // 查询过程
    if (new_ref->data.desc < ref->data.desc)
        p = &(*p)->rb_left;
    else if (new_ref->data.desc > ref->data.desc)
        p = &(*p)->rb_right;
    else
        BUG();
}
```

17.深入解析 JNI 源码

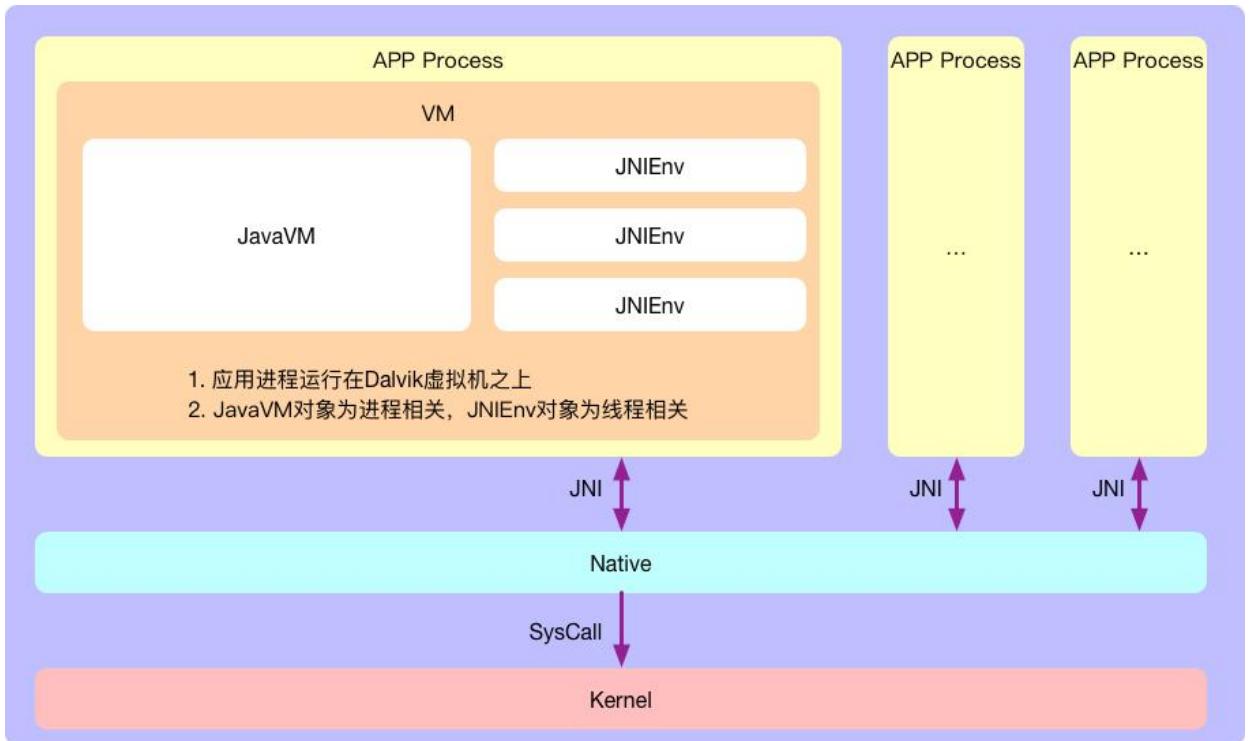
简介

Android NDK 开发中，常常因为效率、安全等原因，需要借助 JNI 的机制调用到 Native 环境中进行 c/cpp 操作，常见的 Java 层需要调用 Native 层的代码时的标准流程是这样的：

1. 调用 `loadLibrary`，依靠 `d1xxx` 系列方法加载动态链接库，然后调用库中的 `JNI_Onload` 方法，解析并保存头文件中的符号表
2. JAVA 层调用事先声明的 `native` 方法
3. 虚拟机通过预先加载的符号表调用 Native 层代码，同时会实例化一个 `JNIEnv` 指针，Native 层可以借用它调用 JAVA 层的代码

这篇文章里，我将会提供典型的 JAVA 与 Native 使用 JNI 接口实现双向调用的实例，同时也会从源码的角度分析两种 JNI 动态注册的原理

架构图



- Android 应用进程各自运行在专属的虚拟机中
- 通过 JNIEnv, 进程可以在 Native 环境中使用 JNI 接口对 JAVA 层进行方法调用等操作
- JNI 技术是 JAVA 与 Native 之间相互调用的"桥梁"
- 系统 SysCall 机制, Native 层可以调用到 Kernel 层

示例

下面的示例中分别会给出从 JAVA 调用到 Native、从 Native 调用到 JAVA 的两个典型示例

在 **JAVA** 中调用 **Native** 方法

- 从 JAVA 调用到 Native 的第一步是需要创建 JAVA project，并进行 native 方法声明与调用

```
public class HelloJni extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_jni);
        TextView tv = (TextView) findViewById(R.id.hello_textview);
        // 通过JNI 调用native 方法
        tv.setText(stringFromJNI());
    }

    // 声明native 方法
    public native String stringFromJNI();

    // 初始化动态so 库
    static {
        System.loadLibrary("hello-jni");
    }
}
```

- 在 cpp 中实现 native 方法

Java_com_example_hellojni_HelloJni_stringFromJNI

```
#include <jni.h>
#include <string>
#include "format.h"

extern "C" {

    JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv * env, jobject obj )
    {
        std::string hello = fmt::format(
            "Hello from C++ Format! GCC version: {}.{}", __GNUC__, __GNUC_MINOR__);
        return env->NewStringUTF(hello.c_str());
    }
}
```

和使用 `RegisterMethodsOrDie` 方法进行注册不同，这个示例使用了 `JNIEXPORT` 和 `JNICALL` 的命名规则进行 native 方法的初始化，通过这种特殊的规则，虚拟机可以在不主动调用注册方法的前提下进行静态注册

1. 使用 makefile 编译动态 so 库

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

$(call import-add-path,$(LOCAL_PATH))

LOCAL_MODULE      := hello-jni
LOCAL_SRC_FILES   := hello-jni.cpp
LOCAL_WHOLE_STATIC_LIBRARIES := cppformat_static

include $(BUILD_SHARED_LIBRARY)

$(call import-module,../../cppformat)
```

在 Native 中调用 JAVA 方法

1. 声明一个类及方法

```
public class JniHandle {
    public String getStringForJava() {
        return "string from method in java";
    }
}
```

1. 在 Native 中使用 JNI 调用 `getStringForJavaJAVA` 方法

```
JNIEXPORT void JNICALL
Java_com_example_hellojni_callJavaMethodFromJni(JNIEnv *env, jclass type) {
    // 通过 JNIEnv 获取到 jclass
    jclass jniHandle = (*env)->FindClass(env, "com/example/JniHandle");
    if (NULL == jniHandle) {
        LOGW("can't find jniHandle");
        return;
}
```

```

jmethodID constructor = (*env)->GetMethodID(env, jniHandle, "<init>", "()V");
if (NULL == constructor) {
    LOGW("can't constructor JniHandle");
    return;
}
// 创建一个 JniHandle 的实例
jobject jniHandleObject = (*env)->NewObject(env, jniHandle, constructor);
if (NULL == jniHandleObject) {
    LOGW("can't new JniHandle");
    return;
}
// 通过 JNIEnv 获取到 jmethod
jmethodID getStringForJava = (*env)->GetMethodID(env, jniHandle, "getStringForJava", "()Ljava/lang/String;");
if (NULL == getStringForJava) {
    LOGW("can't find method of getStringForJava");
    (*env)->DeleteLocalRef(env, jniHandle);
    (*env)->DeleteLocalRef(env, jniHandleObject);
    return;
}
// 调用该 JAVA 方法
jstring result = (*env)->CallObjectMethod(env, jniHandleObject, getStringForJava);
const char *resultChar = (*env)->GetStringUTFChars(env, result, NULL);
// 释放局部引用
(*env)->DeleteLocalRef(env, jniHandle);
(*env)->DeleteLocalRef(env, jniHandleObject);
(*env)->DeleteLocalRef(env, result);
}

```

例子中我给出了静态注册的常规写法，下面详细分析下动态注册在 Android 源码中的两种常用场景：开机 JNI 初始化与调用 `System.loadLibrary` 加载动态库

开机 JNI 初始化

安卓系统开机过程中，会先启动 Init 进程，随后再拉起 zygote 进程。我们知道所有的 APP 应用进程都是通过 fork zygote 进程创建的，所以 zygote 进程在启

动的过程中做了虚拟机初始化的操作，这其中就包括了 framework 所需要的所有 JNI 接口的注册

frameworks/base/cmds/app_process/app_main.cpp

```
int main(int argc, char* const argv[])
{
    AppRuntime runtime(argv[0], computeArgBlockSize(argc, argv));
    ...
    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
    }
    ...
}
```

在 init 拉起 zygote 后，会运行 zygote 的可执行程序 app_main

frameworks/base/core/jni/AndroidRuntime.cpp

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options, bool zygote)
{
    ...
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }
    ...
}

// 声明函数
extern int register_android_os_MessageQueue(JNIEnv* env);

static const RegJNIRec gRegJNI[] = {
    ...
    // 初始化函数指针
    REG_JNI(register_android_os_MessageQueue),
    ...
}
```

```

/*static*/ int AndroidRuntime::startReg(JNIEnv* env)
{
    ...
    // 注册framework 所需要的JNI 接口
    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    ...

    return 0;
}

static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv* env)
{
    for (size_t i = 0; i < count; i++) {
        // 触发对应的注册函数
        if (array[i].mProc(env) < 0) {
            return -1;
        }
    }
    return 0;
}

```

可以看到这个 JNI 注册的核心其实就是遍历 RegJNIRec 数组并触发

RegJNIRec.mProc 的调用

gRegJNI 数组声明中频繁的使用到了 REG_JNI 这个宏，这里特别的看一下它的实现：

```

#define REG_JNI(name)      { name }

struct RegJNIRec {
    int (*mProc)(JNIEnv*);
};

```

REG_JNI 的声明使用了填充结构，这里以 register_android_os_MessageQueue 为例，实际上是让 mProc 指向了它的函数指针，当调用 register_jni_procs 时，

会触发 `register_android_os_MessageQueue` 这个 `extern` 函数，当程序执行这个它时，会自动调用到 `android_os_MessageQueue` 下的对应函数：

frameworks/base/core/jni/android_os_MessageQueue.cpp

```
#include "core_jni_helpers.h"

static const JNINativeMethod gMessageQueueMethods[] = {
    /* name, signature, funcPtr */
    { "nativeInit", "(J)", (void*)android_os_MessageQueue_nativeInit },
    { "nativeDestroy", "(J)V", (void*)android_os_MessageQueue_nativeDestroy },
    { "nativePollOnce", "(JI)V", (void*)android_os_MessageQueue_nativePollOnce },
    { "nativeWake", "(J)V", (void*)android_os_MessageQueue_nativeWake },
    { "nativeIsPolling", "(J)Z", (void*)android_os_MessageQueue_nativeIsPolling },
    { "nativeSetFileDescriptorEvents", "(JII)V",
        (void*)android_os_MessageQueue_nativeSetFileDescriptorEvents}
};

int register_android_os_MessageQueue(JNIEnv* env) {
    // 将JNI方法与对应的android.os.MessageQueue java类中的native方法进行绑定
    int res = RegisterMethodsOrDie(env, "android/os/MessageQueue", gMessageQueueMethods, NELEM(gMessageQueueMethods));
    ...
    return res;
}
```

大致来看，各个 `register_xxx` 方法其实都是调用到了在 `core_jni_helpers.h` 头文件中声明的 `RegisterMethodsOrDie` 内联方法：

frameworks/base/core/jni/core_jni_helpers.h

```
static inline int RegisterMethodsOrDie(JNIEnv* env, const char* className, const JNINativeMethod* gMethods, int numMethods) {
    int res = AndroidRuntime::registerNativeMethods(env, className, gMethods,
numMethods);
    LOG_ALWAYS_FATAL_IF(res < 0, "Unable to register native methods.");
    return res;
}
```

frameworks/base/core/jni/AndroidRuntime.cpp

```
/*static*/ int AndroidRuntime::registerNativeMethods(JNIEnv* env,
    const char* className, const JNINativeMethod* gMethods, int numMethods)
{
    return jniRegisterNativeMethods(env, className, gMethods, numMethods);
}
```

AndroidRuntime 实际上是调用了 JNIHelp 中外联的 jniRegisterNativeMethods 方法

libnativehelper/JNIHelp.cpp

```
extern "C" int jniRegisterNativeMethods(C_JNIEnv* env, const char* className,
const JNINativeMethod* gMethods, int numMethods)
{
    JNIEnv* e = reinterpret_cast<JNIEnv*>(env);

    scoped_local_ref<jclass> c(env, findClass(env, className));
    if (c.get() == NULL) {
        ...
    }

    // 调用_JNI_ENV 结构体的函数成员变量方法
    if ((*env)->RegisterNatives(e, c.get(), gMethods, numMethods) < 0) {
        ...
    }

    return 0;
}
```

这里需要查看 JNIEnv 结构的定义：

libnativehelper/include/nativehelper/jni.h

```
struct _JNIEnv;
typedef _JNIEnv JNIEnv;
struct _JNIEnv {
    /* do not rename this; it does not seem to be entirely opaque */
```

```

const struct JNINativeInterface* functions;
...
jint RegisterNatives(jclass clazz, const JNINativeMethod* methods,
                     jint nMethods)
// 这里实际上是委托给 JNINativeInterface 进行处理
{ return functions->RegisterNatives(this, clazz, methods, nMethods); }
}

struct JNINativeInterface {
...
jint (*RegisterNatives)(JNIEnv*, jclass, const JNINativeMethod*, jint);
...
}

```

JNINativeInterface 的 RegisterNatives 函数指针实际上是指向虚拟机内部定义的 RegisterNatives 函数，这个指向过程和虚拟机的启动过程有关，这里先不分析，直接看该函数的实现：：

art/runtime/jni_internal.cc

```

static jint RegisterNatives(JNIEnv* env, jclass java_class, const JNINativeMethod*
methods, jint method_count) {
    return RegisterNativeMethods(env, java_class, methods, method_count, true);
}

static jint RegisterNativeMethods(JNIEnv* env, jclass java_class, const JNINativeMethod*
methods, jint method_count, bool return_errors) {
    ...
    for (jint i = 0; i < method_count; ++i) {
        const char* name = methods[i].name;
        const char* sig = methods[i].signature;
        const void* fnPtr = methods[i].fnPtr;
        ...
        // 获取到ArtMethod
        ArtMethod* m = nullptr;
        for (mirror::Class* current_class = c;
              current_class != nullptr;
              current_class = current_class->GetSuperClass()) {
            // Search first only comparing methods which are native.
    }
}

```

```
        m = FindMethod<true>(current_class, name, sig);
        if (m != nullptr) {
            break;
        }
        ...
    }
    ...
    // 委托给 ArtMethod 进行注册
    m->RegisterNative(fnPtr, is_fast);
}
return JNI_OK;
}
```

art/runtime/art_method.cc

```
void ArtMethod::RegisterNative(const void* native_method, bool is_fast) {
    ...
    SetEntryPointFromJni(native_method);
}

void SetEntryPointFromJni(const void* entrypoint) {
    ...
    SetEntryPointFromJniPtrSize(entrypoint, sizeof(void*));
}

ALWAYS_INLINE void SetEntryPointFromJniPtrSize(const void* entrypoint, size_t
pointer_size) {
    SetNativePointer(EntryPointFromJniOffset(pointer_size), entrypoint, pointer_
size);
}

// 获取 entry_point_from_jni_ 的 offset
static MemberOffset EntryPointFromJniOffset(size_t pointer_size) {
    return MemberOffset(PtrSizedFieldsOffset(pointer_size) + OFFSETOF_MEMBER(
        PtrSizedFields, entry_point_from_jni_) / sizeof(void*) * pointer_size);
}

template<typename T>
ALWAYS_INLINE void SetNativePointer(MemberOffset offset, T new_value, size_t p
ointer_size) {
    // 获取 jni 偏移地址
    const auto addr = reinterpret_cast<uintptr_t>(this) + offset.Uint32Value
();
    // 对 jni 地址进行更新
}
```

```
if (pointer_size == sizeof(uint32_t)) {
    uintptr_t ptr = reinterpret_cast<uintptr_t>(new_value);
    *reinterpret_cast<uint32_t*>(addr) = dchecked_integral_cast<uint32_t>
(ptr);
} else {
    *reinterpret_cast<uint64_t*>(addr) = reinterpret_cast<uintptr_t>(new_v
alue);
}
}
```

最终 register 方法会调用到虚拟机，并对对应 ArtMethod 的 entry_point_from_jni_ 进行更新

对开机 framework JNI 初始化流程做个总结：

1. 开机时系统启动 init 进程，然后启动 zygote 进程
2. zygote 进程是所有 APP 进程的父进程，在启动的过程中会去启动虚拟机并初始化 framework JNI 的注册
3. zygote 通过调用给定的 register_xxx 系列方法，对各个模块进行注册
4. 各个模块都会调用 RegisterMethodsOrDie 对 JNINativeMethod 数组进行注册
5. 调用最终会深入到虚拟机，通过 SetNativePointer 方法更新各个 ArtMethod 的 entry_point_from_jni_ 入口指针

自此 JNI 注册操作完成，在调用到 JAVA 中的 native 方法后，虚拟机自动调用到 JNI 实现，后续再会介绍 JAVA 层如何调用到 Native 层的

System.loadLibrary() 原理

除了 framework 开机对必备的 JNI 接口初始化以外，APP 自身也会有使用 JNI 接口的需求，这就需要使用到常用的 System.loadLibrary() 方法：

libcore/ojluni/src/main/java/java/lang/System.java

```
public static void loadLibrary(String libname) {
    Runtime.getRuntime().loadLibrary0(VMStack.getCallingClassLoader(), libname);
}
```

loadLibrary 的操作是委托 Runtime 进行完成的

libcore/ojluni/src/main/java/java/lang/Runtime.java

```
synchronized void loadLibrary0(ClassLoader loader, String libname) {
    ...
    String libraryName = libname;
    ...
    // 调用 System 的 native 方法进行 map
    String filename = System.mapLibraryName(libraryName);
    List<String> candidates = new ArrayList<String>();
    String lastError = null;
    // 获取 so 库的目录路径
    for (String directory : getLibPaths()) {
        String candidate = directory + filename;
        candidates.add(candidate);

        if (IoUtils.canOpenReadOnly(candidate)) {
            // 拼接路径完成后进行 Load
            String error = doLoad(candidate, loader);
            if (error == null) {
                return; // We successfully loaded the library. Job done.
            }
            lastError = error;
        }
    }
    ...

}

private String doLoad(String name, ClassLoader loader) {
    ...
    synchronized (this) {
        return nativeLoad(name, loader, librarySearchPath);
    }
}
```

```
}

// 调用到native 中进行Load
private static native String nativeLoad(String filename, ClassLoader loader, S
tring librarySearchPath);
```

调用 `System.loadLibrary()` 后，实际上在 JAVA 层只是做了些库路径的拼接操作，最终还是在 native 中实现的库加载

libcore/ojluni/src/main/native/Runtime.c

```
JNIEXPORT jstring JNICALL  
Runtime_nativeLoad(JNIEnv* env, jclass ignored, jstring javaFilename,  
                      jobject javaLoader, jstring javaLibrarySearchPath)  
{  
    return JVM_NativeLoad(env, javaFilename, javaLoader, javaLibrarySearchPat-  
h);  
}
```

随后调用到虚拟机进行加载

art/runtime/openjdkjvm/OpenjdkJvm.cc

```
}
```

art/runtime/java_vm_ext.cc

```
bool JavaVMExt::LoadNativeLibrary(JNIEnv* env,
                                    const std::string& path,
                                    jobject class_loader,
                                    jstring library_path,
                                    std::string* error_msg) {
    // 打开动态 so 库
    void* handle = android::OpenNativeLibrary(env, runtime_->GetTargetSdkVersion(),
                                                path_str, class_loader, library_path);
    ...
    // 获取动态库中 JNI_OnLoad 方法，并调用进行 JNI 初始化
    sym = library->FindSymbol("JNI_OnLoad", nullptr);
    ...
    typedef int (*JNI_OnLoadFn)(JavaVM*, void*);
    JNI_OnLoadFn jni_on_load = reinterpret_cast<JNI_OnLoadFn>(sym);
    int version = (*jni_on_load)(this, nullptr);
}
```

在 `JNI_OnLoad` 方法中，一般也是需要调用 `RegisterNativeMethods` 方法来实现 native 方法与 java 与 native 层的 JNI 映射

总结一下 `System.loadLibrary()` 的操作流程：

1. JAVA 层在调用 native 方法前首先调用 `System.loadLibrary()` 方法进行动态库初始化
2. 经过 native 层的中转，会调用到虚拟机中的实现，调用 `LoadNativeLibrary` 进行加载
3. 虚拟机加载动态库中的 `JNI_OnLoad` 进行初始化，之后的流程和上一节一致

可以看到，无论是 framework 初始化、还是 APP 主动加载动态库，要想使用 JNI 就必须调用 `RegisterNatives` 系列方法初始化 JNI 映射

18. 深入解析 Glide 源码

功能介绍

使用文章介绍以及和 Picasso 的对比分析请参考 [Introduction to Glide, Image Loader Library for Android, recommended by Google](#)

由于这篇文章使用 glide 的老版本，因此有些使用方法可能不太一致了。

本文基于 github 上 Glide 最新代码 4.0.0 版本做解析。

最基本的使用方式如下：

```
Glide.with(this)
    .asDrawable()
    .load("http://i6.topit.me/6/5d/45/1131907198420455d6o.jpg")
    .apply(fitCenterTransform(this))
    .apply(placeHolderOf(R.drawable.skyblue_logo_wechatfavorite_checked))
    .into(imageView);
```

Glide 使用了现在非常流行的流氏编码方式，方便了开发者的使用，简明、扼要。接下来主要对上面这一段流氏操作做拆分。

Glide 主入口

这个类有点像门脸模式的统一代理入口，不过实际作用在 4.0.0 中很多功能都被放到后面的其他类中，此类关注的点就很少了。虽然整个 libray 的所有需要的组建都在这个类中，但是实际也只是一个统一初始化的地方。

RequestManager (Glide.with(...))

这个类主要用于管理和启动 Glide 的所有请求，可以使用 activity, fragment 或者连接生命周期的事件去智能的停止，启动，和重启请求。也可以检索或者通过

实例化一个新的对象，或者使用静态的 Glide 去利用构建在 Activity 和 Fragment 生命周期处理中。它的方法跟你的 Fragment 和 Activity 的是同步的。

RequestBuilder

通用类，可以处理设置选项，并启动负载的通用资源类型。

在这个类中，主要是应用请求的很多选项(如下的选项从字面都能看出具体的用处，在 ImageView 控件中经常也能看到，另外之前版本可不是这么使用的)：

```
public final class RequestOptions extends BaseRequestOptions<RequestOptions> {  
  
    private static RequestOptions skipMemoryCacheTrueOptions;  
  
    private static RequestOptions skipMemoryCacheFalseOptions;  
  
    private static RequestOptions fitCenterOptions;  
  
    private static RequestOptions centerCropOptions;  
  
    private static RequestOptions circleCropOptions;  
  
    private static RequestOptions noTransformOptions;  
  
    private static RequestOptions noAnimationOptions;  
  
    // ...省略...}  

```

RequestBuilder<transcodetype> transition(TransitionOptions transitionOptions)} 这个方法主要是用于加载对象从占位符（placeholder）或者缩略图（thumbnail）到真正对象加载完成的转场动画。</transcodetype>

`RequestBuilder<transcodetype> load(...)` 多太方法中，这里可以加载很多类型的数据对象，可以是 `String`, `Uri`, `File`, `resourceId`, `byte[]` 这些。当然这些后面对应的编码方式也是不一样的。

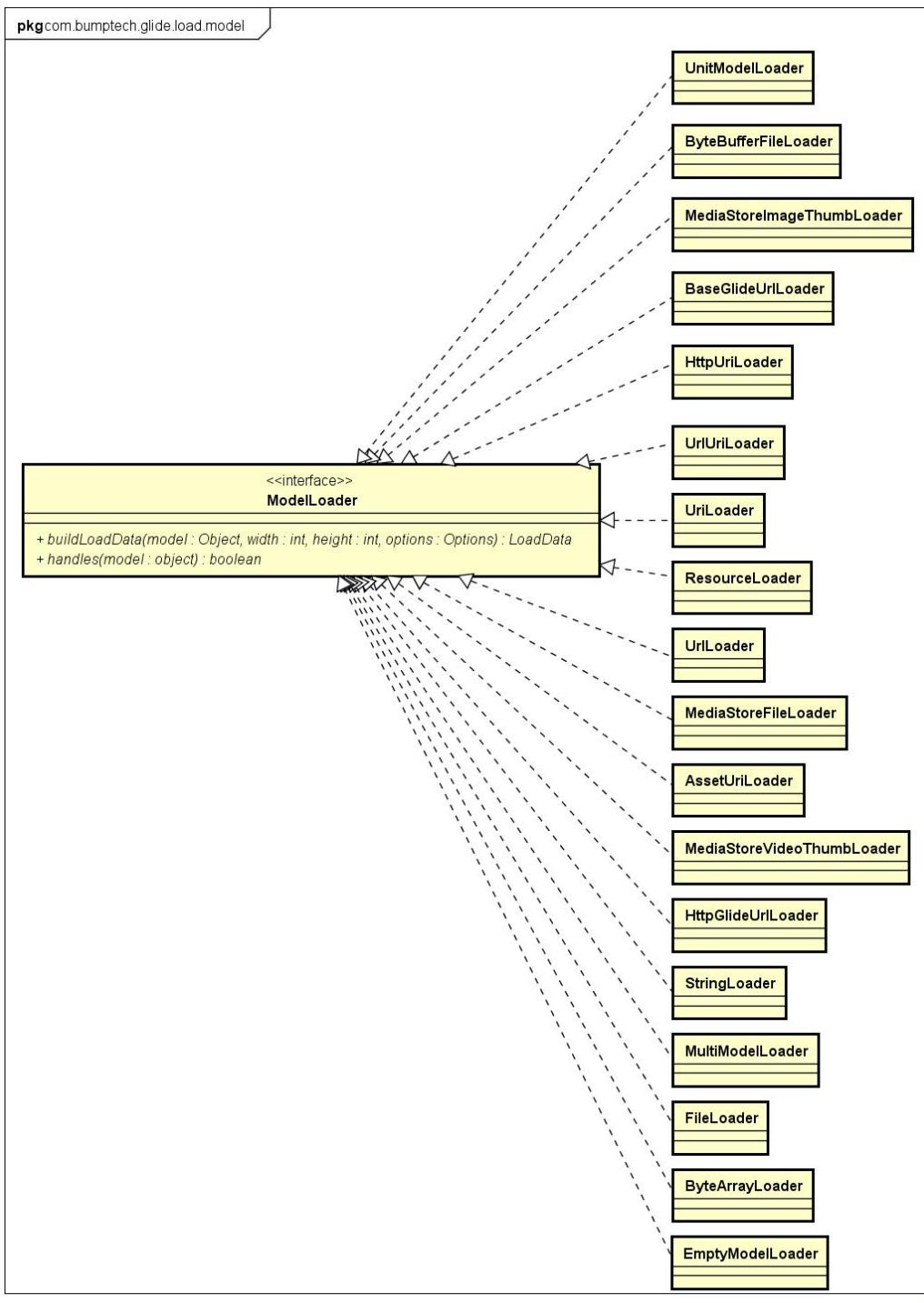
`Target<transcodetype> into(...)` 这个方法是触发 `Request` 真正启动的地方，在上边的示例中最后就是调用这个方法发起请求。

不得不说的是 `registry` 域，这个域挂载了很多元件，该注册器中囊括了模块加载器（`ModelLoader`）、编码器（`Encoder`）、资源解码器（`ResourceDecoder`）、资源编码器（`ResourceEncoder`）、数据回转器（`DataRewinder`）、转码器（`Transcoder`）。这些都是 `Glide` 在对资源编解码中既是基础又是核心功能。

代码结构

这里主要列举一下一些重要的组件以及他们的结构关系：

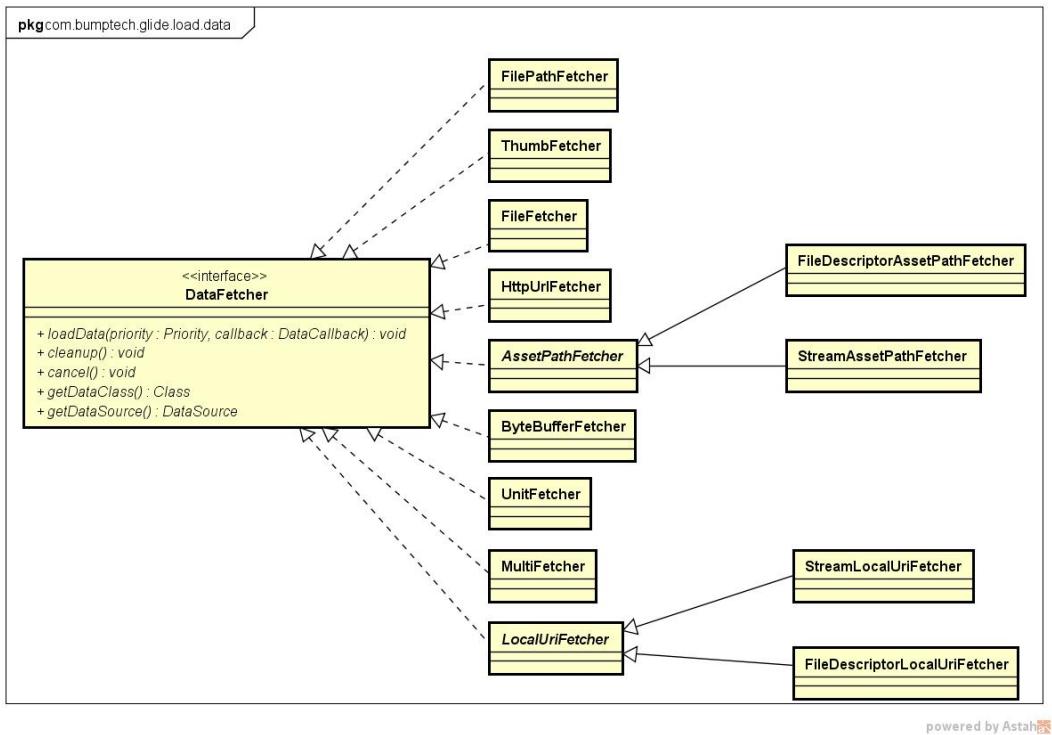
ModelLoader



powered by Astah

ModelLoader

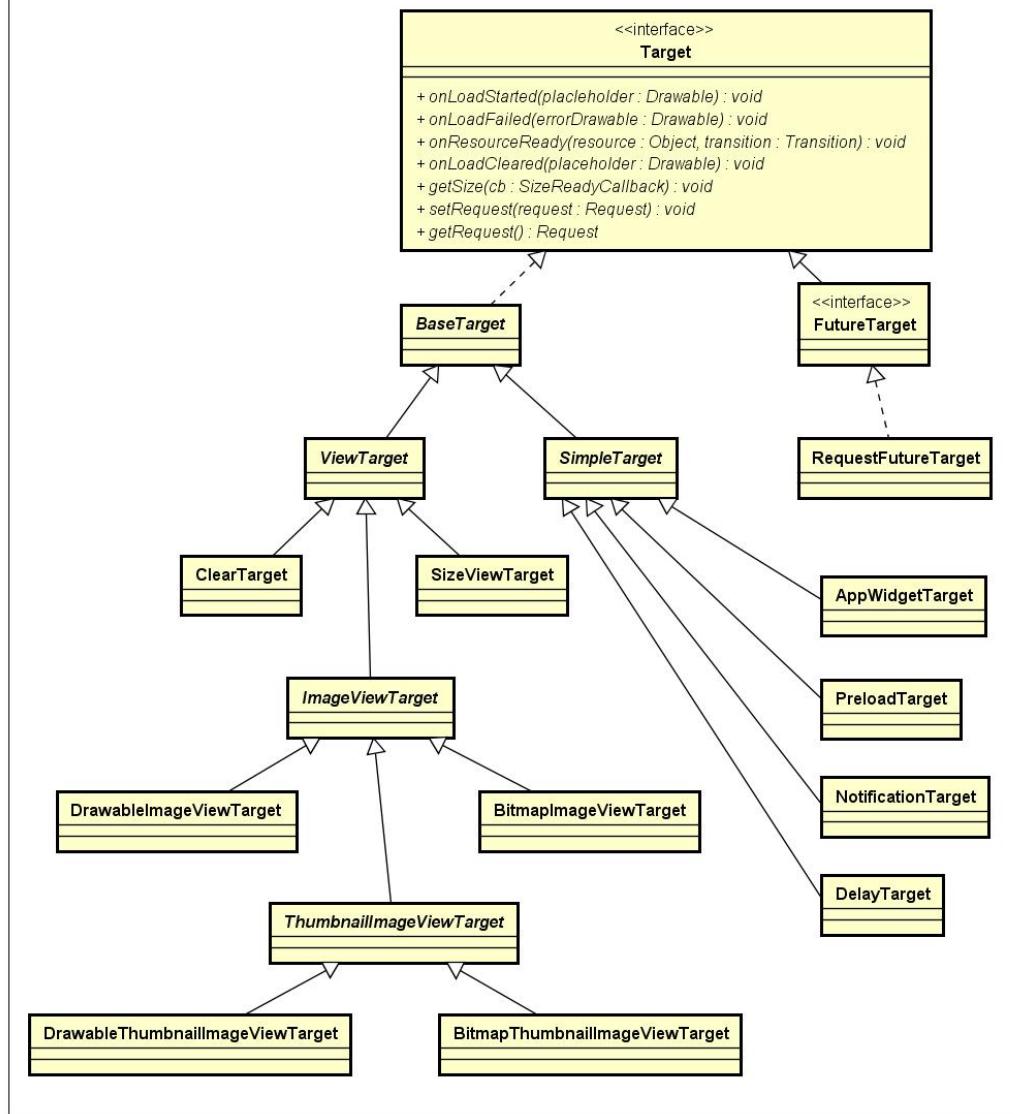
DataFetcher



DataFetcher

Target

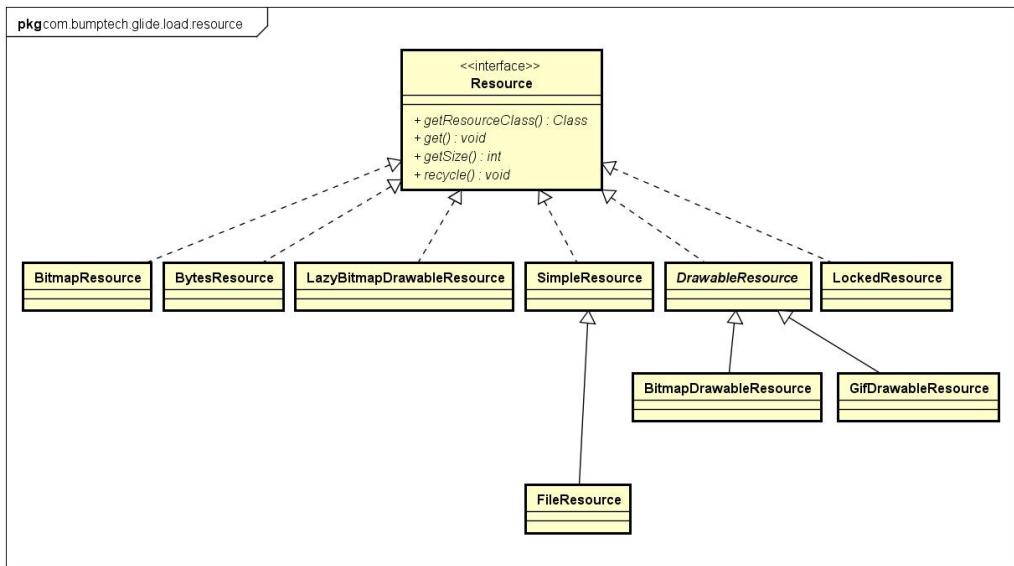
pkg com.bumptech.glide.request.target



powered by Astah

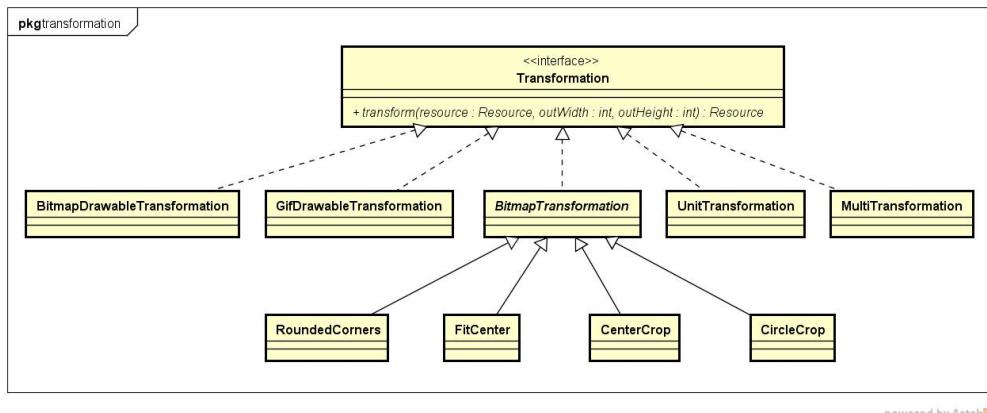
Target

Resource



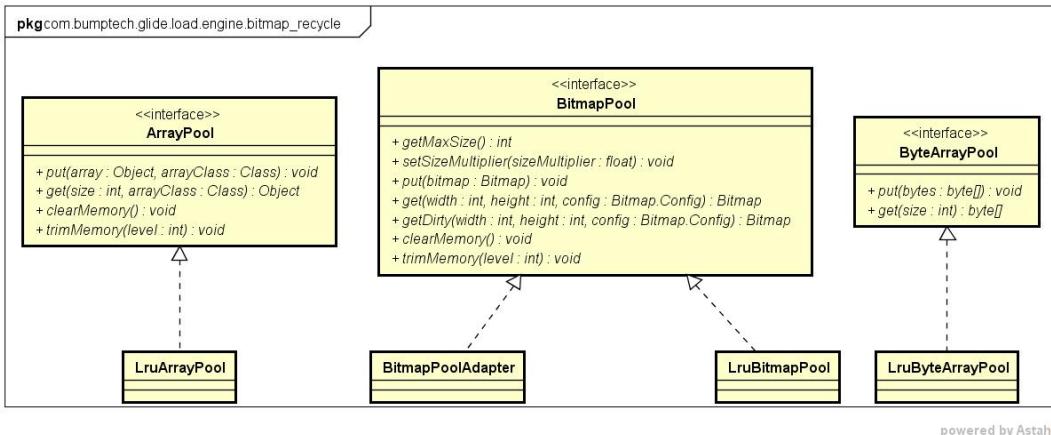
Resource

ResourceTransformation



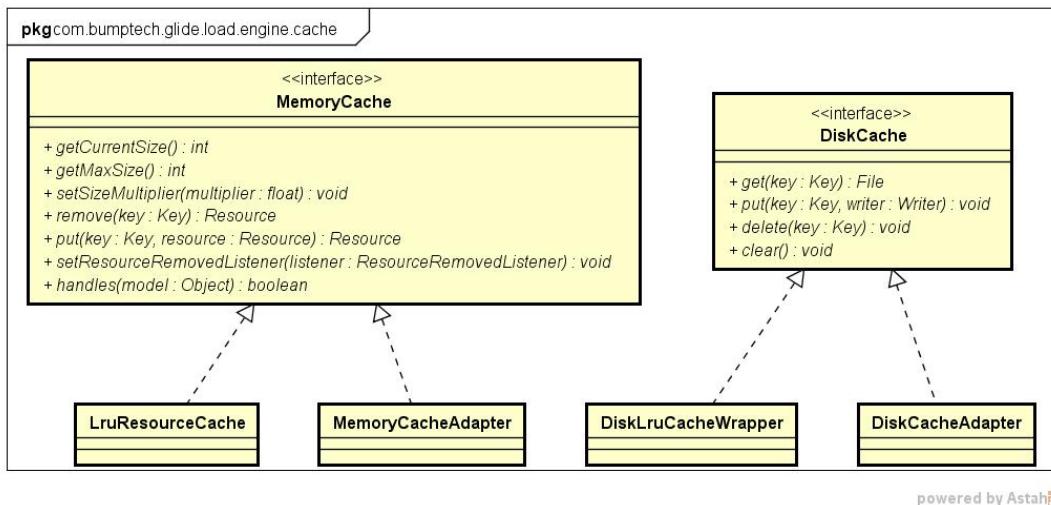
ResourceTransformation

Pool



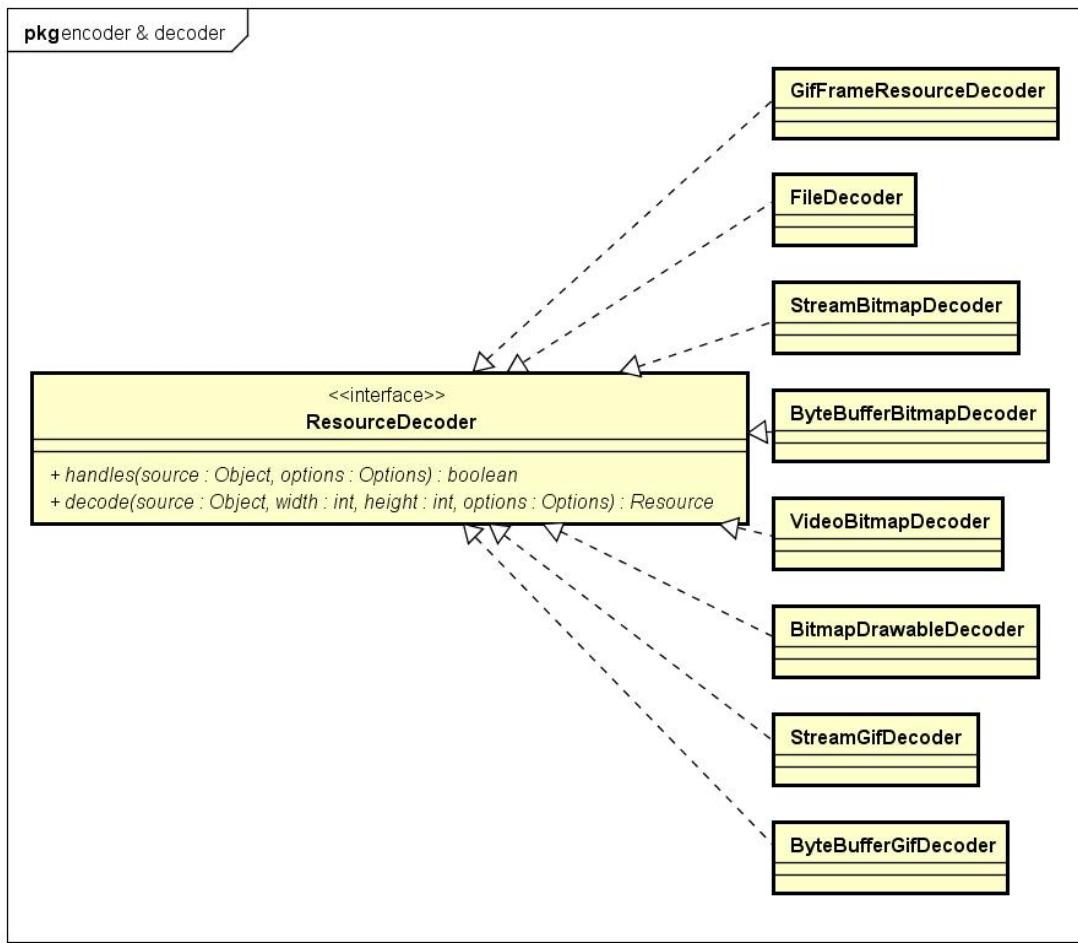
Pool

Cache



Cache

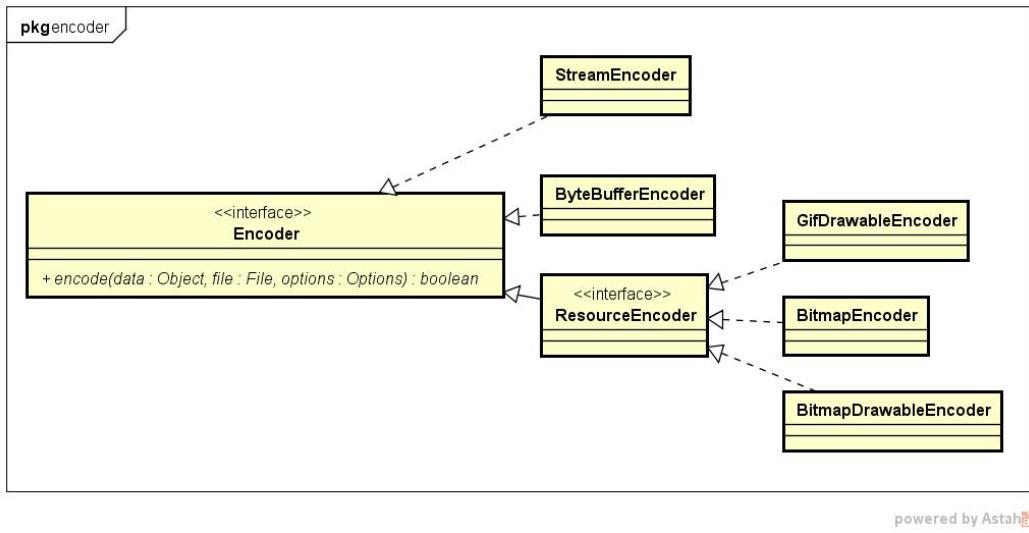
Decoder



powered by Astah

Decoder

Encoder



Encoder

把这些组件代码结构列举出来主要是为了让读者和使用者一目了然的看到自己需要的一些功能。

执行流程

1、根据不同版本的 Fragment 创建 RequestManagerFragment 或者 SupportRequestManagerFragment， 并加入到对应的 FragmentManager 中。这两种 Fragment 是不带有任何界面的，主要是用于同步生命周期。具体实现如下：

```

public static RequestManager with(Context context) {
    RequestManagerRetriever retriever = RequestManagerRetriever.get();
    return retriever.get(context);
}

// RequestManagerRetriever.get(...) @TargetApi(Build.VERSION_CODES.HONEYCOMB)

public RequestManager get(Activity activity) {
}

```

```
if (Util.isOnBackgroundThread() || Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {

    return get(activity.getApplicationContext());

} else {

    assertNotDestroyed(activity);

    android.app.FragmentManager fm = activity.getFragmentManager();

    return fragmentGet(activity, fm, null);

}

}

@TargetApi(Build.VERSION_CODES.HONEYCOMB)

RequestManager fragmentGet(Context context, android.app.FragmentManager fm,

    android.app.Fragment parentHint) {

    RequestManagerFragment current = getRequestManagerFragment(fm, parentHint);

    RequestManager requestManager = current.getRequestManager();

    if (requestManager == null) {

        requestManager =

            new RequestManager(context, current.getLifecycle(), current.getRequestManagerTreeNode());

        current.setRequestManager(requestManager);

    }

    return requestManager;

}
```

```
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR1)

RequestManagerFragment getRequestManagerFragment(
    final android.app.FragmentManager fm, android.app.Fragment parentHint)
{

    RequestManagerFragment current = (RequestManagerFragment) fm.findFragmentByTag(FRAGMENT_TAG);

    if (current == null) {

        current = pendingRequestManagerFragments.get(fm);

        if (current == null) {

            current = new RequestManagerFragment();

            current.setParentFragmentHint(parentHint);

            pendingRequestManagerFragments.put(fm, current);

            fm.beginTransaction().add(current, FRAGMENT_TAG).commitAllowingStateLoss();

            handler.obtainMessage(ID_REMOVE_FRAGMENT_MANAGER, fm).sendToTarget();

        }

    }

    return current;

}
```

2、创建一个 RequestBuilder，并添加一个 DrawableTransitionOptions 类型的转场动画

```
public RequestBuilder<Drawable> asDrawable() {

    return as(Drawable.class).transition(new DrawableTransitionOptions());
}
```

3、加载对象（model 域）

```
public RequestBuilder<TranscodeType> load(@Nullable Object model) {  
  
    return loadGeneric(model);  
  
}  
  
private RequestBuilder<TranscodeType> loadGeneric(@Nullable Object model)  
{  
  
    this.model = model;  
  
    isModelSet = true;  
  
    return this;  
  
}
```

4、装载对象（包含请求的发起点）。

```
public <Y extends Target<TranscodeType>> Y into(@NonNull Y target) {  
  
    Util.assertMainThread();  
  
    Preconditions.checkNotNull(target);  
  
    if (!isModelSet) {  
  
        throw new IllegalArgumentException("You must call #load() before calling #into()");  
  
    }  
  
  
  
    Request previous = target.getRequest();  
  
  
  
  
    if (previous != null) {  
  
        requestManager.clear(target);  
  
    }
```

```
}

requestOptions.lock();

Request request = buildRequest(target);

target.setRequest(request);

requestManager.track(target, request);

return target;

}
```

一般而言，大部分使用者都是用来装载图片的，因此都会调用如下这个方法：

```
public Target<TranscodeType> into(ImageView view) {

    Util.assertMainThread();

    Preconditions.checkNotNull(view);

    if (!requestOptions.isTransformationSet()

        && requestOptions.isTransformationAllowed()

        && view.getScaleType() != null) {

        if (requestOptions.isLocked()) {

            requestOptions = requestOptions.clone();

        }

        switch (view.getScaleType()) {

            case CENTER_CROP:

                requestOptions.optionalCenterCrop(context);


```

```
        break;

    case FIT_CENTER:
    case FIT_START:
    case FIT_END:
        requestOptions.optionalFitCenter(context);
        break;
    // $CASES-OMITTED$

    default:
        // Do nothing.

    }
}

return into(context.buildImageViewTarget(view, transcodeClass));
}
```

这里针对 ImageView 的填充方式做了筛选并对应设置到 requestOptions 上。最终的是通过 ImageView 和转码类型 (transcodeClass) 创建不通过的 Target (例如 Bitmap 对应的 BitmapImageViewTarget 和 Drawable 对应的 DrawableImageViewTarget)

4.1 Request 的创建 buildRequest(target)。

在 Request 的创建中会针对是否有缩略图来创建不同尺寸的请求，缩略图方法可以使用 RequestBuilder.thumbnail(...)方法来添加上。

Glide 中的 Request 都是使用了 SingleRequest 类，当然缩略图采用的是 ThumbnailRequestCoordinator 类：

```
private Request obtainRequest(Target<TranscodeType> target,  
    BaseRequestOptions<?> requestOptions, RequestCoordinator requestCoordinator,  
    TransitionOptions<?, ? super TranscodeType> transitionOptions, Priority priority,  
    int overrideWidth, int overrideHeight) {  
  
    requestOptions.lock();  
  
    //  
  
    return SingleRequest.obtain(  
        context,  
        model,  
        transcodeClass,  
        requestOptions,  
        overrideWidth,  
        overrideHeight,  
        priority,  
        target,  
        requestOptions.listener,  
        requestCoordinator,  
        context.getEngine(),  
        transitionOptions.getTransitionFactory());
```

```
}
```

比较值得推崇的是 SingleRequest.obtain 写法，个人认为比 new 关键字更简洁明了吧。

target.setRequest(request)也是一个比较值得注意的地方，如果 target 是 ViewTarget，那么 request 会被设置到 View 的 tag 上。这样其实是有好处，每一个 View 有一个自己的 Request，如果有重复请求，那么都会先去拿到上一个已经绑定的 Request，并且从 RequestManager 中清理回收掉。这应该是去重的功能。

4.2 requestManager.track(target, request)

这个方法非常的复杂，主要用于触发请求、编解码、装载和缓存这些功能。下面就一步一步来看吧：

4.2.1 缓存 target，并启动 Request

```
void track(Target<?> target, Request request) {  
  
    targetTracker.track(target);  
  
    requestTracker.runRequest(request);  
  
}  
  
/**  
 * Starts tracking the given request.  
 */  
  
public void runRequest(Request request) {  
  
    requests.add(request); //添加内存缓存  
  
    if (!isPaused) {  
  
        request.begin(); // 开始
```

```
    } else {

        pendingRequests.add(request); // 挂起请求

    }

}
```

继续看一下 SingleRequest 中的 begin 方法：

```
@Override

public void begin() {

    stateVerifier.throwIfRecycled();

    startTime = LogTime.getLogTime();

    // 如果 model 空的，那么是不能执行的。这里的 model 就是前面提到的 RequestBuilder 中的 model

    if (model == null) {

        if (Util.isValidDimensions	overrideWidth, overrideHeight)) {

            width = overrideWidth;

            height = overrideHeight;

        }

        // Only log at more verbose log levels if the user has set a fallback drawable, because

        // fallback Drawables indicate the user expects null models occasionally.

        int logLevel = getFallbackDrawable() == null ? Log.WARN : Log.DEBUG;

        onLoadFailed(new GlideException("Received null model"), logLevel);

        return;

    }

}
```

```
status = Status.WAITING_FOR_SIZE;

// 如果当前的 View 尺寸已经加载获取到了，那么就会进入真正的加载流程。

if (Util.isValidDimensions(overrideWidth, overrideHeight)) {

    onSizeReady(overrideWidth, overrideHeight);

} else {

    // 反之，当前 View 还没有画出来，那么是没有尺寸的。

    // 这里会调用到 ViewTreeObserver.addOnPreDrawListener。

    // 等待 View 的尺寸都 ok，才会继续

    target.getSize(this);

}

// 如果等待和正在执行状态，那么当前会加载占位符 Drawable

if ((status == Status.RUNNING || status == Status.WAITING_FOR_SIZE)

&& canNotifyStatusChanged()) {

    target.onLoadStarted(getPlaceholderDrawable());

}

if (Log.isLoggable(TAG, Log.VERBOSE)) {

    logV("finished run method in " + LogTime.getElapsedMillis(startTime));

}

}
```

接下来是 target.getSize(this)方法。这里主要说一下尺寸未加载出来的情况

(ViewTarget.java):

```
void getSize(SizeReadyCallback cb) {  
  
    int currentWidth = getViewWidthOrParam();  
  
    int currentHeight = getViewHeightOrParam();  
  
    if (isValidSize(currentWidth) && isValidSize(currentHeight)) {  
  
        cb.onSizeReady(currentWidth, currentHeight);  
  
    } else {  
  
        // We want to notify callbacks in the order they were added and we only  
        // expect one or two  
  
        // callbacks to  
  
        // be added a time, so a List is a reasonable choice.  
  
        if (!cbs.contains(cb)) {  
  
            cbs.add(cb);  
  
        }  
  
        if (layoutListener == null) {  
  
            final ViewTreeObserver observer = view.getViewTreeObserver();  
  
            layoutListener = new SizeDeterminerLayoutListener(this);  
  
            // 绘画之前加入尺寸的监听。这一点我想大部分 Android 开发同学应该都知道。  
  
            // 接下来在看看系统触发该 Listener 时 target 又干了些什么。  
  
            observer.addOnPreDrawListener(layoutListener);  
  
        }  
  
    }  
  
}  
  
private static class SizeDeterminerLayoutListener implements ViewTreeObserver
```

```
.OnPreDrawListener {

    // 注意这里是弱引用

    private final WeakReference<SizeDeterminer> sizeDeterminerRef;

    public SizeDeterminerLayoutListener(SizeDeterminer sizeDeterminer) {

        sizeDeterminerRef = new WeakReference<>(sizeDeterminer);
    }

    @Override

    public boolean onPreDraw() {

        if (Log.isLoggable(TAG, Log.VERBOSE)) {

            Log.v(TAG, "OnGlobalLayoutListener called listener=" + this);

        }

        SizeDeterminer sizeDeterminer = sizeDeterminerRef.get();

        if (sizeDeterminer != null) {

            // 通知 SizeDeterminer 去重新检查尺寸，并触发后续操作。

            // SizeDeterminer 有点像工具类，又作为尺寸回调的检测接口

            sizeDeterminer.checkCurrentDimens();

        }

        return true;
    }
}
```

ok, 继续回到 SingleRequest.onSizeReady 方法, 主要就是 Engine 发起 load 操作

```
public void onSizeReady(int width, int height) {

    stateVerifier.throwIfRecycled();

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("Got onSizeReady in " + LogTime.getElapsedMillis(startTime));

    }

    if (status != Status.WAITING_FOR_SIZE) {

        return;

    }

    status = Status.RUNNING;

    float sizeMultiplier = requestOptions.getSizeMultiplier();

    this.width = Math.round(sizeMultiplier * width);

    this.height = Math.round(sizeMultiplier * height);

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logV("finished setup for calling load in " + LogTime.getElapsedMillis(startTime));

    }

    loadStatus = engine.load(

        glideContext,

        model,

        requestOptions.getSignature(),

        this.width,
```

```
this.height,  
requestOptions.getResourceClass(),  
transcodeClass,  
priority,  
requestOptions.getDiskCacheStrategy(),  
requestOptions.getTransformations(),  
requestOptions.isTransformationRequired(),  
requestOptions.getOptions(),  
requestOptions.isMemoryCacheable(),  
this);  
  
if (Log.isLoggable(TAG, Log.VERBOSE)) {  
    logV("finished onSizeReady in " + LogTime.getElapsedMillis(startTime));  
}  
}
```

特别的，所有的操作都是来之唯一一个 Engine，它的创建是来自于 Glide 的初始化。如果有需要修改缓存配置的同学可以继续看一下 diskCacheFactory 的创建：

```
if (engine == null) {  
  
    engine = new Engine(memoryCache, diskCacheFactory, diskCacheExecutor, s  
ourceExecutor);  
  
}
```

继续看一下 Engine.load 的详细过程：

```
public <R> LoadStatus load(
```

```
GlideContext glideContext,  
  
Object model,  
  
Key signature,  
  
int width,  
  
int height,  
  
Class<?> resourceClass,  
  
Class<R> transcodeClass,  
  
Priority priority,  
  
DiskCacheStrategy diskCacheStrategy,  
  
Map<Class<?>, Transformation<?>> transformations,  
  
boolean isTransformationRequired,  
  
Options options,  
  
boolean isMemoryCacheable,  
  
ResourceCallback cb) {  
  
    Util.assertMainThread();  
  
    long startTime = LogTime.getLogTime();  
  
    // 创建 key，这是给每次加载资源的唯一标示。  
  
    EngineKey key = keyFactory.buildKey(model, signature, width, height, tr  
ansformations,  
  
        resourceClass, transcodeClass, options);  
  
    // 通过 key 查找缓存资源（PS 这里的缓存主要是内存中的缓存，切记，可以查看 Memo  
ryCache）
```

```
EngineResource<?> cached = loadFromCache(key, isMemoryCacheable);

if (cached != null) {

    // 如果有，那么直接利用当前缓存的资源。

    cb.onResourceReady(cached, DataSource.MEMORY_CACHE);

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logWithTimeAndKey("Loaded resource from cache", startTime, key);

    }

    return null;

}

// 这是一个二级内存的缓存引用，很简单用了一个 Map<Key, WeakReference<EngineResource<?>>>装载起来的。

// 这个缓存主要是谁来放进去呢？ 可以参考上面一级内存缓存 loadFromCache 方法。

EngineResource<?> active = loadFromActiveResources(key, isMemoryCacheable);

if (active != null) {

    cb.onResourceReady(active, DataSource.MEMORY_CACHE);

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logWithTimeAndKey("Loaded resource from active resources", startTime, key);

    }

    return null;

}
```

```
// 根据 key 获取缓存的 job。

EngineJob current = jobs.get(key);

if (current != null) {

    current.addCallback(cb); // 给当前 job 添加上回调 Callback

    if (Log.isLoggable(TAG, Log.VERBOSE)) {

        logWithTimeAndKey("Added to existing load", startTime, key);
    }

    return new LoadStatus(cb, current);
}

// 创建 job

EngineJob<R> engineJob = engineJobFactory.build(key, isMemoryCacheable);

DecodeJob<R> decodeJob = decodeJobFactory.build(
    glideContext,
    model,
    key,
    signature,
    width,
    height,
    resourceClass,
    transcodeClass,
    priority,
```

上面有一些值得注意的地方：

1. 内存缓存：在 Glide 中默认是 LruResourceCache。当然你也可以自定义；
 2. 为何要两级内存缓存（loadFromActiveResources）。个人理解是一级缓存采用 LRU 算法进行缓存，并不能保证全部能命中，添加二级缓存提高命中率之用；

3. EngineJob 和 DecodeJob 各自职责： EngineJob 充当了管理和调度者，主要负责加载和各类回调通知；DecodeJob 是真正干活的劳动者，这个类实现了 Runnable 接口。

下面来看看 DecodeJob 是如何执行的：

```
private void runWrapped() {  
  
    switch (runReason) {  
  
        case INITIALIZE:  
  
            // 初始化 获取下一个阶段状态  
  
            stage = getNextStage(Stage.INITIALIZE);  
  
            currentGenerator = getNextGenerator();  
  
            // 运行  
  
            runGenerators();  
  
            break;  
  
        case SWITCH_TO_SOURCE_SERVICE:  
  
            runGenerators();  
  
            break;  
  
        case DECODE_DATA:  
  
            decodeFromRetrievedData();  
  
            break;  
  
        default:  
  
            throw new IllegalStateException("Unrecognized run reason: " + runReason);  
    }  
}
```

```
}

// 这里的阶段策略首先是从 resource 中寻找, 然后再是 data, , 再是 sourceprivate
Stage getNextStage(Stage current) {

    switch (current) {

        case INITIALIZE:

            // 根据定义的缓存策略来回去下一个状态

            // 缓存策略来之于 RequestBuilder 的 requestOptions 域

            // 如果你有自定义的策略, 可以调用 RequestBuilder.apply 方法即可

            // 详细的可用缓存策略请参看 DiskCacheStrategy.java

            return diskCacheStrategy.decodeCachedResource()

        ? Stage.RESOURCE_CACHE : getNextStage(Stage.RESOURCE_CACHE);

        case RESOURCE_CACHE:

            return diskCacheStrategy.decodeCachedData()

        ? Stage.DATA_CACHE : getNextStage(Stage.DATA_CACHE);

        case DATA_CACHE:

            return Stage.SOURCE;

        case SOURCE:

            case FINISHED:

                return Stage.FINISHED;

            default:

                throw new IllegalArgumentException("Unrecognized stage: " + current);

    }

    // 根据 Stage 找到数据抓取生成器。private DataFetcherGenerator getNextGenerator() {
}
```

```
switch (stage) {  
  
    case RESOURCE_CACHE:  
  
        // 产生含有降低采样/转换资源数据缓存文件的 DataFetcher。  
  
        return new ResourceCacheGenerator(decodeHelper, this);  
  
    case DATA_CACHE:  
  
        // 产生包含原始未修改的源数据缓存文件的 DataFetcher。  
  
        return new DataCacheGenerator(decodeHelper, this);  
  
    case SOURCE:  
  
        // 生成使用注册的 ModelLoader 和加载时提供的 Model 获取源数据规定的 DataFetcher。  
  
        // 根据不同的磁盘缓存策略，源数据可首先被写入到磁盘，然后从缓存文件中加载，而不是直接返回。  
  
        return new SourceGenerator(decodeHelper, this);  
  
    case FINISHED:  
  
        return null;  
  
    default:  
  
        throw new IllegalStateException("Unrecognized stage: " + stage);  
    }  
}
```

经过很多流程，最后来到了发起实际请求的地方 SourceGenerator.startNext()方法：

```
public boolean startNext() {  
  
    if (dataToCache != null) {  
  
        Object data = dataToCache;
```

```
        dataToCache = null;

        cacheData(data);

    }

    if (sourceCacheGenerator != null && sourceCacheGenerator.startNext()) {

        return true;

    }

    sourceCacheGenerator = null;

    loadData = null;

    boolean started = false;// 查找 ModelLoader

    while (!started && hasNextModelLoader()) {

        loadData = helper.getLoadData().get(loadDataListIndex++);

        if (loadData != null

            && (helper.getDiskCacheStrategy().isDataCacheable(loadData.fetcher.getDataSource())

            || helper.hasLoadPath(loadData.fetcher.getDataClass())))) {

            started = true;

            根据 model 的 fetcher 加载数据

            loadData.fetcher.loadData(helper.getPriority(), this);

        }

    }

    return started;
```

```
}
```

这里的 Model 必须是实现了 GlideModule 接口的, fetcher 是实现了 DataFetcher 接口。有兴趣同学可以继续看一下 integration 中的 okhttp 和 volley 工程。Glide 主要采用了这两种网络 libray 来下载图片。

4.2.2 数据下载完成后的缓存处理 SourceGenerator.onDataReady

```
public void onDataReady(Object data) {

    DiskCacheStrategy diskCacheStrategy = helper.getDiskCacheStrategy();

    if (data != null && diskCacheStrategy.isDataCacheable(loadData.fetcher.
getDataSource())) {

        dataToCache = data;

        // We might be being called back on someone else's thread. Before doing
        anything, we should

        // reschedule to get back onto Glide's thread.

        cb.reschedule();

    } else {

        cb.onDataFetcherReady(loadData.sourceKey, data, loadData.fetcher,
loadData.fetcher.getDataSource(), originalKey);

    }
}
```

有些小伙伴可能看不太明白为什么就一个 dataToCache = data 就完了...其实 cb.reschedule()很重要，这里的 cb 就是 DecodeJob.reschedule():

```
public void reschedule() {

    runReason = RunReason.SWITCH_TO_SOURCE_SERVICE;
```

```
    callback.reschedule(this);

}
```

这里又有一个 Callback，继续追踪，这里的 Callback 接口是定义在 DecodeJob 内的，而实现是在外部的 Engine 中（这里会用线程池重新启动当前 job，那为什么要这样做呢？源码中的解释是为了不同线程的切换，因为下载都是借用第三方网络库，而实际的编解码是在 Glide 自定义的线程池中进行的）：

```
public void reschedule(DecodeJob<?> job) {

    if (isCancelled) {

        MAIN_THREAD_HANDLER.obtainMessage(MSG_CANCELLED, this).sendToTarget();

    } else {

        sourceExecutor.execute(job);

    }
}
```

接下来继续 DecodeJob.runWrapped() 方法。这个时候的 runReason 是 SWITCH_TO_SOURCE_SERVICE，因此直接执行 runGenerators()，这里继续执行 SourceGenerator.startNext() 方法，值得注意的 dataToCache 域，因为上一次执行的时候是下载，因此再次执行的时候内存缓存已经存在，因此直接缓存数据 cacheData(data)：

```
private void cacheData(Object dataToCache) {

    long startTime = LogTime.getLogTime();

    try {// 根据不同的数据获取注册的不同 Encoder

        Encoder<Object> encoder = helper.getSourceEncoder(dataToCache);

        DataCacheWriter<Object> writer =

            new DataCacheWriter<>(encoder, dataToCache, helper.getOptions());
    }
}
```

```
    originalKey = new DataCacheKey(loadData.sourceKey, helper.getSignature());
}

// 这里的 DiskCache 实现是 Engine 中 LazyDiskCacheProvider 提供的 DiskCacheAdapter.

helper.getDiskCache().put(originalKey, writer);

if (Log.isLoggable(TAG, Log.VERBOSE)) {
    Log.v(TAG, "Finished encoding source to cache"
        + ", key: " + originalKey
        + ", data: " + dataToCache
        + ", encoder: " + encoder
        + ", duration: " + LogTime.elapsedMillis(startTime));
}

} finally {
    loadData.fetcher.cleanup();
}

}

// 创建针对缓存的 Generator

sourceCacheGenerator =
    new DataCacheGenerator(Collections.singletonList(loadData.sourceKey), helper, this);
}
```

继续回到 SourceGenerator.startNext()方法，这个时候已经有了

sourceCacheGenerator，那么直接执行 DataCacheGenerator.startNext()方法：

```
public boolean startNext() {
```

```
while (modelLoaders == null || !hasNextModelLoader()) {  
  
    sourceIdIndex++;  
  
    if (sourceIdIndex >= cacheKeys.size()) {  
  
        return false;  
  
    }  
  
  
  
    Key sourceId = cacheKeys.get(sourceIdIndex);  
  
    Key originalKey = new DataCacheKey(sourceId, helper.getSignature());  
  
    cacheFile = helper.getDiskCache().get(originalKey);  
  
    if (cacheFile != null) {  
  
        this.sourceKey = sourceId;  
  
        modelLoaders = helper.getModelLoaders(cacheFile);  
  
        modelLoaderIndex = 0;  
  
    }  
  
}  
  
  
  
loadData = null;  
  
boolean started = false;  
  
// 这里会通过 model 寻找注册过的 ModelLoader  
  
while (!started && hasNextModelLoader()) {  
  
    ModelLoader<File, ?> modelLoader = modelLoaders.get(modelLoaderIndex+  
+);  
  
    loadData =
```

```
modelLoader.buildLoadData(cacheFile, helper.getWidth(), helper.getHeight(),
    helper.getOptions());

// 通过 FileLoader 继续加载数据

if (loadData != null && helper.hasLoadPath(loadData.fetcher.getDataClasses())) {

    started = true;

    loadData.fetcher.loadData(helper.getPriority(), this);

}

}

return started;

}
```

这里的 ModelLoader 跟之前提到过的 Register 的模块加载器（ModelLoader）

对应是 modelLoaderRegistry 域，具体执行的操作是

Registry.getModelLoaders(...)方法如下：

```
public <Model> List<ModelLoader<Model, ?>> getModelLoaders(Model model)
{
    List<ModelLoader<Model, ?>> result = modelLoaderRegistry.getModelLoaders(model);

    if (result.isEmpty()) {

        throw new NoModelLoaderAvailableException(model);

    }

    return result;
}
```

继续回到 DataCacheGenerator.startNext()方法，找到了 ModelLoader，这里笔者跟踪到的是 FileLoader 类(FileFetcher.loadData(...))方法)：

```
public void loadData(Priority priority, DataCallback<? super Data> callback) {  
  
    // 读取文件数据  
  
    try {  
  
        data = opener.open(file);  
  
    } catch (FileNotFoundException e) {  
  
        if (Log.isLoggable(TAG, Log.DEBUG)) {  
  
            Log.d(TAG, "Failed to open file", e);  
  
        }  
  
        //失败  
  
        callback.onLoadFailed(e);  
  
        return;  
  
    }  
  
    // 成功  
  
    callback.onDataReady(data);  
  
}
```

4.2.3 装载流程

回调通知这里就不打算多讲了，主要线路如下：

```
-->DataCacheGenerator.onDataReady  
  
-->SourceGenerator.onDataFetcherReady  
  
-->DecodeJob.onDataFetcherReady
```

```
-->DecodeJob.decodeFromRetrievedData  
-->DecodeJob.notifyEncodeAndRelease  
-->DecodeJob.notifyComplete  
-->EngineJob.onResourceReady
```

Debug 流程图:

装载流程 Debug 流程图

需要说明的就是在 EngineJob 中有一个 Handler 叫 MAIN_THREAD_HANDLER。

为了实现在主 UI 中装载资源的作用，ok 继续上边的流程：

```
-->EngineJob.handleResultOnMainThread  
-->SingleRequest.onResourceReady  
-->ImageViewTarget.onResourceReady  
-->ImageViewTarget.setImageResource  
-->ImageView.setImageDrawable/ImageView.setImageBitmap
```

Debug 流程图 2:

装载流程 Debug 流程图 2

数据的装载过程中有一个很重要的步骤就是 decode，这个操作发生在 DecodeJob.decodeFromRetrievedData 的时候，继续看代码：

```
>private void decodeFromRetrievedData() {
```

```
if (Log.isLoggable(TAG, Log.VERBOSE)) {  
  
    logWithTimeAndKey("Retrieved data", startFetchTime,  
  
    "data: " + currentData  
  
    + ", cache key: " + currentSourceKey  
  
    + ", fetcher: " + currentFetcher);  
  
}  
  
Resource<R> resource = null;  
  
try {  
  
    resource = decodeFromData(currentFetcher, currentData, currentDataSource);  
  
} catch (GlideException e) {  
  
    e.setLoggingDetails(currentAttemptingKey, currentDataSource);  
  
    exceptions.add(e);  
  
}  
  
if (resource != null) {  
  
    notifyEncodeAndRelease(resource, currentDataSource);  
  
} else {  
  
    runGenerators();  
  
}  
}
```

这中间发生了很多转换主要流程:

```
-->DecodeJob.decodeFromData  
  
-->DecodeJob.decodeFromFetcher
```

```
-->DecodeJob.runLoadPath  
  
-->LoadPath.load  
  
-->LoadPath.loadWithExceptionList  
  
-->LoadPath.decode  
  
-->LoadPath.decodeResource  
  
-->LoadPath.decodeResourceWithList  
  
-->ResourceDecoder.handles  
  
-->ResourceDecoder.decode
```

这里讲到了 decode，那么 encode 发生在什么时候呢？直接通过 Encoder 接口调用发现，在数据缓存的时候才会触发编码。具体调用在 DiskLruCacheWrapper 和 DataCacheWriter 中。一些值得参考的写法例如 BitmapEncoder 对 Bitmap 的压缩处理。

结束语

最近看开源库 Glide 关注度一直比较高，因此打算一探究竟。由于时间比较紧，因此一些应该有的时序图没有画，这里也只能简单用箭头代替。不过个人认为整体执行流程已经表达清楚了。

1. 总体来说代码写的挺漂亮的，单从使用者角度来说入手是比较容易的。
2. 源码使用了大量的工厂方法来创建对象，就像 String.valueOf(...) 方法一样，这也体现编码的优雅。
3. 不过想要对这个库进行改造，可能并非易事，笔者在跟踪代码的过程中发现很多地方有 Callback 这样的接口，来来回回查找几次很容易就晕头转向了。。。

4. 另外一个感觉难受的地方就是构造方法带入参数太多，就拿 SingleRequest 来说就是 12 个构造参数。
5. 单例的使用感觉还是有些模糊，就比如 GlideContext，有些时候通过 Glide.get(context).getGlideContext() 获取，而有些类中采用构造传入。个人觉得既然让 Glide 作为单例，那么还这样传入参数是不是有点多余？代码的编写都是可以折中考虑，不过如果整个项目拟定好了一个规则的话，我想最好还是遵循它。另外再吐槽一下单例，很多开发人员喜欢用单例，如果你是有代码洁癖的开发者，那么你肯定很讨厌这样，单例很容易造成代码的散落和结构不清晰。

思考

源码的解析只是把最重要的加载流程走了一遍，有一些比较细节的地方没有关注，如果你有需要，可以自己跟着这个主线 debug 一下就能查找到。

1. 为何要使用额外的无界面的 Fragment？
2. 如果开发者要使用这个 libray 作为图片加载库，而且项目本身对 App 的内存占用和 Size 都是有要求的话，那么 Register 是否有过重的嫌疑？

