**1、EventBus**

（1）通过注解+反射来进行方法的获取

**注解的使用**：@Retention(RetentionPolicy.*RUNTIME*)表示此注解在运行期可知，否则使用CLASS或者SOURCE在运行期间会被丢弃。

**通过反射来获取类和方法：**因为映射关系实际上是类映射到所有此类的对象的方法上的，所以应该通过反射来获取类以及被注解过的方法，并且将方法和对象保存为一个调用实体。

（2）使用ConcurrentHashMap来保存映射关系

**调用实体的构建**：调用实体中对于Object，也就是实际执行方法的对象不应该使用强引用而是应该**使用弱引用**，因为Map的static的，生命周期有可能长于被调用的对象，如果使用强引用就会出现内存泄漏的问题。

（3）方法的执行

使用Dispatcher进行方法的分派，异步则使用线程池来处理，同步就直接执行，而UI线程则使用MainLooper创建一个Handler，投递到主线程中去执行。

**2、Okhttp**

（1）**任务队列**

Okhttp使用了一个线程池来进行异步网络任务的真正执行，而对于任务的管理采用了任务队列的模型来对任务执行进行相应的管理，有点类似服务器的反向代理模型。Okhttp使用分发器Dispatcher来维护一个正在运行任务队列和一个等待队列。如果当前并发任务数量小于64，就放入执行队列中并且放入线程池中执行。而如果当前并发数量大于64就放入等待队列中，在每次有任务执行完成之后就在finally块中调用分发器的finish函数，在等待队列中查看是否有空余任务，如果有就进行入队执行。Okhttp就是使用任务队列的模型来进行任务的执行和调度的。

**（2）复用连接池**

Http使用的TCP连接有长连接和短连接之分，对于访问某个服务器的频繁通信，使用短连接势必会造成在建立连接上大量的时间消耗；而长连接的长时间无用保持又会造成资源你的浪费。Okhttp底层是采用Socket建立流连接，而连接如果不手动close掉，就会造成内存泄漏，那我们使用Okhttp时也没有做close操作，其实是Okhttp自己来进行连接池的维护的。在Okhttp中，它使用类似引用计数的方式来进行连接的管理，这里的计数对象是StreamAllocation，它被反复执行[aquire](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L296-L296)与[release](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L301-L301)操作，这两个函数其实是在改变Connection中的List<WeakReference<StreamAllocation>>大小。List中Allocation的数量也就是物理socket被引用的计数（Refference Count），如果计数为0的话，说明此连接没有被使用，是空闲的，需要通过淘汰算法实现回收。

在连接池内部维护了一个线程池，这个线程池运行的cleanupRunnable实际上是一个阻塞的runnable，内部有一个**无限循环**，在清理完成之后调用wait进行等待，等待的时间由cleanup的返回值决定，在等待时间到了之后再进行清理任务。

while (true) {

//执行清理并返回下场需要清理的时间

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) {

}

}

}

}

**Cleanup的过程如下所示：**

1. 遍历Deque中所有的RealConnection，标记泄漏的连接
2. 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**
3. 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
4. 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
5. 如果(没有任何连接)，就返回-1,跳出清理的死循环

再次注意：这里的“并发”==(“空闲”＋“活跃”)==5，而不是说并发连接就一定是活跃的连接

**如何标记空闲的连接呢？我们前面也说了，如果一个连接身上的引用为0，那么就说明它是空闲的，那么就要使用pruneAndGetAllocationCount来计算它身上的引用数，如同引用计数过程。**

过程其实很简单，就是遍历它的List<Reference<StreamAllocation>>，删除所有已经为null的弱引用，剩下的数量就是现在它的引用数量，如下段代码所示。

//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();

}

**3、Retrofit**

（1）Retrofit中的动态代理

**Java中的动态代理：**

首先动态代理是区别于静态代理的，代理模式中需要代理类和实际执行类同时实现一个相同的接口，并且在每个接口定义的方法前后都要加入相同的代码，这样有可能很多方法代理类都需要重复。而动态代理就是将这个步骤放入运行时的过程，一个代理类只需要实现InvocationHandler接口中的invoke方法，当需要动态代理时只需要根据接口和一个实现了InvocationHandler的代理对象A生成一个**最终的自动生成的代理对象A\***。这样最终的代理对象A\*无论调用什么方法，都会执行InvocationHandler的代理对象A的invoke函数，**你就可以在这个invoke函数中实现真正的代理逻辑。**

动态代理的实现机制实际上就是使用Proxy.newProxyInstance函数为动态代理对象A生成一个代理对象A\*的类的字节码从而生成具体A\*对象过程，这个A\*类具有几个特点，一是它需要实现传入的接口，第二就是所有接口的实现中都会调用A的invoke方法，并且传入相应的**调用实际方法（即接口中的方法）**。

**Retrofit中的动态代理**

Retrofit中使用了动态代理是不错，但是并不是为了真正的代理才使用的，它只是为了动态代理一个非常重要的功能，就是“拦截”功能。我们知道动态代理中自动生成的A\*对象的所有方法执行都会调用实际代理类A中的invoke方法，再由我们在invoke中实现真正代理的逻辑，实际上也就是A\*的所有方法都被A对象给拦截了。而Retrofit最重要的是什么？就是把一个网络执行变成像方法调用一样方便的过程：

public interface ZhuanLanApi {

@GET("/api/columns/{user} ")

Call<ZhuanLanAuthor> getAuthor(@Path("user") String user)

}

再用这个retrofit对象创建一个ZhuanLanApi对象：

ZhuanLanApi api = retrofit.create(ZhuanLanApi.class);

Call<ZhuanLanAuthor> call = api.getAuthor("qinchao");

也就是一个网络调用你只需要在你创建的接口里面通过注解进行设置，然后通过retrofit创建一个api然后调用，就可以自动完成一个Okhttp的Call的创建。通过create的源码的查看：

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

Utils.validateServiceInterface(service);

if (validateEagerly) {

eagerlyValidateMethods(service);

}

return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] { service },

new InvocationHandler() {

private final Platform platform = Platform.get();

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

throws Throwable {

// If the method is a method from Object then defer to normal invocation.

if (method.getDeclaringClass() == Object.class) {

return method.invoke(this, args);

}

if (platform.isDefaultMethod(method)) {

return platform.invokeDefaultMethod(method, service, proxy, args);

}

ServiceMethod serviceMethod = loadServiceMethod(method);

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

return serviceMethod.callAdapter.adapt(okHttpCall);

}

});

我们可以看出怎么从接口类创建成一个API对象？就是使用了动态代理中的拦截技术，通过创建一个符合此接口的动态代理对象A\*，那A呢？就是这其中创建的这个匿名类了，它在内部实现了invoke函数，这样A\*调用的就是A中的invoke函数，也就是被拦截了，实际运行invoke。而invoke就是根据调用的method的注解（前面红色标注的，会传入相应实际函数），从而生成一个符合条件的Okhttp的Call对象，供你使用（调用进行真正网络请求）。

（2）Retrofit实际作用

Retrofit实际上是为了更方便的使用Okhttp，因为Okhttp的使用就是构建一个Call，而构建Call的大部分过程都是相似的，而Retrofit正是利用了代理机制带我们动态的创建Call，而Call的创建信息就来自于你的注解。并且还可以根据配置Adapter等等对网络请求进行相应的处理和改变，这种插件式的解耦方式也提供了很大的扩展性。

**4、RxJava**

**1、观察者与被观察者通信**

（1）Observable的create函数

public final static <T> Observable<T> create(OnSubscribe<T> f) {

return new Observable<T>(hook.onCreate(f));

}

构造函数如下

protected Observable(OnSubscribe<T> f) {

this.onSubscribe = f;

}

创建了一个Observable我们记为**Observable1**，保存了传入的OnSubscribe对象为**onSubscribe**，这个很重要，后面会说到。

（2）onSubscribe方法

public final Subscription subscribe(Subscriber<? super T> subscriber) {

return Observable.subscribe(subscriber, this);

}

private static <T> Subscription subscribe(Subscriber<? super T> subscriber, Observable<T> observable) {

...

subscriber.onStart();

**onSubscribe.call(subscriber);**

return hook.onSubscribeReturn(subscriber);

}

重点在加粗部分，实际上调用的就是之前我们传入的**onSubscribe**的call方法，这样就实现了被观察者和观察者之间的通信逻辑，运行我们写好的call函数。

**2、变换过程（lift）**

（1）map函数

public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {

return lift(new OperatorMap<T, R>(func));

}

map函数直接调用了lift函数并且把我们的func传了进去，func就是我们所做的具体变换操作，我们看一下map平时使用的方式，加粗部分就是我们传进去的call函数的实现：

Observable.from(students)

.map(new Func1<Student, String>() {

**@Override**

**public String call(Student student) {**

**return student.getName();**

**}**

})

.subscribe(**subscriber**);

而这里的**Subscriber**，我们记为**Subscriber1。**

（2）lift函数

public <R> Observable<R> lift(Operator<? extends R, ? super T> operator) {

return **Observable.create**(new OnSubscribe<R>() {

@Override

**public void call(Subscriber subscriber)** {

**Subscriber newSubscriber = operator.call(subscriber);**

newSubscriber.onStart();

onSubscribe.call(newSubscriber);

}

});

}

我们可以看到这里我们又创建了一个新的Observable对象，我们记为**Observable2**，也就是说当我们执行map时，实际上返回了一个新的Observable对象，我们之后的subscribe函数实际上执行再我们新创建的**Observable2上**，这时他调用的就是我们新的call函数，也就是**Observable2**的call函数（加粗部分），我们来看一下这个operator的call的实现。这里call传入的就是我们的**Subscriber1对象，也就是调用最终的subscribe的处理对象。**

（3）Operator的call函数

public Subscriber<? super T> call(final Subscriber<? super R> o) {

return new Subscriber<T>(o) {

@Override

public void onNext(T t) {

o.onNext(transformer.call(t));

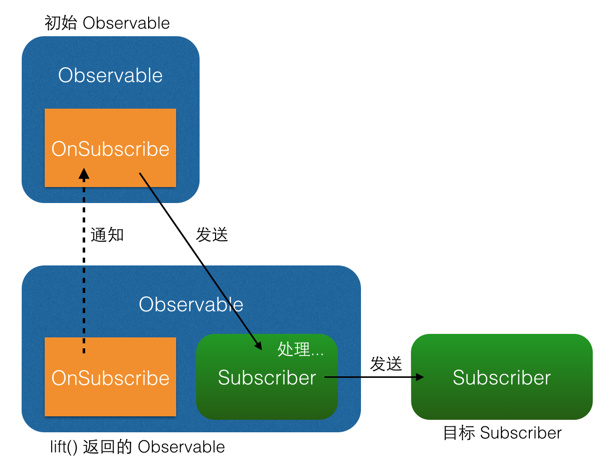
}

};

}

这里的transformer就是我们在map调用是传进去的func函数，也就是变换的具体过程。那看之后的onSubscribe.call（回到call中），这里的onSubscribe是谁呢？就是我们**Observable1保存的**onSubscribe对象，也就是我们前面说很重要的那个对象。而这个o（又回来了）就是我们的**Subscriber1，这里可以看出，在调用了转换函数之后我们还是调用了一开始的Subscriber1的onNext，最终事件经过转换传给了我们的结果。**

**（4）总结**

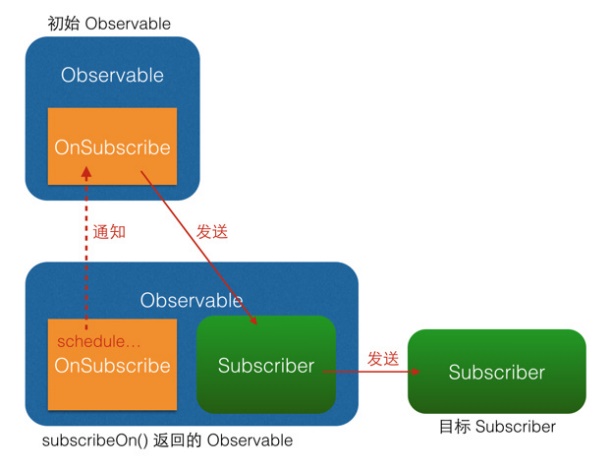
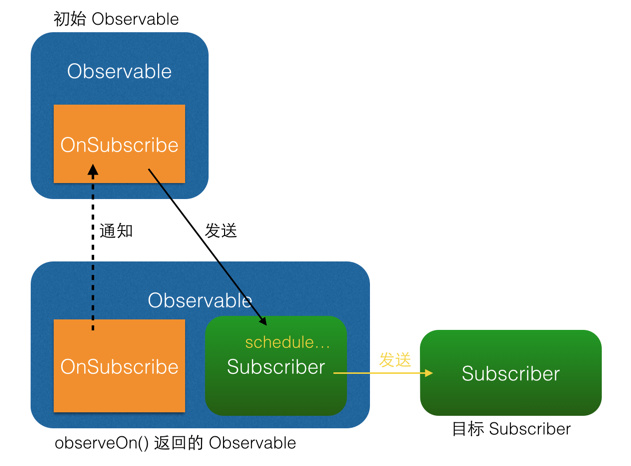


图上给出了直观的结果，实际上我们通过lift创建了一个新的Observable对象，记为Observable2，我们之后的subscribe实际上执行在了它身上。它执行了之前Observable1的call函数，并且创建了一个新的Subscriber对象，记为Subscriber2，它的作用就是接受原来Observable1的事件，然后经过转换，传递给最终的Subscriber1，执行它的onNext函数（这个逻辑在他的onNext中可以看出来）。我们这样就跑通了变换的整个逻辑了，我们也可以发现这个逻辑类似于拦截，通过拦截subscribe函数，再把原始Observable的subscribe拦截到新的Subscriber2对象中来执行，从而实现转换的逻辑。

**3、线程切换过程（Scheduler）**

RxJava最好用的特点就是提供了方便的线程切换，但它的原理归根结底还是lift，使用subscribeOn()的原理就是创建一个新的Observable，把它的call过程开始的执行投递到需要的线程中；而 observeOn() 则是把线程切换的逻辑放在自己创建的Subscriber中来执行。把对于最终的Subscriber1的执行过程投递到需要的线程中来进行。

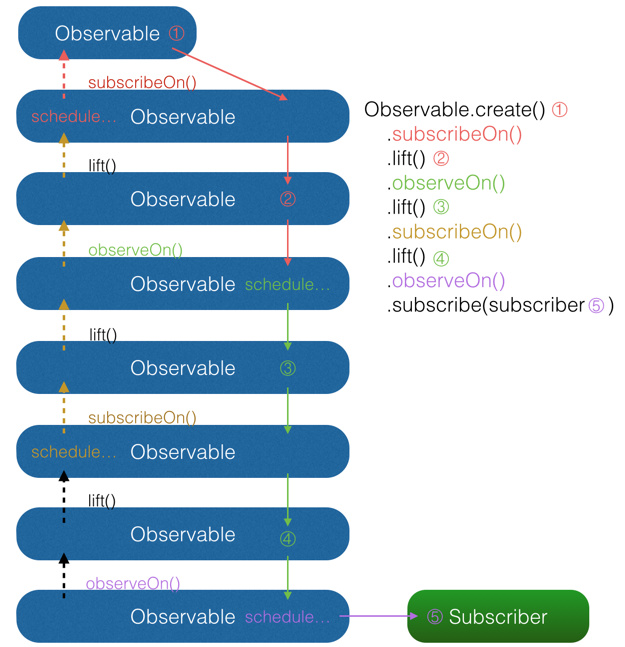
（1）区别

从图中可以看出，subscribeOn() 和 observeOn() 都做了线程切换的工作（图中的 "schedule..." 部位）。不同的是， subscribeOn()的线程切换发生在 OnSubscribe 中，即在它通知上一级 OnSubscribe 时，这时事件还没有开始发送，因此 subscribeOn() 的线程控制可以从事件发出的开端就造成影响；而 observeOn() 的线程切换则发生在它内建的 Subscriber 中，即发生在它即将给下一级 Subscriber 发送事件时，因此 observeOn() 控制的是它后面的线程。

（2）为什么subscribeOn()只有第一个有效？

因为它是从通知开始将后面的执行全部投递到需要的线程来执行，但是之后的投递会受到在它的上级的（但是执行在它之后）的影响，如果上面还有subscribeOn() ，又会投递到不同的线程中去，这样就不受到它的控制了。所以只有第一个有效果：



图中共有 5 处含有对事件的操作。由图中可以看出，①和②两处受第一个 subscribeOn() 影响，运行在红色线程；③和④处受第一个 observeOn() 的影响，运行在绿色线程；⑤处受第二个 onserveOn() 影响，运行在紫色线程；而第二个 subscribeOn() ，由于在通知过程中线程就被第一个 subscribeOn() 截断，因此对整个流程并没有任何影响。这里也就回答了前面的问题：当使用了多个 subscribeOn() 的时候，只有第一个 subscribeOn() 起作用。