# [谜之RxJava （二） —— Magic Lift](https://segmentfault.com/a/1190000004049841)

## 例子

Observable.create(new Observable.OnSubscribe<String>() {  
 @Override  
 public void call(Subscriber<? super String> subscriber) {  
 subscriber.onNext("hello");  
 }  
})  
.map(new Func1<String, String>() {  
 @Override  
 public String call(String s) {  
 return s + "word";  
 }  
})  
.subscribe(new Subscriber<String>() {  
 @Override  
 public void onCompleted() {  
  
 }  
  
 @Override  
 public void onError(Throwable e) {  
  
 }  
  
 @Override  
 public void onNext(String s) {  
 Log.*d*("rx", s);  
 }  
});

## lift

我们先看下进行链式调用map之后，发生了什么。

public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {  
 return lift(new OperatorMap<T, R>(func));  
}

对，就是调用了lift函数！，然后把我们的转换器（Transfomer）传入进去，看下它做了什么事。

public final <R> Observable<R> lift(final Operator<? extends R, ? super T> operator) {  
 return new Observable<R>(new OnSubscribe<R>() {  
 @Override  
 public void call(Subscriber<? super R> o) {  
 try {  
 Subscriber<? super T> st = hook.onLift(operator).call(o);  
 try {  
 // new Subscriber created and being subscribed with so 'onStart' it  
 st.onStart();  
 onSubscribe.call(st);  
 } catch (Throwable e) {  
 // localized capture of errors rather than it skipping all operators   
 // and ending up in the try/catch of the subscribe method which then  
 // prevents onErrorResumeNext and other similar approaches to error handling  
 if (e instanceof OnErrorNotImplementedException) {  
 throw (OnErrorNotImplementedException) e;  
 }  
 st.onError(e);  
 }  
 } catch (Throwable e) {  
 if (e instanceof OnErrorNotImplementedException) {  
 throw (OnErrorNotImplementedException) e;  
 }  
 // if the lift function failed all we can do is pass the error to the final Subscriber  
 // as we don't have the operator available to us  
 o.onError(e);  
 }  
 }  
 });  
}

来，我来简化一下

public final <R> Observable<R> lift(final Operator<? extends R, ? super T> operator) {  
 return new Observable<R>(...);  
}

返回了一个新的Observable对象，这才是重点！ 这种链式调用看起来特别熟悉？有没有像javascript中的Promise/A，在then中返回一个Promise对象进行链式调用？

OK，那么我们要看下它是如何工作的啦。

在map()调用之后，我们操作的就是新的Observable对象，我们可以把它取名为Observable$2，OK，我们这里调用subscribe，完整的就是Observable$2.subscribe，继续看到subscribe里，重要的几个调用：

hook.onSubscribeStart(observable, observable.onSubscribe).call(subscriber);  
return hook.onSubscribeReturn(subscriber);

注意注意 ！ 这里的observable是Observable$2！！也就是说，这里的onSubscribe是，lift中定义的！！

OK，我们追踪下去，回到lift的定义中。

return new Observable<R>(new OnSubscribe<R>() {  
 @Override  
 public void call(Subscriber<? super R> o) {  
 try {  
 Subscriber<? super T> st = hook.onLift(operator).call(o);  
 try {  
 // new Subscriber created and being subscribed with so 'onStart' it  
 st.onStart();  
 onSubscribe.call(st); //请注意我！！ 这个onSubscribe是原始的OnSubScribe对象！！  
 } catch (Throwable e) {  
 // localized capture of errors rather than it skipping all operators   
 // and ending up in the try/catch of the subscribe method which then  
 // prevents onErrorResumeNext and other similar approaches to error handling  
 if (e instanceof OnErrorNotImplementedException) {  
 throw (OnErrorNotImplementedException) e;  
 }  
 st.onError(e);  
 }  
 } catch (Throwable e) {  
 if (e instanceof OnErrorNotImplementedException) {  
 throw (OnErrorNotImplementedException) e;  
 }  
 // if the lift function failed all we can do is pass the error to the final Subscriber  
 // as we don't have the operator available to us  
 o.onError(e);  
 }  
 }  
});

一定一定要注意这段函数执行的上下文！，这段函数中的onSubscribe对象指向的是外部类，也就是第一个Observable的onSubScribe！而不是Observable$2中的onSubscribe，OK，谨记这一点之后，看看

Subscriber<? super T> st = hook.onLift(operator).call(o);

这行代码，就是定义operator，生成一个经过operator操作过的Subscriber，看下OperatorMap这个类中的call方法

@Override  
public Subscriber<? super T> call(final Subscriber<? super R> o) {  
 return new Subscriber<T>(o) {  
  
 @Override  
 public void onCompleted() {  
 o.onCompleted();  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 o.onError(e);  
 }  
  
 @Override  
 public void onNext(T t) {  
 try {  
 o.onNext(transformer.call(t));  
 } catch (Throwable e) {  
 Exceptions.*throwIfFatal*(e);  
 onError(OnErrorThrowable.*addValueAsLastCause*(e, t));  
 }  
 }  
  
 };  
}

没错，对传入的Subscriber做了一个代理，把转换后的值传入。  
这样就生成了一个代理的Subscriber，

最后我们最外层的OnSubscribe对象对我们代理的Subscriber进行了调用。。  
也就是

@Override  
public void call(Subscriber<? super String> subscriber) {  
 //此处的subscriber就是被map包裹(wrapper)后的对象。  
 subscriber.onNext("hello");  
}

然后这个subscriber传入到内部，链式的通知，最后通知到我们在subscribe函数中定义的对象。