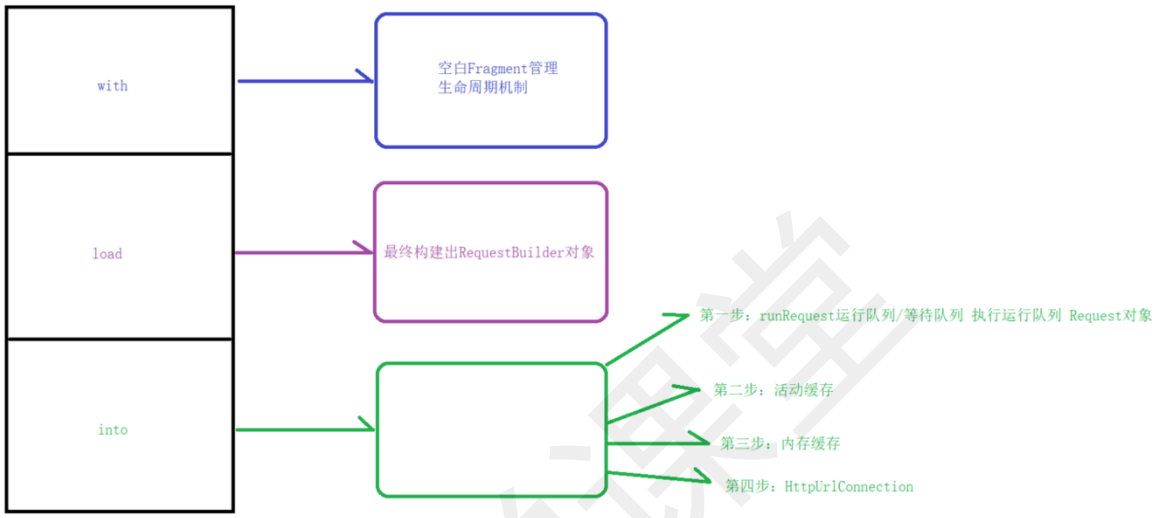
Glide第一节课(最新Glide4.11源码)

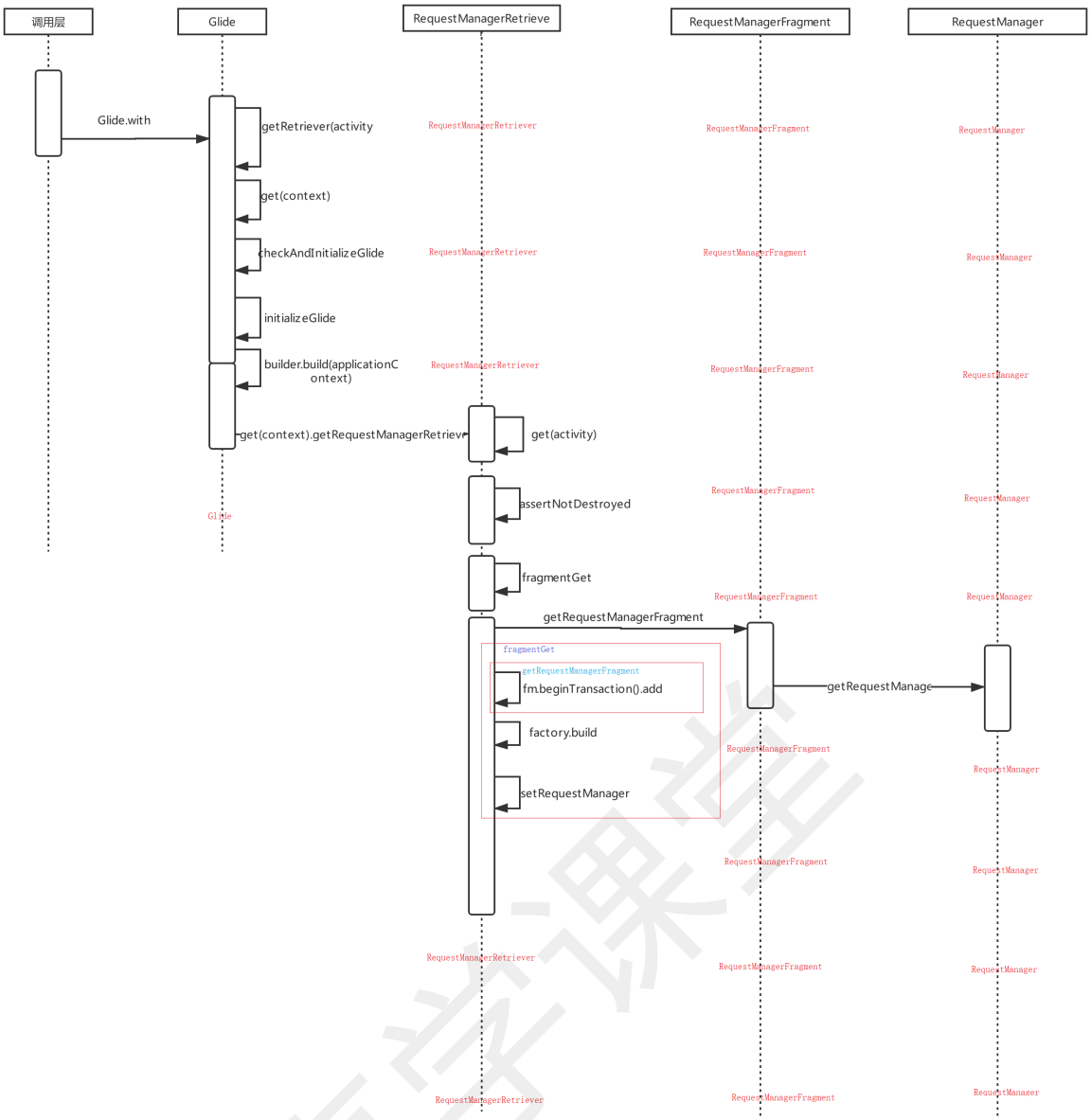
同学们，由于Glide太庞大了，所有把Glide拆分三份，本节课把第一份with搞定，我们会把 with 拿下

上课时必须给同学们画的图



with 函数 生命周期整体安排-必须要给同学们画

Glide with 时序图：



同学们说到前面的话：

同学们所认为的是不是这样的？

这样加载：

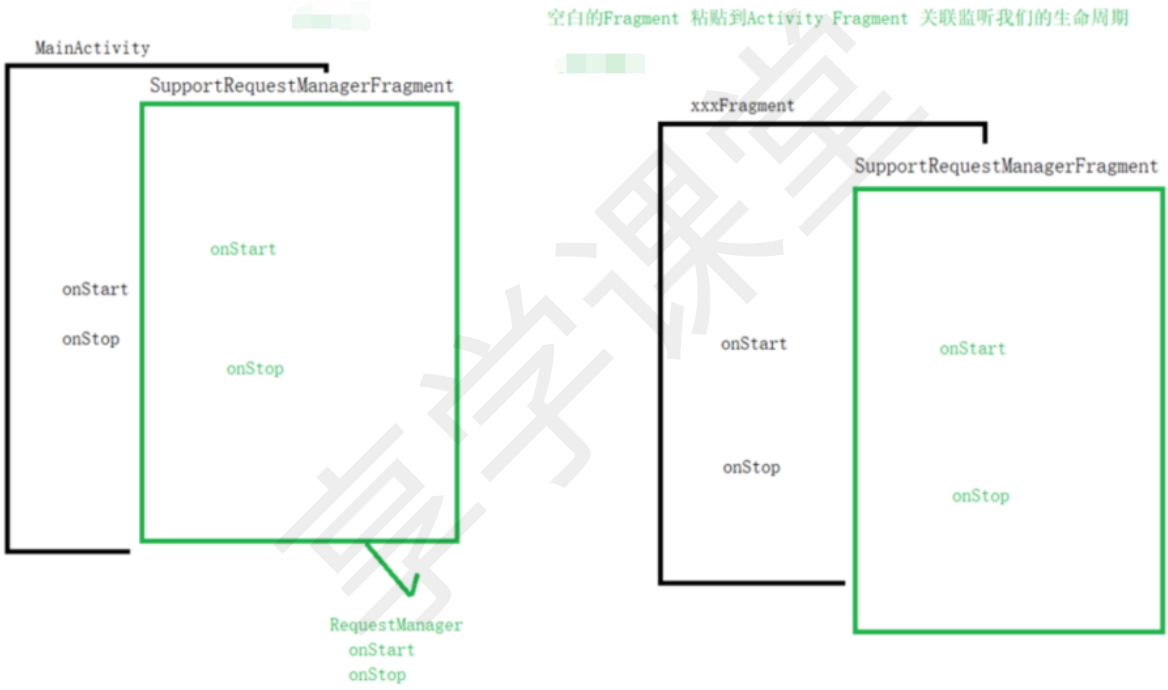
```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val imageView: ImageView = findViewById(R.id.image)  // 同学们：获取ImageView控件而已

        // TODO  常规方式
        Glide.with(this).load(URL).into(imageView)
}
```

对应着这样取消：

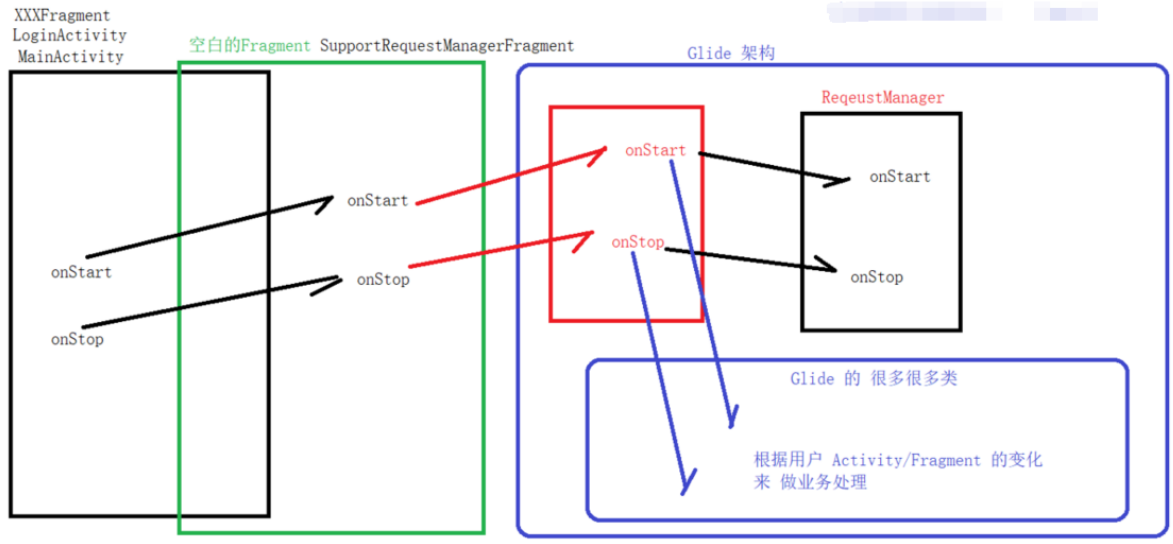
```
override fun onDestroy() {
        super.onDestroy()
        Glide.with(this).clear(imageView)
}
```

同学们应该都会认为，应该及时取消不必要的加载请求，但这并不是必须的操作，因为**【Glide内部会在 Activity/Fragment生命周期监听，网络变化监听，自动取消加载或者重新加载，等等】**

同学们关注绿色区域：with 会搞一个空白的Fragment 覆盖到 我们的xxxxMainActivity或xxxFragment上，就可以监听生命周期：



同学们，这个是 with 生命周期，整体的表现，看不懂没有关系，课程会讲：



同学们：来一个小总结

Fragment/Activity生命周期： 当我们的Fragment或Activity不可见的时候 暂停请求， 当我们的Fragment或Activity可见的时候 恢复请求；

【目录一：1.为何去监听Fragment/Activity生命周期】

第一大点：为何去监听Activity/Fragment生命周期?

1.给同学们举例：之前的 百度地图 的做法；，其实严格来说，百度地图 或者是其他第三方库 等 他们的做法是 不思进取的，如下：

```
    @Override
    protected void onStart() {
        super.onStart();
        百度地图.startActionxxx(); // 手动设置的方式来处理 生命周期管理很容易出现人为失误
    }

    @Override
```

```
    protected void onStop() {
        super.onStop();
        百度地图.stopActionxxx(); // 手动设置的方式来处理 生命周期管理 很容易出现人为失误
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        百度地图.onRelease();  // 手动设置的方式来处理 生命周期管理 很容易出现人为失误
        百度地图.recycle();
    }
```

2.100个请求，还有50个请求 在等待队列，当界面销毁时，剩余50个请求 是不是跟随着生命周期销毁掉比较合理；

3.一个请求很耗时，当回来时，发现界面都被关闭了，是不是应该停止这一切操作，来避免引发奔溃的问题；

总结：

以确保优先处理前台可见的 Activity / Fragment，提高资源利用率；

在有必要时释放资源以避免在应用在后台时被杀死，提高稳定性；

【目录二：2.生命周期作用域Application，Activity，Fragment】

第二大点：生命周期的作用域（1.Application， 2.Activity， 3.Fragment）

找到入口开始分析with源码吧：

```
private final RequestManagerRetriever requestManagerRetriever;
```

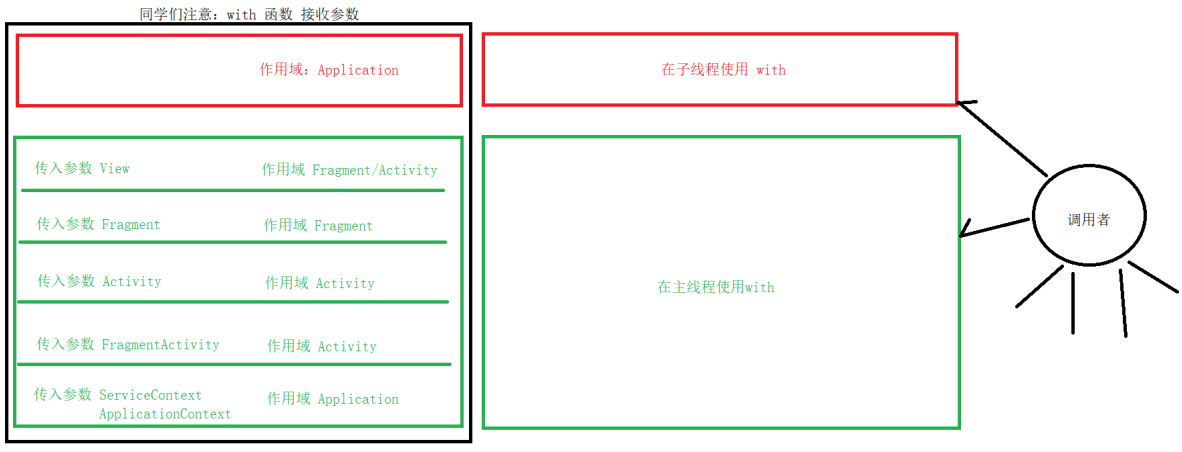
重载这些入口方法：

```
public static RequestManager with(Context context) {
    return getRetriever(context).get(context);
}

重载这些入口方法：
public static RequestManager with(Activity activity) {
    return getRetriever(activity).get(activity);
}

同学们注意：此处省略参数为 FragmentActivity、Fragment、View 的类似方法...

private static RequestManagerRetriever getRetriever(Context
context) {
    其中，Glide.get(context) 基于 DCL 单例
    return Glide.get(context).getRequestManagerRetriever();
}

public RequestManagerRetriever getRequestManagerRetriever() {
    return requestManagerRetriever;
}
```

同学们可以看到，`with(...)`方法的返回值是`RequestManager`，而真正创建的地方在`RequestManagerRetriever#get(...)`中；

请看一进一出图：

同学们呀 在前面已经看到，with是很多函数的重载，也意味着，根据传入的参数不同，将对应于 **Application Activity Fragment** 的作用域，具体如下（作用域图）：



对上图的总结： 一共分为两种：第一种是作用域Application，它的生命周期是全局的，不搞空白Fragment就绑定Activity/Fragment

第二种是作用域非Application，它的生命周期是，专门搞空白Fragment就绑定Activity/Fragment

【**1**.Application作用域源码分析（Application 域请求管理）
RequestManagerRetriever.java：】

```
  public RequestManager get(@NonNull FragmentActivity activity) {
    if (Util.isOnBackgroundThread()) {
      return get(activity.getApplicationContext()); // 同学们 会调用到下面的 get 函数
    } else {
      assertNotDestroyed(activity);
      FragmentManager fm = activity.getSupportFragmentManager();
      return supportFragmentGet(activity, fm, /*parentHint=*/ null, isActivityVisible(activity));
    }
  }
```

```
public RequestManager get(@NonNull Context context) {
    if (Util.isOnMainThread() && !(context instanceof Application)) {

        2、FragmentActivity
        if (context instanceof FragmentActivity) {
            return get((FragmentActivity) context);
        }

        3、Activity
        else if (context instanceof Activity) {
            return get((Activity) context);
        }
    }
    1、Application  // 若上面的判断都不满足，就会执行下面这句代码，同学们想知道Application作用域 就需要关心这句代码
    return getApplicationManager(context); // 会调用到 Application 作用域处理区域
  }

public RequestManager get(@NonNull FragmentActivity activity) {
    if (Util.isOnBackgroundThread()) {
```

```
        return get(activity.getApplicationContext()); // 会调用到
Application作用域处理区域
    } else {
        同学们 见下文 ...
    }
}

// 同学们注意：这里就是Application作用域处理区域
private volatile RequestManager applicationManager;
private RequestManager getApplicationManager(@NonNull Context
context) {
    源码基于 DCL 单例
    return applicationManager;
}
```

同学们，主要关注以下两点：

第一点：Application 域对应的是 applicationManager，它是与 RequestManagerRetriever 对象绑定的；

第二点：在子线程调用`get(...)`，或者传入参数是 ApplicationContext & ServiceContext 时，对应的请求是 Application 域。

【2.Activity作用域源码分析，RequestManagerRetriever.java：】

```
同学们注意，已经省略成吨代码 ...
public RequestManager get(FragmentActivity activity) {
    FragmentManager fm = activity.getSupportFragmentManager(); //
同学们 fm 是 FragmentActivity 的
    return supportFragmentGet(activity, fm, null,
isActivityVisible(activity)); // 说这个就够了
  }

public RequestManager get(Activity activity) {
    android.app.FragmentManager fm =
activity.getFragmentManager(); // 同学们 fm 是 Activity 的
    return fragmentGet(activity, fm, null,
isActivityVisible(activity));
}
```

同学们可以看到，第一个函数是不是获得了 FragmentActivity 的 FragmentManager，之后调用`supportFragmentGet(...)`获得 RequestManager。

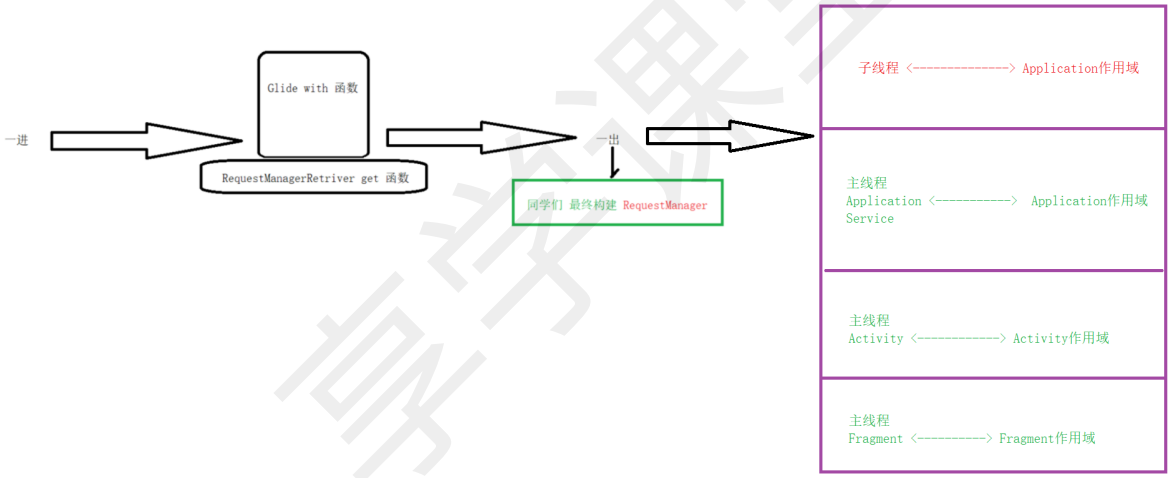
第二个函数不说了么，第二个函数和第一个函数的思路一模一样，这里就不说了哈

【3.Fragment作用域源码分析，RequestManagerRetriever.java：】

```
同学们注意，已经省略成吨代码 ...
public RequestManager get(Fragment fragment) {
    FragmentManager fm = fragment.getChildFragmentManager(); // 同学们 fm 是 Fragment 的
    return supportFragmentGet(fragment.getContext(), fm, fragment, fragment.isVisible());
}
```

同学们可以看到，这里先获得了 Fragment 的 FragmentManager（`getChildFragmentManager()`），之后调用 `supportFragmentGet(...)` 获得 RequestManager对象的哦

到这里为止，需要给同学们把，上面的 **一进一出图** 和 **作用域图** 进行衔接合成：



同学们到目前为止，，Activity 域和 Fragment 域，是不是理解了呀，Activity/Fragment/FragmentActivity作用域属于一类 都是一样的 都会搞一个空白的Fragment去监听Activity/Fragment/FragmentActivity， Application作用域是另外一类，不会搞空白的Fragment去监听；

【目录三：3.生命周期的绑定】

第三大点：生命周期的绑定：

同学们在上面的分析我们已经得知Activity 域和 Fragment 域都会调用 `supportFragmentGet(...)` 来获得 RequestManager，那么就专门分析这个方法吧 RequestManagerRetriever.java：

```
（同学们注意：这个方法必须在主线程执行，因为子线程不可能调用到这里来）

// 同学们下面这行代码用于【记录保存】 FragmentManager - SupportRequestManagerFragment 的映射关系
final Map<FragmentManager, SupportRequestManagerFragment> pendingSupportRequestManagerFragments = new HashMap<>();

private RequestManager supportFragmentGet(
      Context context,
      FragmentManager fm,
      Fragment parentHint,
      boolean isParentVisible) {

    1、从 FragmentManager 中获取 SupportRequestManagerFragment
    SupportRequestManagerFragment current =
        getSupportRequestManagerFragment(fm, parentHint, isParentVisible);

    2、从该 Fragment 中获取 RequestManager
    RequestManager requestManager = current.getRequestManager();

    3、首次获取，则实例化 RequestManager
    if (requestManager == null) {

        3.1 实例化
        Glide glide = Glide.get(context);
        requestManager = factory.build(...);

        3.2 设置 Fragment 对应的 RequestMananger
        current.setRequestManager(requestManager);
```

```
    }

    return requestManager;
}
```

getSupportRequestManagerFragment函数分析：

```
->  1、从 FragmentManager 中获取 SupportRequestManagerFragment
private SupportRequestManagerFragment
getSupportRequestManagerFragment(FragmentManager fm, Fragment
parentHint, boolean isParentVisible) {

    1.1 尝试获取 FRAGMENT_TAG 对应的 Fragment
    SupportRequestManagerFragment current =
        (SupportRequestManagerFragment)
fm.findFragmentByTag(FRAGMENT_TAG);

    if (current == null) {
        1.2 尝试从临时记录中获取 Fragment
        current = pendingSupportRequestManagerFragments.get(fm);

        1.3 实例化 Fragment
        if (current == null) {

            1.3.1 创建对象
            current = new SupportRequestManagerFragment();
            current.setParentFragmentHint(parentHint);

            1.3.2 如果父层可见，则调用 onStart() 生命周期
            if (isParentVisible) {
                current.getGlideLifecycle().onStart();
            }

            1.3.3 临时记录映射关系
            pendingSupportRequestManagerFragments.put(fm,
current);

            1.3.4 提交 Fragment 事务
            fm.beginTransaction().add(current,
FRAGMENT_TAG).commitAllowingStateLoss();

            1.3.5 post 一个消息

 handler.obtainMessage(ID_REMOVE_SUPPORT_FRAGMENT_MANAGER,
fm).sendToTarget();
```

```
            }
        }
        return current;
    }
```

-> 1.3.5 post 一个消息分析：

```
-> 1.3.5 post 一个消息
case ID_REMOVE_SUPPORT_FRAGMENT_MANAGER:

    1.3.6 移除临时记录中的映射关系
    FragmentManager supportFm = (FragmentManager) message.obj;
    key = supportFm;
    removed =
pendingSupportRequestManagerFragments.remove(supportFm);
break;
```

同学们上面三段代码中，重点关心下面三点：

第一点：从 FragmentManager 中获取 SupportRequestManagerFragment；

第二点：从该 Fragment 中获取 RequestManager；

第三点：首次获取，则实例化 RequestManager，后续从同一个 SupportRequestManagerFragment 中都获取的是这个 RequestManager；

整个的关键核心在 **getSupportRequestManagerFragment**函数：

第一步：尝试获取 FRAGMENT_TAG 对应的 Fragment

第二步：尝试从临时记录中获取 Fragment

第三步：实例化 Fragment

- 第一点： 创建对象
- 第二点：如果父层可见，则调用 onStart() 生命周期
- 第三点：临时记录映射关系
- 第四点：提交 Fragment 事务
- 第五点：post 一个消息
- 第六点：移除临时记录中的映射关系

同学们会发现上面的【记录保存】比较难理解，为什么难理解？ 因为在提交 Fragment 事务之前，为什么需要先保存记录？

就是**为了避免 SupportRequestManagerFragment 在一个作用域中重复创建。**因为`commitAllowingStateLoss()`是将事务 post 到消息队列中的，也就是说，事务是异步处理的，而不是同步处理的。假设没有临时保存记录，一旦在事务异步等待执行时调用了`Glide.with(...)`，就会在该作用域中重复创建 Fragment。

【目录四：4.生命周期的监听机制】

第四大点：生命周期的监听机制：

同学们通过上面的学习，已经明白**框架为每个Activity 和 Fragment 作用域创建了一个无UI的Fragment**，而现在我们来分析 Glide 如何监听这个无界面 Fragment 的生命周期的 SupportRequestManagerFragment.java：

```
private final ActivityFragmentLifecycle lifecycle;

public SupportRequestManagerFragment() {
    this(new ActivityFragmentLifecycle());
}

@Override
public void onStart() {
    super.onStart();
    lifecycle.onStart();
}

@Override
public void onStop() {
    super.onStop();
    lifecycle.onStop();
}

@Override
public void onDestroy() {
```

```
    super.onDestroy();
    lifecycle.onDestroy();
    unregisterFragmentWithRoot();
}

@NonNull
ActivityFragmentLifecycle getGlideLifecycle() {
    return lifecycle;
}
```

RequestManagerRetriever.java 源码的分析：

```
// 实例化 RequestManager
Glide glide = Glide.get(context);
requestManager = factory.build(glide,
current.getGlideLifecycle(),
    current.getRequestManagerTreeNode(), context);

RequestManager 工厂接口
public interface RequestManagerFactory {
    RequestManager build(
        Glide glide,
        Lifecycle lifecycle,
        RequestManagerTreeNode requestManagerTreeNode,
        Context context);
    }

默认 RequestManager 工厂接口实现类
private static final RequestManagerFactory DEFAULT_FACTORY = new
RequestManagerFactory() {
    @Override
    public RequestManager build(
        Glide glide,
        Lifecycle lifecycle,
        RequestManagerTreeNode requestManagerTreeNode,
        Context context) {
            return new RequestManager(glide, lifecycle,
requestManagerTreeNode, context);
        }
    };
}
```

RequestManager.java 源码的分析：

```
// 同学们注意：此类省略 成吨代码
```

```
final Lifecycle lifecycle;

RequestManager(Glide glide, Lifecycle lifecycle, ...){
    ...
    this.lifecycle = lifecycle;

    添加监听
    lifecycle.addListener(this);
}

@Override
public synchronized void onDestroy() {
    ...
    移除监听
    lifecycle.removeListener(this);
}
```

同学们可以看到，实例化 RequestManager 时需要一个 Lifecycle对象，这个对象是在无界面 Fragment 中创建的，当 Fragment 的生命周期变化时，就是通过这个 Lifecycle 对象将事件分发到 RequestManager

【目录五：5.生命周期的回调】

第五大点：生命周期的回调：

同学们我们来看 RequestManager 收到生命周期回调后的处理：

```
public interface LifecycleListener {
    void onStart();
    void onStop();
    void onDestroy();
}
```

RequestManager.java 回调相关的源码分析：

- Activity/Fragment 不可见时暂停请求（onStop() ）掉用函数
- Activity/Fragment 可见时恢复请求（onStart() ）掉用函数
- Activity/Fragment 销毁时销毁请求（onDestroy() ）掉用函数

```
private final RequestTracker requestTracker;

public class RequestManager
    implements ComponentCallbacks2, LifecycleListener, ... {

    @Override
    public synchronized void onStop() {
        1、onStop() 时暂停任务（页面不可见）
        pauseRequests();
        targetTracker.onStop();
    }

    @Override
    public synchronized void onStart() {
        2、onStart() 时恢复任务（页面可见）
        resumeRequests();
        targetTracker.onStart();
    }

    @Override
    public synchronized void onDestroy() {
        3、onDestroy() 时销毁任务（页面销毁）
        targetTracker.onDestroy();
        for (Target<?> target : targetTracker.getAll()) {
            clear(target);
        }
        targetTracker.clear();
        requestTracker.clearRequests();
        lifecycle.removeListener(this);
        lifecycle.removeListener(connectivityMonitor);
        mainHandler.removeCallbacks(addSelfToLifecycle);
        glide.unregisterRequestManager(this);
    }

    public synchronized void pauseRequests() {
        requestTracker.pauseRequests();
    }

    public synchronized void resumeRequests() {
        requestTracker.resumeRequests();
    }
```

```
}
```

Derry觉得分析源码，会让同学们，过几个晚上就忘记了，记忆不深刻，感觉枯燥又无味，所以Derry为了让同学们更加记忆深刻，就需要干代码，手写 生命周期代码 如下：