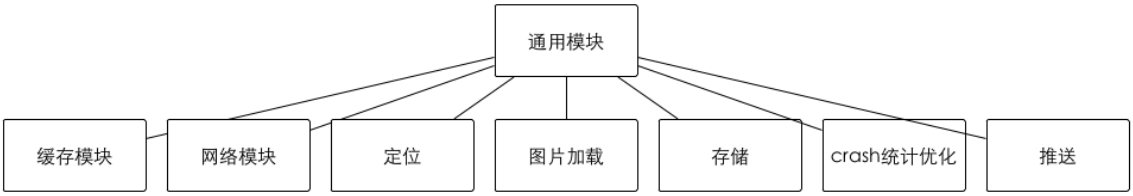
android中的flux架构

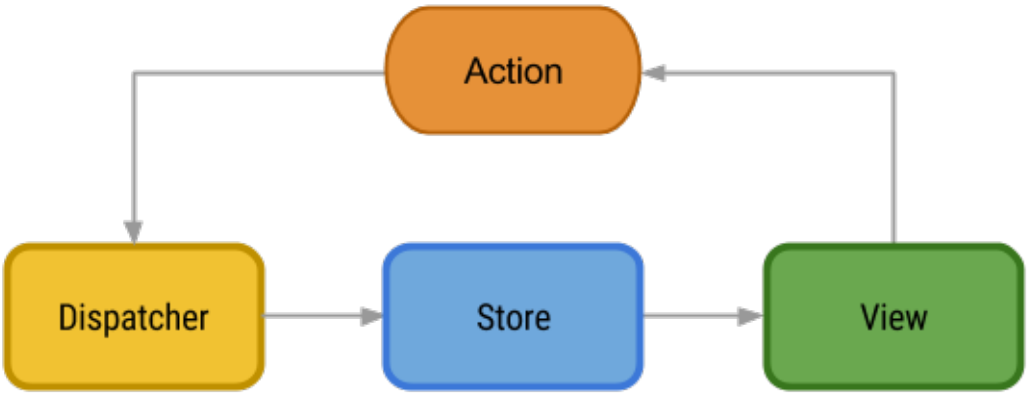
Flux 架构介绍

目前团队的框架用的是flux架构，核心的分派机制是Event Bus。

客户端的业务逻辑不会像后端一样的复杂。虽然你要处理很多平台上的问题：内存，存储，暂停，恢复，网络，定位等等，但是这些都不是业务逻辑。所有app都有这些东西。



Flux 架构被Facebook使用来构建他们的客户端web应用。跟Clean Architecture一样，它不是为移动应用设计的，但是它的特性和简单可以让我们很好的在安卓项目中采用。



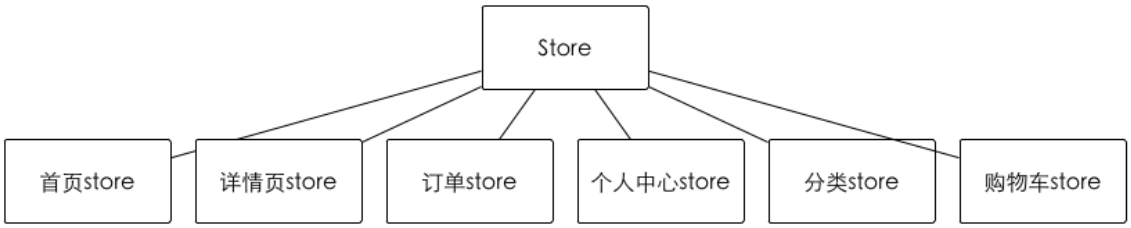
理解Flux，有两个关键的特点

- 数据流总是单向的

  一个单向的数据流 是 Flux 架构的核心，也是它简单易学的原因。就如下面讨论的，在进行应用测试的时候，它提供了非常大的帮助。
- 应用被分成三个主要部分：
  - View: 应用的界面。这里创建响应用户操作的action。
  - Dispatcher: 中心枢纽，传递所有的action，负责把它们运达每个Store。
  - Store:
    维护一个特定application domain的状态。它们根据当前状态响应action，执行业务逻辑，同时在完成的时候发出一个change事件。这个事件用于view更新其界面。

这三个部分都是通过Action来通信的：一个简单的基本对象，以类型来区分，包含了和操作相关的数据。

目前项目的store概要分类



第一步是找到Flux元素和安卓app组件之间的映射。

其中两个元素非常容易找到与实现。

- View: Activity o或者Fragment
- Dispatcher: 一个事件总线（ event bus ），在我的例子中将使用Otto，但是其它任何实现都应该是ok的。

Actions

Actions也不复杂。它们的实现和POJO一样简单，有两个主要属性：

- Type: 一个String，定义了事件的类型。
- Data: 一个map，装载了本次操作。

比如，一个显示用户详情的典型action如下：

```
1 Bundle data = new Bundle();
2 data.put("USER_ID", id);
3 Action action = new ViewAction("SHOW_USER", data);
```

Stores

这可能是Flux理论中最难的部分。

如果你之前使用过Clean Architecture，你可能难以接受。因为Stores承担了原本被分成多层的责任。

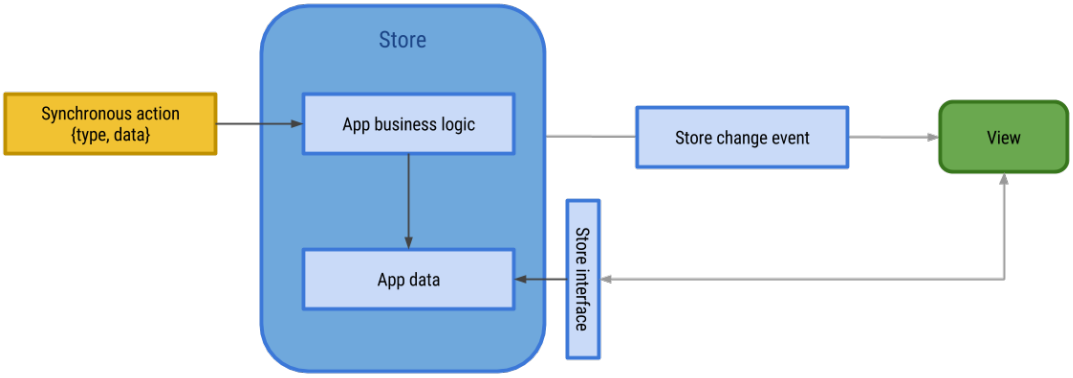
Stores包含了application的状态与它的业务逻辑。它们类似于rich data models但是可以管理多个对象的状态，而不仅仅是一个对象。

Stores响应Dispatcher发出的Action，执行业务逻辑并发送change事件。

Stores的唯一输出是这单一的事件：change。其它对Store内部状态感兴趣的组件必须监听这个事件，同时使用它获取需要的数据。

系统中不再需要任何其它组建去了解application的任何状态信息。

最后，stores必须对外公开一个获取application状态的接口。这样，view元素可以查询Stores然后相应的更新UI。



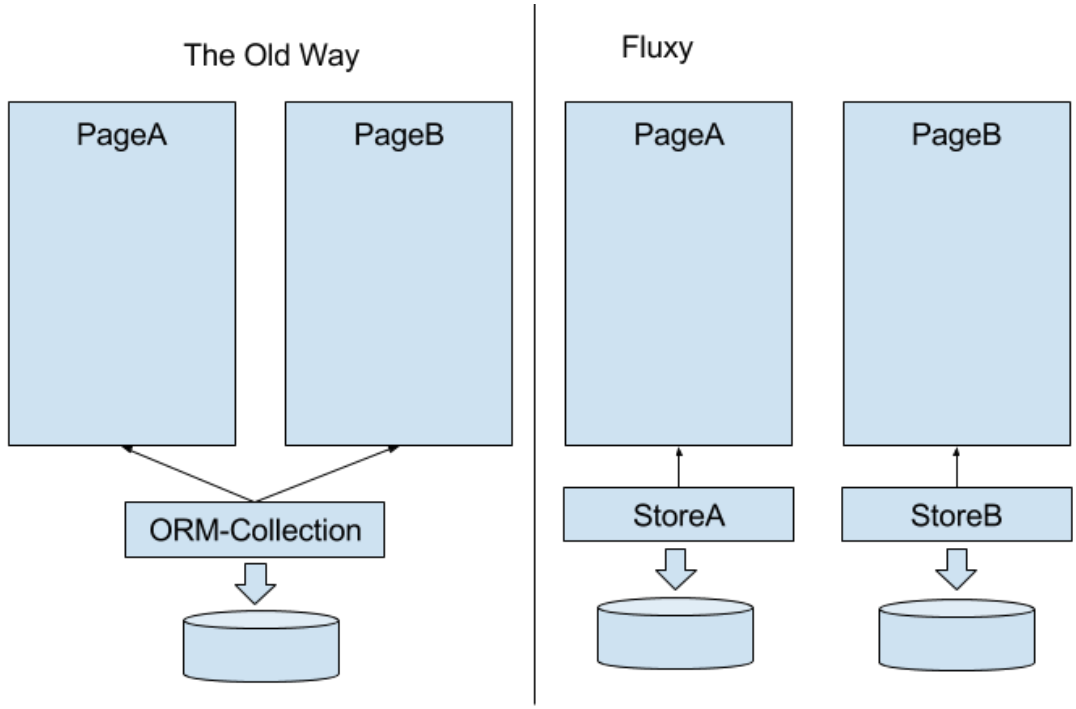
比如，在一个Pub Discovery App中，SearchStore被用来跟踪被搜索的item，搜索结果以及搜索历史。在同一个应用中，一个ReviewedStore同样包含了浏览pub的列表以及必要的逻辑比如根据review排序。

但是有一个重要的概念需要记住：Stores并不是仓库。它们的职责不是从一个外部源（API或者数据库）获取数据，而是跟踪actions提供的数据。

Store数据缓存

在一般的Android应用开发中，本地数据存储是很重要的一部分。

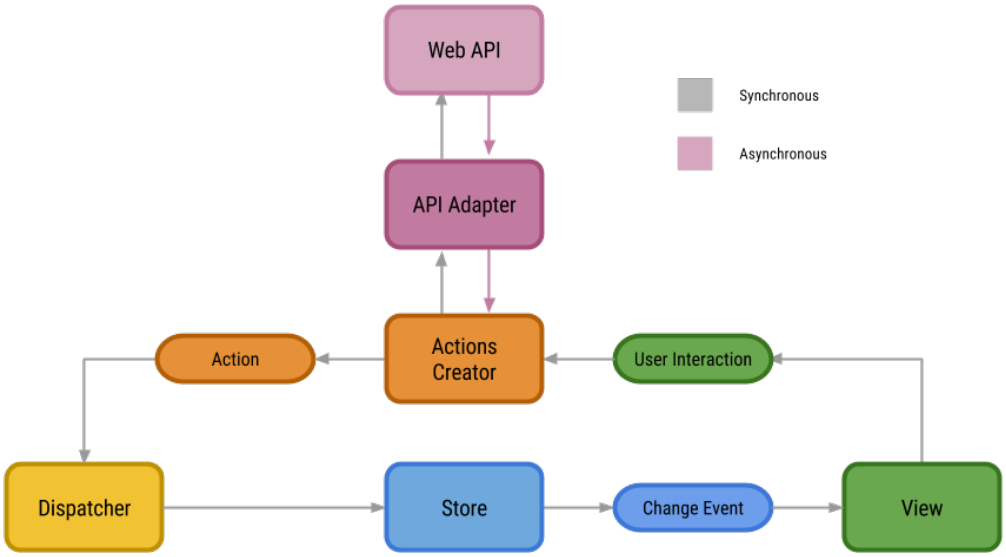
把存储放在Store中，仅仅是Store在维护App的UI状态的手段而已。而实现方式，根据数据类型不同有可能采用DB来缓存（比如，大量的列表型数据）、或者SharedPreferences来缓存。



那么，Flux application是如何获得数据的呢？

网络请求与异步调用

在第一幅Flux示意图中我有意跳过了一部分：网络调用。接下来的示意图完善第一幅图并添加了更多细节：



异步网络调用是被一个Actions Creator触发的。一个Network 适配器完成相应API的异步调用并且返回结果给Actions Creator。

最终Actions Creator分发带有返回数据的相应类型的Action。

把所有网络工作和异步工作独立于Stores之外有两个主要的优点：

- 你的Stores是完全同步的：这让Store中的逻辑更容易跟踪。Bug也更容易跟踪。同时，因为所有的状态变化都是同步的，那么Store的测试变会的非常简单：启动actions然后等待期望的结果。
- 所有的action都是从一个Action Creator触发的：在一处单一的点创建与发起所有用户操作可以大大简化寻找错误的过程。忘掉在多个类中寻找某个操作的源头吧，所有的事情都是在这里发生的。同时，因为异步调用发生在这之前，所有来自于ActionCreator的东西都是同步的。这大大提高了代码的可跟踪与可测试性。