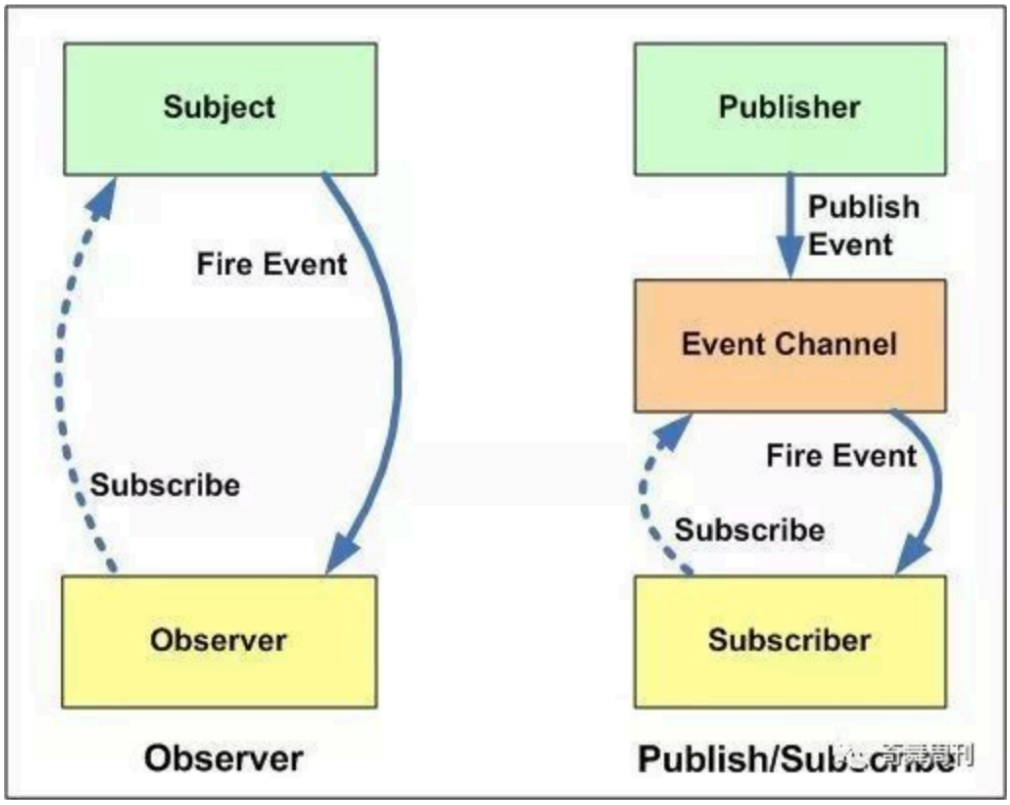
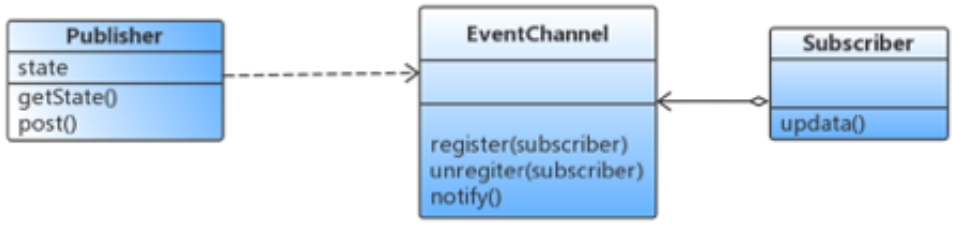


开局一张图，故事全靠编。

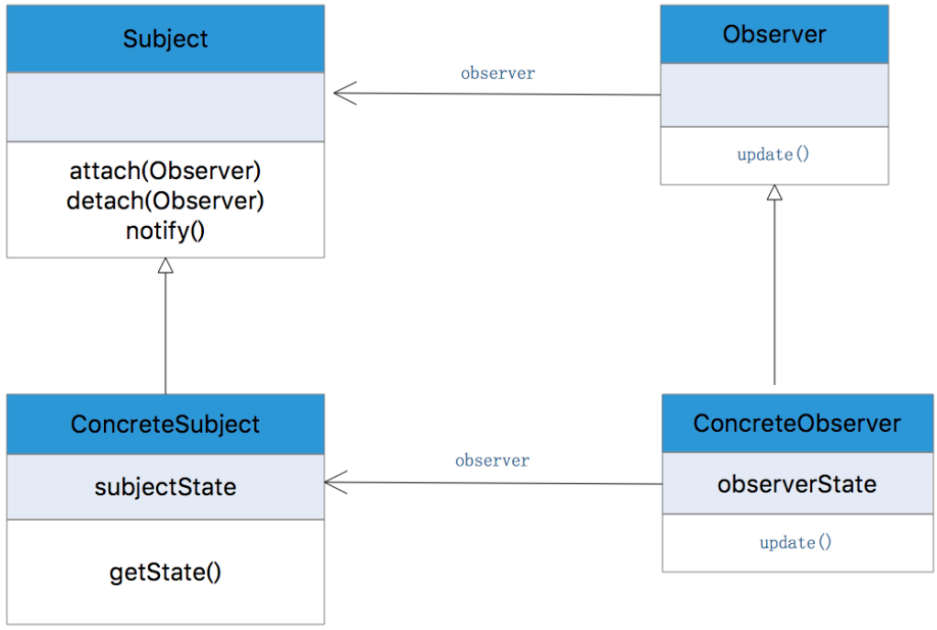
1.EventBus源于订阅发布模式。

在订阅发布模式（上图右）中，发布者和订阅着互相不感知对方的存在，双方通过消息代理进行通信，各组件间松耦合。Eventbus的作用正如上图的EventChannel，它提供的功能更是一种总线机制，甚至可以说是路由机制。发布者将消息发布到总线Eventbus上，剩下的工作交有Eventbus来处理。订阅者被EventBus持有和维护，EventBus将消息一一发布给订阅者。正因此，我们项目中，一版可以显示找到订阅者入activity，但是发布者则隐藏于各种，随处都可以是发布者，随处都可以post出来消息。

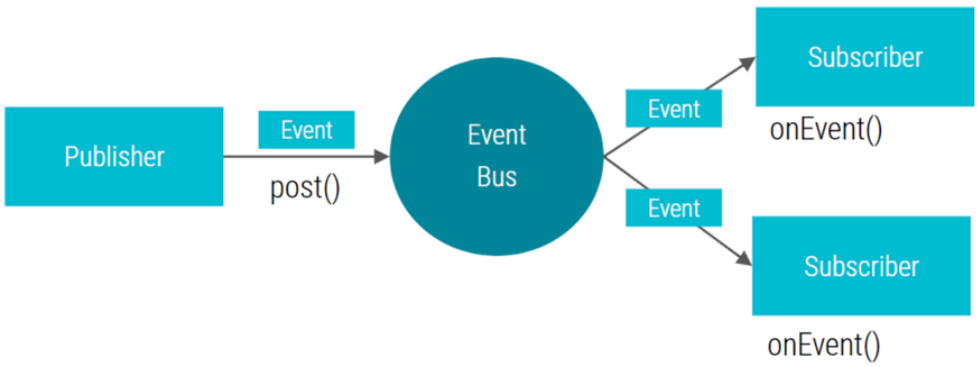
而在观察者模式（上图左）中，观察者和被观察者没有完全解耦，抽象被观察者持有抽象观察者，并维护观察者列表。与订阅模式相当于在观察者和被观察者之间架了一层，由这个中间层来持有观察者，这样被观察者无须感知观察者。



（订阅发布模式）



（观察者模式）



（EventBus核心结构）

2.register过程

```
public void register(Object subscriber) {
    Class<?> subscriberClass = subscriber.getClass();
    List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscriberMethods(subscriberClass);
    synchronized (this) {
        // 因为subscriptionsByEventType  Evenbus只维护了一个，不同线程所有的注册者都在这个map里，故对它的读取都要加锁
        for (SubscriberMethod subscriberMethod : subscriberMethods) {
            subscribe(subscriber, subscriberMethod);
        }
    }
}
```

通过findSubscriberMethods查找回来一个包含订阅者所有订阅方法的订阅列表。跟进findSubscriberMethods如何查找。

```
List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {
    //METHOD_CACHE是个Map<Class<?>, List<SubscriberMethod>>, key为订阅者类，值为订阅者的订阅方法
    //作用是将查找过的订阅者类和它的订阅方法缓存起来，下次再注册时直接能找到该订阅者所有的订阅方法
    List<SubscriberMethod> subscriberMethods = METHOD_CACHE.get(subscriberClass);
    if (subscriberMethods != null) {
        return subscriberMethods;
    }

    if (ignoreGeneratedIndex) {
        //如果设置忽略索引，则直接通过运行时反射去订阅者里遍历查找订阅方法
        subscriberMethods = findUsingReflection(subscriberClass);
    } else {
        //开启索引的时，通过索引去查找
        subscriberMethods = findUsingInfo(subscriberClass);
    }
    {
        //缓存订阅者类和它的订阅方法
        METHOD_CACHE.put(subscriberClass, subscriberMethods);
        return subscriberMethods;
    }
}
```

通过运行时反射遍历查找

```
private List<SubscriberMethod> findUsingReflection(Class<?> subscriberClass) {
    //FindState的作用的查找过程中保存查找结果
    //同时通过大小为4的FIND_STATE_POOL缓存池缓存FindState，避免频繁的创建对象
    FindState findState = prepareFindState();
    findState.initForSubscriber(subscriberClass);
    while (findState.clazz != null) {
        findUsingReflectionInSingleClass(findState);
        findState.moveToSuperclass();
    }
    return getMethodsAndRelease(findState);
}
```

```java
private void findUsingReflectionInSingleClass(FindState findState) {
    Method[] methods;
    try {
        // This is faster than getMethods, especially when subscribers are fat classes like Activities
        //getDeclaredMethods是获得本类中所有的方法包括私有的(private、protected、默认以及public)的方法。
        // getMethods()是获得本类及父类或父接口中的所有public方法
        methods = findState.clazz.getDeclaredMethods();
    } catch (Throwable th) {
        methods = findState.clazz.getMethods();
        findState.skipSuperClasses = true;
    }
    for (Method method : methods) {
        int modifiers = method.getModifiers();
        //只支持订阅方法必须是public类型
        if ((modifiers & Modifier.PUBLIC) != 0 && (modifiers & MODIFIERS_IGNORE) == 0) {
            Class<?>[] parameterTypes = method.getParameterTypes();
            if (parameterTypes.length == 1) {
                //只有方法参数个数是1个的情况下才有可能是我们的订阅方法
                //通过注解找到订阅方法
                Subscribe subscribeAnnotation = method.getAnnotation(Subscribe.class);
                if (subscribeAnnotation != null) {
                    //eventType是事件类型
                    Class<?> eventType = parameterTypes[0];
                    if (findState.checkAdd(method, eventType)) {
                        ThreadMode threadMode = subscribeAnnotation.threadMode();
                        //找到订阅方法并加入该类型订阅者方法队列
                        findState.subscriberMethods.add(new SubscriberMethod(method, eventType, threadMode,
                                subscribeAnnotation.priority(), subscribeAnnotation.sticky()));
                    }
                }
            }
        }
    }
}
```

如果设置了索引加速，通过索引查找。原理是EvetBusAnnotationProcessor。

```java
private List<SubscriberMethod> findUsingInfo(Class<?> subscriberClass) {
    FindState findState = prepareFindState();
    findState.initForSubscriber(subscriberClass);
    while (findState.clazz != null) {
        //通过索引去查找
        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 {
            //索引查找未找到，降级运行时反射查找
            findUsingReflectionInSingleClass(findState);
        }
        findState.moveToSuperclass();
    }
    return getMethodsAndRelease(findState);
}
```

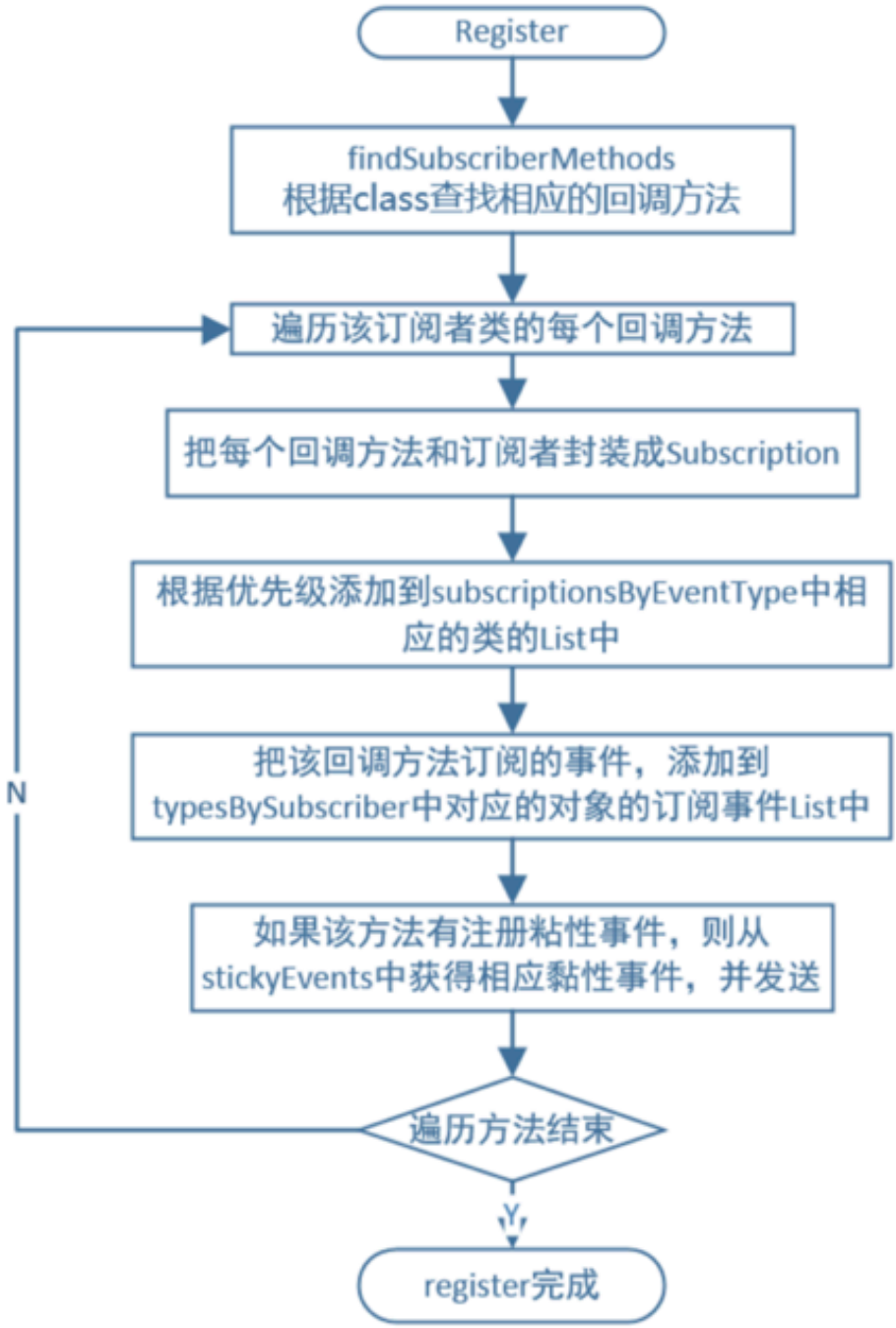
找打订阅方法后，开始进行一系列的存储操作

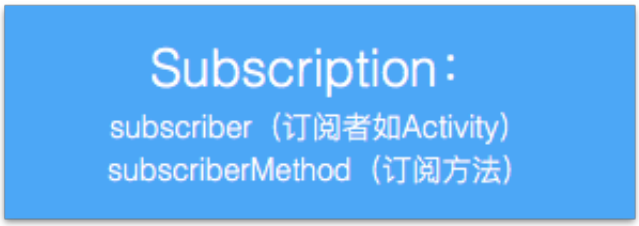
```java
// Must be called in synchronized block
private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {
    Class<?> eventType = subscriberMethod.eventType;
    //将订阅者对象和它里面的订阅方法一一封装成Subscription对象
    Subscription newSubscription = new Subscription(subscriber, subscriberMethod);
    //subscriptionsByEventType Map<Class<?>, CopyOnWriteArrayList<Subscription>>
    //以事件类型为key, List<Subscription>为value存储订阅者和订阅方法
    CopyOnWriteArrayList<Subscription> subscriptions = subscriptionsByEventType.get(eventType);
    if (subscriptions == null) {
        subscriptions = new CopyOnWriteArrayList<>();
        subscriptionsByEventType.put(eventType, subscriptions);
    } else {
        //检查是否已经包含Subscription
        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++) {
        //根据优先级加入CopyOnWriteArrayList<Subscription> subscriptions
        if (i == size || subscriberMethod.priority > subscriptions.get(i).subscriberMethod.priority) {
            subscriptions.add(i, newSubscription);
            break;
        }
    }
```

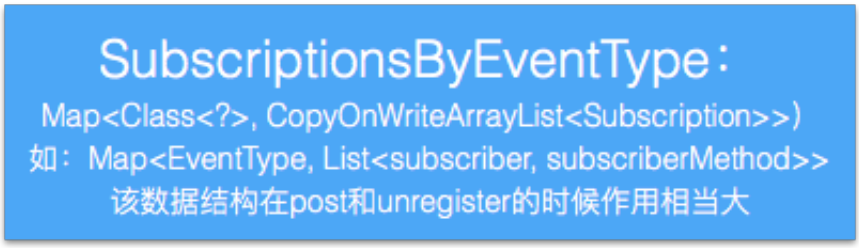
```
    //typesBySubscriber Map类型Map<Object, List<Class<?>>>
    //以订阅者对象为key，订阅者对象所订阅的所有事件类型（注意是事件类）组成的列表为value
    //该数据结构主要是在解注册的时候方便查找订阅者和订阅方法去删除
    List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber);
    if (subscribedEvents == null) {
        subscribedEvents = new ArrayList<>();
        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 sticky events,
            // thus data structure should be changed to allow a more efficient lookup
            // (e.g. an additional map storing sub classes of super classes: Class -> 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 {
            //如果事件为黏性属性，则需要去stickyEvents里取出存储的黏性事件立即发布该事件
            //只有当前注册的订阅者才会执行该黏性事件
            Object stickyEvent = stickyEvents.get(eventType);
            checkPostStickyEventToSubscription(newSubscription, stickyEvent);
        }
    }
}
```

如果是黏性属性的方法，则立即去缓存黏性事件的stickyEvents查找是否有黏性事件，有则立即post到当前订阅者里执行，且只有当前订阅者会执行一次，其他地方不会执行，也没有查找订阅者这一过程。

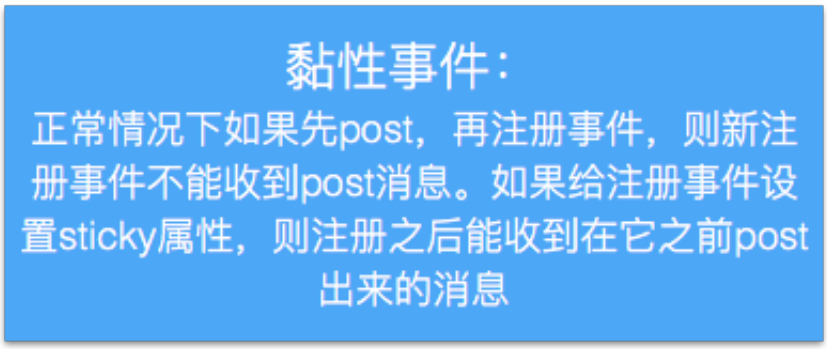
register小结


Register
findSubscriberMethods
根据class查找相应的回调方法
遍历该订阅者类的每个回调方法
把每个回调方法和订阅者封装成Subscription
根据优先级添加到subscriptionsByEventType中相应的类的List中
把该回调方法订阅的事件，添加到typesBySubscriber中对应的对象的订阅事件List中
如果该方法有注册粘性事件，则从stickyEvents中获得相应黏性事件，并发送
遍历方法结束
N
Y
register完成


Subscription：
subscriber（订阅者如Activity）
subscriberMethod（订阅方法）


SubscriptionsByEventType：
Map<Class<?>, CopyOnWriteArrayList<Subscription>>)
如：Map<EventType, List<subscriber, subscriberMethod>>
该数据结构在post和unregister的时候作用相当大


TypesBySubscriber：
Map<Object, List<Class<?>>>
如：Map<Activity, List<EventType>>
该数据结构在unregister的时候作用相当大


黏性事件：
正常情况下如果先post，再注册事件，则新注册事件不能收到post消息。如果给注册事件设置sticky属性，则注册之后能收到在它之前post出来的消息

3.post过程

```java
public void post(Object event) {
    //PostingThreadState利用ThreadLocal线程本地变量来存储，则不存在并发问题。
    //A线程post出来的事件会存储在A线程的事件队列里，不会出现在B线程的事件队列里
    PostingThreadState postingState = currentPostingThreadState.get();
    List<Object> eventQueue = postingState.eventQueue;
    eventQueue.add(event);

    if (!postingState.isPosting) {
        //根据Looper判断是不是主线程
        postingState.isMainThread = isMainThread();
        //设置正在post的状态，后续post事件加入队列等待
        postingState.isPosting = true;
        if (postingState.canceled) {
            throw new EventBusException("Internal error. Abort state was not reset");
        }
        try {
            //一直消费线程队列里post出来的事件直到所有事件分发完
            while (!eventQueue.isEmpty()) {
                postSingleEvent(eventQueue.remove( i: 0), postingState);
            }
        } finally {
            postingState.isPosting = false;
            postingState.isMainThread = false;
        }
    }
}
```

```java
private void postSingleEvent(Object event, PostingThreadState postingState) throws Error {
    Class<?> eventClass = event.getClass();
    boolean subscriptionFound = false;
    if (eventInheritance) {
        //查找事件类型的父类
        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) {
        //没有订阅者则发出一个NoSubscriberEvent
        if (logNoSubscriberMessages) {
            logger.log(Level.FINE, msg: "No subscribers registered for event " + eventClass);
        }
        if (sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class &&
                eventClass != SubscriberExceptionEvent.class) {
            post(new NoSubscriberEvent( eventBus: this, event));
        }
    }
}
```

```java
// 开始真正分发事件
private boolean postSingleEventForEventType(Object event, PostingThreadState postingState, Class<?> eventClass) {
    CopyOnWriteArrayList<Subscription> subscriptions;
    synchronized (this) {
        // 通过事件类型EventType找到该事件类型对应的Subscription队列
        // subscription = new Subscription(subscriber, subscriberMethod);
        subscriptions = subscriptionsByEventType.get(eventClass);
    }
    if (subscriptions != null && !subscriptions.isEmpty()) {
        for (Subscription subscription : subscriptions) {
            // 遍历Subscription队列，分发事件到具体每个订阅者，执行相应的订阅方法
            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;
}
```

```java
//具体事件分发，执行
private void postToSubscription(Subscription subscription, Object event, boolean isMainThread) {
    switch (subscription.subscriberMethod.threadMode) {
        case POSTING:
            //哪个线程post出来则在哪个线程执行
            invokeSubscriber(subscription, event);
            break;
        case MAIN:
            // 主线程
            if (isMainThread) {
                invokeSubscriber(subscription, event);
            } else {
                mainThreadPoster.enqueue(subscription, event);
            }
            break;
        case BACKGROUND:
            //子线程post出来则子线程中执行
            //主线程中post出来则加入子线程队列去执行，这时候这个子线程可以是EventBus默认线程池里的一条专门的线程
            // 可以是你在Builder里手动配置的线程池的线程，如果你有配置的话
            if (isMainThread) {
                backgroundPoster.enqueue(subscription, event);
            } else {
                invokeSubscriber(subscription, event);
            }
            break;
        case ASYNC:
            //无论哪个线程post出来，都会由一条空闲线程执行，线程来源同BACKGROUND
            asyncPoster.enqueue(subscription, event);
            break;
        default:
            throw new IllegalStateException("Unknown thread mode: " + subscription.subscriberMethod.threadMode);
    }
}
```

ThreaMode说明

PostThread：默认的 ThreadMode，表示在执行 Post 操作的线程直接调用订阅者的事件响应方法（哪个线程post出来就在哪个线程中执行），不论该线程是否为主线程（UI 线程）。当该线程为主线程时，响应方法中不能有耗时操作，否则有卡主线程的风险。适用场景：对于是否在主线程执行无要求，但若 Post 线程为主线程，不能耗时的操作；

MainThread：在主线程中执行响应方法。如果发布线程就是主线程，则直接调用订阅者的事件响应方法，否则通过主线程的 Handler 发送消息在主线程中处理——调用订阅者的事件响应函数。显然，MainThread类的方法也不能有耗时操作，以避免卡主线程。适用场景：必须在主线程执行的操作；

BackgroundThread：在后台线程中执行响应方法。如果发布线程不是主线程，则直接调用订阅者的事件响应函数，否则启动唯一的后台线程去处理。由于后台线程是唯一的，当事件超过一个的时候，它们会被放在队列中依次执行，因此该类响应方法虽然没有PostThread类和MainThread类方法对性能敏感，但最好不要有重度耗时的操作或太频繁的轻度耗时操作，以造成其他操作等待。适用场景：操作轻微耗时且不会过于频繁，即一般的耗时操作都可以放在这里；

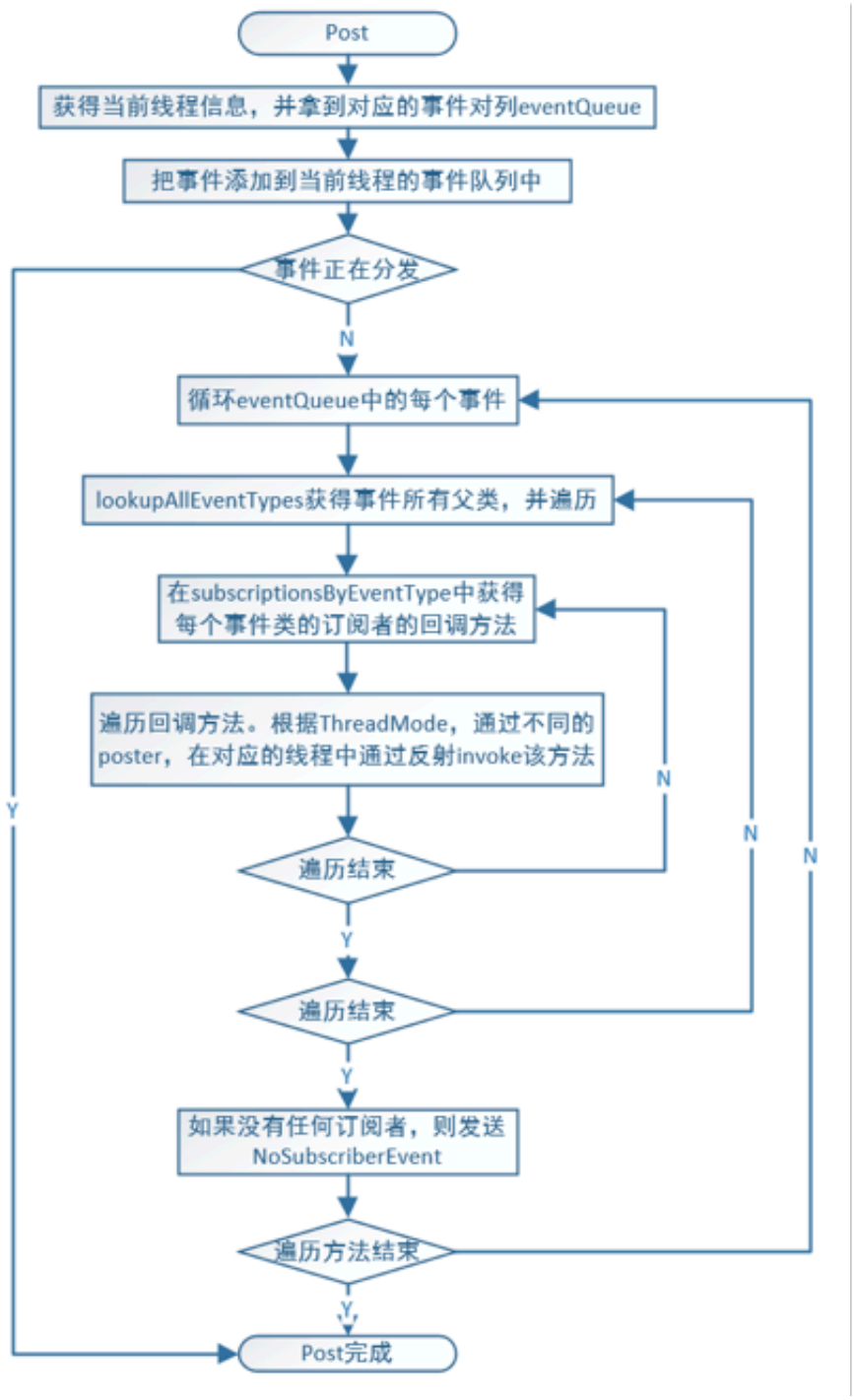
Async：不论发布线程是否为主线程，都使用一个空闲线程来处理。和BackgroundThread不同的是，Async类的所有线程是相互独立的，因此不会出现卡线程的问题。适用场景：长耗时操作，例如网络访问。

再来看看黏性事件的post过程

```
public void postSticky(Object event) {
    synchronized (stickyEvents) {
        //先将黏性事件存储到管理黏性事件的map里
        stickyEvents.put(event.getClass(), event);
    }
    // Should be posted after it is putted, in case the subscriber wants to remove immediately
    //像正常事件一样投递黏性事件，这样订阅方法里未指定黏性属性的也能收到该黏性事件
    post(event);
}
```

注意看，这里有个加锁的过程，为什么postSticky需要加锁，而post不需要加锁？因为postSticky的事件会先存储到Map<Class<?>, Object> stickyEvents中，而stickyEvents是EventBus全局只有一个，也就是所有的黏性事件都存储在这个map中，故当不同线程同时poststicky事件的时候存在并发问题。而普通的post出来的事件，都是存储在线程的本地变量的事件队列里，各线程互不干扰互相不能访问对方的数据，故不存在并发问题。

post小结




```
/** For ThreadLocal, much faster to set (and get multiple values). */
final static class PostingThreadState {
    final List<Object> eventQueue = new ArrayList<>(); // 当前线程的post事件队列，一旦触发post会消费当前线程下队列里的所有事件
    boolean isPosting;
    boolean isMainThread;  // 当前post线程是否是主线程
    Subscription subscription; // subscription = new Subscription(subscriber, subscriberMethod);
    Object event;
    boolean canceled;
}
```

ThreadLocal:线程用来存储私有变量（一个很有意思的东西）

名义上以ThreadLocalMap变量形式在线程内部,但是底层实现是基于Entry[]数组而不是HashMap。(这里引发一个思考，为什么要用数组实现，其实数组也可以称一种map，用数组实现开放定址法处理冲突，
用数组存储key和value更节省内存，普通的HashMap是拉链法解决冲突，基于数组和链表，每个Entry元素除了key和value还要一个Entry类型的next指针，占用更多内存。同样的思想也在Android的sparseArray和ArrayMap中使用。)

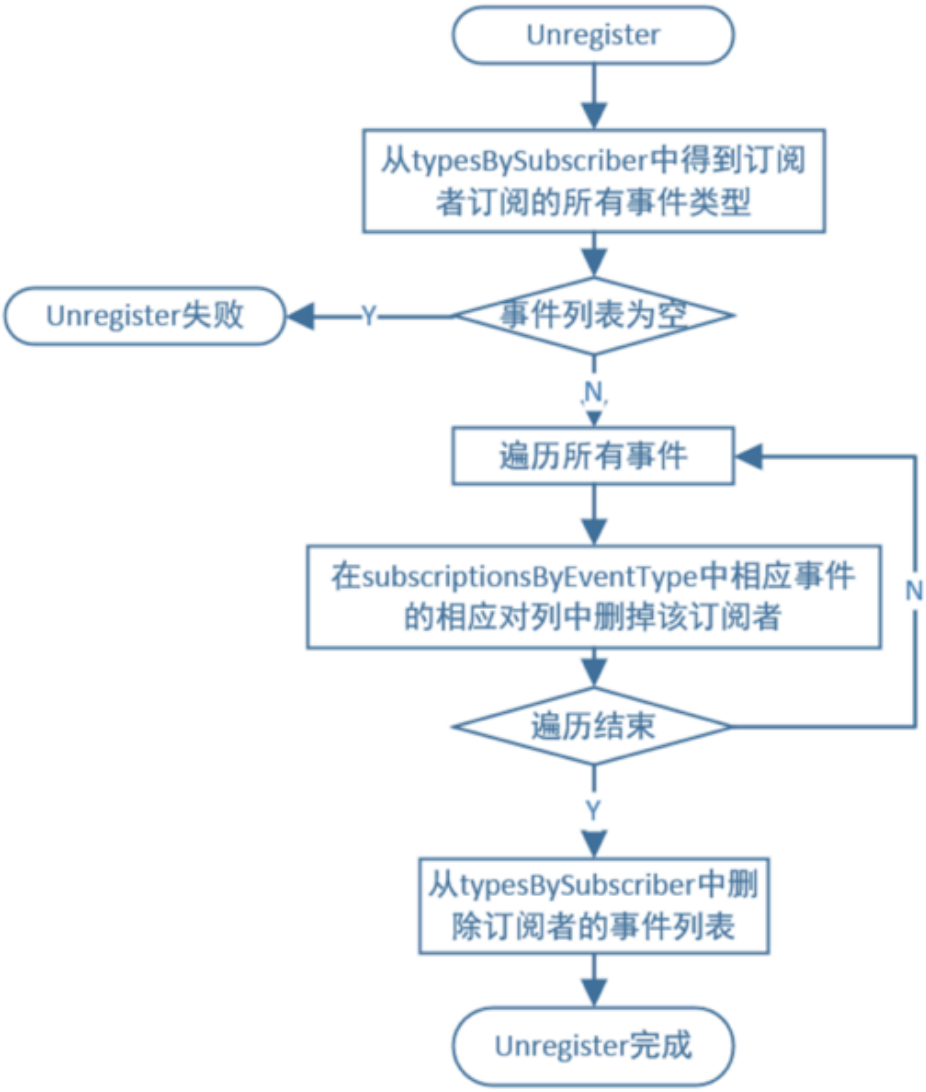
Entry{
Object key； // key实际上是我们定义的ThreadLocal对象，而非当前线程对象
Object value；
}

```
private final ThreadLocal<PostingThreadState> currentPostingThreadState = new ThreadLocal<PostingThreadState>() {
    @Override
    protected PostingThreadState initialValue() {
        return new PostingThreadState();
    }
};
```

4.unregister

```
public synchronized void unregister(Object subscriber) {
    //通过订阅者对象找到该订阅者里所有订阅过的事件类型（注意是事件类型）
    List<Class<?>> subscribedTypes = typesBySubscriber.get(subscriber);
    if (subscribedTypes != null) {
        for (Class<?> eventType : subscribedTypes) {
            unsubscribeByEventType(subscriber, eventType);
        }
        typesBySubscriber.remove(subscriber);
    } else {
        logger.log(Level.WARNING, msg: "Subscriber to unregister was not registered before: " + subscriber.getClass());
    }
}
```

```
private void unsubscribeByEventType(Object subscriber, Class<?> eventType) {
    //通过事件类型找到该事件类型对应的List<Subscription>
    List<Subscription> subscriptions = subscriptionsByEventType.get(eventType);
    if (subscriptions != null) {
        int size = subscriptions.size();
        for (int i = 0; i < size; i++) {
            //一一删除列表上的订阅者和订阅方法
            Subscription subscription = subscriptions.get(i);
            if (subscription.subscriber == subscriber) {
                subscription.active = false;
                subscriptions.remove(i);
                i--;
                size--;
            }
        }
    }
}
```



5思考&总结

优点：

1.事件总线通信，使用简单

2.解耦，干脆利落

...

缺点：

1.极致的解耦导致项目维护和阅读难度增大，出现EventBus满天飞的场景

2.更甚者，它使我们往往懒于去代码中找寻设计的快感

3.如果不及时unregister则会内存泄露

...

关于性能：

1.EventBus3.0以前大量使用反射，存在性能瓶颈，3.0以后引入APT（注解处理器）后在编译期解析注解，性能飙升

2.EventBus在每次查找到订阅者和其绑定的订阅方法集合后会放在Map里缓存。（这是注解框架的一贯套路，Butterknife亦如是）