Trường ĐH Khoa Học Tự Nhiên Tp. Hồ Chí Minh
TRUNG TÂM TIN HỌC

Lập trình Android

Bài 36: *Thư viện Event Bus*

Phòng LT & Mạng

http://csc.edu.vn/lap-trinh-va-csdl

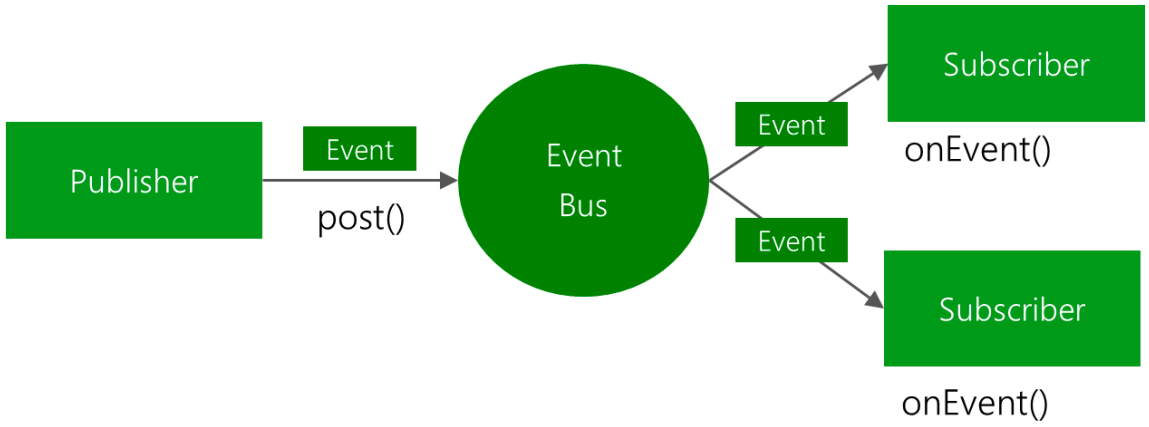
2019

Nội dung

1. Giới thiệu EventBus
2. Các trường hợp sử dụng EventBus
3. Minh hoạ sử dụng EventBus
4. Post Event vs. Post Sticky Event
5. Các loại EventBus Thread
6. Độ ưu tiên trong Event Bus
7. Huỷ gửi Event tới các subscriber khác

Giới thiệu

- Có nhiều cách để truyền dữ liệu giữa các thành phần trong ứng dụng: Intent, Listener, BroadcastReceiver …
- **EventBus** là một thư viện mã nguồn mở cho Android và Java sử dụng mô hình **publisher/subscriber**
- **EventBus** cho phép xử lý tập trung truyền thông bên trong ứng dụng Android ngắn gọn hơn, giúp việc lập trình nhanh chóng hơn, dễ dàng kết nối giữa 2 thành phần trong ứng dụng
- Truyền thông tin từ thành phần A đến thành phần B trong ứng dụng với **EventBus**:
  - Thành phần A đóng vài trò là **Publisher**, gửi (**post()**) một **event** tới trung tâm **EventBus**
  - **EventBus** sẽ chuyển tiếp **event** tới các các **Subscriber**, là các thành phần đã đăng ký nhận **event** từ **EventBus** trước đó (trong đó có thành phần B)
  - Các **Subscriber** sẽ nhận **event** từ **EventBus** gởi tới (tại **onEvent()**) và xử lý thông tin của event nhận được



Các trường hợp sử dụng EventBus

- **EventBus** có thể thay thế cho các trường hợp sau:
  - Sử dụng Intent để mở một activity khác trong cùng ứng dụng và truyền thông tin thông qua extra của Intent
  - Sử dụng Bundle để truyền dữ liệu vào fragment khi khởi tạo fragment
  - Sử dụng mẫu thiết kế listener đối với fragment để thực hiện một phương thức của listener gửi kèm theo dữ liệu từ fragment
  - Các service dựa trên BroadcastReceiver để gửi thông tin đến các thành phần khác của ứng dụng

Minh hoạ sử dụng EventBus

- Minh hoạ sau mô tả tổng quan cách thức truyền dữ liệu từ thành phần activity A tới thành phần activity B trong cùng một ứng dụng, sử dụng **EventBus**.
- Cấu hình **dependencies** trong **build.gradle** của **module**:

```
dependencies {
    // …
    implementation 'org.greenrobot:eventbus:3.1.1'
}
```

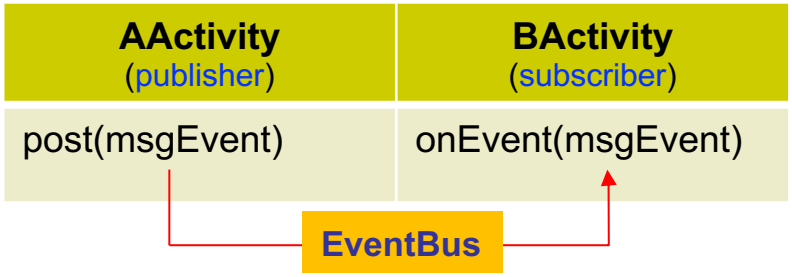
- Định nghĩa model class **MessageEvent**:

```
public class MessageEvent {
    static final String FROM_A_TO_B = "FROM_A_TO_B";
    String message;
    String type;

    public MessageEvent(String message, String type) {
        this.message = message;
        this.type = type;
    }
}
```

Minh hoạ sử dụng EventBus (2)

- Từ activity A, thực hiện **Intent** để chuyển tới activity B trước đó, gửi **event** thông qua phương thức **post()** tới **EventBus:**




```
Intent intent = new Intent(AActivity.this, BActivity.class);
startActivity(intent);
EventBus.getDefault().postSticky(new MessageEvent("Hello B", MessageEvent.FROM_A_TO_B));
```

- Tại activity B, đăng ký nhận **event** từ **EventBus** trong onStart() và huỷ nhận **event** trong onStop():

```
@Override
public void onStart() {
    super.onStart();
    EventBus.getDefault().register(this);
}
```

```
@Override
public void onStop() {
    super.onStop();
    EventBus.getDefault().unregister(this);
}
```

- Khi activity B nhận được **event**, phương thức **onEvent()** sẽ được gọi:

```
@Subscribe(sticky = true, threadMode = ThreadMode.MAIN)
public void onEvent(MessageEvent event) {
    if (event != null && MessageEvent.FROM_A_TO_B.equals(event.type)) {
        tvMessage.setText(event.message);
        EventBus.getDefault().removeStickyEvent(event);
    }
}
```

Post Event vs. Post Sticky Event

- Cả **post()** và **postSticky()** đều dùng để gửi đi một event tới EventBus

```
EventBus.getDefault().post(new MessageEvent("Hello B", MessageEvent.FROM_A_TO_B));

EventBus.getDefault().postSticky(new MessageEvent("Hello B", MessageEvent.FROM_A_TO_B));
```

- **postSticky()** giúp event được lưu trong **cache**. Khi một activity/fragment mới được tạo ra và đăng ký nhận event từ EventBus, nó sẽ nhận sticky event mới nhất từ cache thay vì phải chờ EventBus gửi lại event
- Thông thường, chỉ cần dùng **post()**, nếu không subscriber nào được tìm thấy, event sẽ bị huỷ bỏ
- Chỉ dùng sticky event khi subscriber được tạo ra sau, khi đó cần lưu trước event ở trong cache
- Cần chỉ rõ việc nhận sticky event ở annotation **@Subscrible** của phương thức **onEvent()**

```
@Subscribe(sticky = true, threadMode = ThreadMode.MAIN)
public void onEvent(MessageEvent event) {

}
```

- Để xoá sticky event từ cache, gọi hàm removeStickyEvent(Event), hoặc gọi removeAllStickyEvents() để xoá hết tất cả sticky events

```
EventBus.getDefault().removeStickyEvent(event);
```

Các loại EventBus Thread

- **POSTING** – là trường hợp mặc định. Các subscriber sẽ nhận event trong cùng thread với nơi event được gửi

  @Subscribe(**threadMode** = ThreadMode.POSTING)

- **MAIN** – các subscriber sẽ nhận event ở main UI thread, bất chấp nơi gửi event từ đâu

  @Subscribe(**threadMode** = ThreadMode.MAIN)

- **BACKGROUND** – các subscriber sẽ nhận event trong cùng thread với nơi gửi event. Tuy nhiên, nếu nơi gửi event là main UI thread, thì subscriber sẽ nhận event tại background thread

  @Subscribe(**threadMode** = ThreadMode.BACKGROUND)

- **ASYNC** – các subscriber sẽ nhận event ở một thread độc lập khác, không cùng nơi với thread hiện tại hay main UI thread

  @Subscribe(**threadMode** = ThreadMode.ASYNC)

Độ ưu tiên trong Event Bus

- Nếu muốn thay đổi độ ưu tiên hay thứ tự trong việc nhận event của các subscriber khi có nhiều subscriber cùng đăng ký nhận event, có thể thiết lập chỉ số độ ưu tiên (priority) cho từng subscriber
- Mặc định độ ưu tiên (priority) bằng 0
- Độ ưu tiên càng cao nếu chỉ số priority càng nhỏ

```
@Subscribe(priority = 1);
public void onEvent(MessageEvent event) {


}
```

Huỷ gửi Event tới các subscriber khác

- Để huỷ việc gửi event từ EventBus đến các subscriber khác, gọi phương thức **cancelEventDelivery()** trong phương thức **onEvent()** của subscriber đang xử lý event đó

```
@Subscribe
public void onEvent(MessageEvent event){
    EventBus.getDefault().cancelEventDelivery(event);
}
```

Q&A