EventBus使用详解

Lauren_Liuling 关注

0.928 2016.01.09 14:33:34 字数 2,351 阅读 44,156

> 前言：EventBus出来已经有一段时间了，github上面也有很多开源项目中使用了EventBus。所以抽空学习顺便整理了一下。目前EventBus最新版本是3.0，所以本文是基于EventBus3.0的。

> 相关文章
>
> EventBus使用详解
>
> EventBus源码解析

概述

EventBus是针一款对Android的发布/订阅事件总线。它可以让我们很轻松的实现在Android各个组件之间传递消息，并且代码的可读性更好，耦合度更低。

如何使用

(1)首先需要定义一个消息类，该类可以不继承任何基类也不需要实现任何接口。如：

```
public class MessageEvent {
    ......
}
```

(2)在需要订阅事件的地方注册事件

```
EventBus.getDefault().register(this);
```

(3)产生事件，即发送消息

```
EventBus.getDefault().post(messageEvent);
```

(4)处理消息

```
@Subscribe(threadMode = ThreadMode.PostThread)
public void XXX(MessageEvent messageEvent) {
    ...
}
```

而在3.0之后，消息处理的方法可以随便取名，但是需要添加一个注解@Subscribe，并且要指定线程模型（默认为PostThread），四种线程模型，下面会讲到。
注意，事件处理函数的访问权限必须为public，否则会报异常。

(5)取消消息订阅

```
EventBus.getDefault().unregister(this);
```

有何优点

采用消息发布/订阅的一个很大的优点就是代码的简洁性，并且能够有效地降低消息发布者和订阅者之间的耦合度。
举个例子，比如有两个界面，ActivityA和ActivityB，从ActivityA界面跳转到ActivityB界面后，ActivityB要给ActivityA发送一个消息，ActivityA收到消息后在界面上显示出来。我们最先想到的方法就是使用广播，使用广播实现此需求的代码如下：
首先需要在ActivityA中定义一个广播接收器：

```
public class MessageBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        mMessageView.setText("Message from SecondActivity:" + intent.getStringExtra("mes
    }
}
```

还需要在onCreate()方法中注册广播接收器：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    //注册事件
    EventBus.getDefault().register(this);
    //注册广播
    IntentFilter intentFilter = new IntentFilter("message_broadcast");
    mBroadcastReceiver = new MessageBroadcastReceiver();
    registerReceiver(mBroadcastReceiver, intentFilter);
    ......
}
```

然后在onDestory()方法中取消注册广播接收器：

```
@Override
protected void onDestroy() {
    super.onDestroy();
    ......
    //取消广播注册
    unregisterReceiver(mBroadcastReceiver);
}
```

最后我们需要在ActivityB界面中发送广播消息：

```
findViewById(R.id.send_broadcast).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String message = mMessageET.getText().toString();
```

```
        if(TextUtils.isEmpty(message)) {
            message = "defaule message";
        }
        Intent intent = new Intent();
        intent.setAction("message_broadcast");
        intent.putExtra("message", message);
        sendBroadcast(intent);
    }
});
```

看着上面的实现代码，感觉也没什么不妥，挺好的！下面对比看下使用EventBus如何实现。

根据文章最前面所讲的EventBus使用步骤，首先我们需要定义一个消息事件类：

```
public class MessageEvent {

    private String message;

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

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

在ActivityA界面中我们首先需要注册订阅事件：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    //注册事件
    EventBus.getDefault().register(this);
    ......
}
```

然后在onDestory()方法中取消订阅：

```
@Override
protected void onDestroy() {
    super.onDestroy();
    //取消事件注册
    EventBus.getDefault().unregister(this);
}
```

当然还要定义一个消息处理的方法：

```
@Subscribe(threadMode = ThreadMode.MainThread)
public void onShowMessageEvent(MessageEvent messageEvent) {
    mMessageView.setText("Message from SecondActivity:" + messageEvent.getMessage());
}
```

至此，消息订阅者我们已经定义好了，我们还需要在ActivityB中发布消息：

```
findViewById(R.id.send).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
```

```
        String message = mMessageET.getText().toString();
        if(TextUtils.isEmpty(message)) {
            message = "defaule message";
        }
        EventBus.getDefault().post(new MessageEvent(message));
    }
});
```

对比代码一看，有人会说了，这尼玛有什么区别嘛！说好的简洁呢？哥们，别着急嘛！我这里只是举了个简单的例子，仅仅从该例子来看，EventBus的优势没有体现出来。现在我将需求稍微改一下，ActivityA收到消息后，需要从网络服务器获取数据并将数据展示出来。如果使用广播，ActivityA中广播接收器代码应该这么写：

```
public class MessageBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                //从服务器上获取数据
                ......
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        //将获取的数据展示在界面上
                        ......
                    }
                });
            }
        }).start();
    }
}
```

看到这段代码，不知道你何感想，反正我是看着很不爽，嵌套层次太多，完全违反了Clean Code的原则。那使用EventBus来实现又是什么样呢？我们看一下。

```
@Subscribe(threadMode = ThreadMode.BackgroundThread)
public void onGetDataEvent(MessageEvent messageEvent) {
    //从服务器上获取数据
    ......
    EventBus.getDefault().post(new ShowMessageEvent());
}

@Subscribe(threadMode = ThreadMode.MainThread)
public void onShowDataEvent(ShowMessageEvent showMessageEvent) {
    //将获取的数据展示在界面上
    ......
}

```

对比一下以上两段代码就能很明显的感觉到EventBus的优势，代码简洁、层次清晰，大大提高了代码的可读性和可维护性。我这只是简单的加了一个小需求而已，随着业务越来越复杂，使用EventBus的优势愈加明显。

常用API介绍

线程模型

在EventBus的事件处理函数中需要指定线程模型，即指定事件处理函数运行所在的想线程。在

上面我们已经接触到了EventBus的四种线程模型。那他们有什么区别呢?
在EventBus中的观察者通常有四种线程模型，分别是PostThread（默认）、MainThread、BackgroundThread与Async。

- PostThread：如果使用事件处理函数指定了线程模型为PostThread，那么该事件在哪个线程发布出来的，事件处理函数就会在这个线程中运行，也就是说发布事件和接收事件在同一个线程。在线程模型为PostThread的事件处理函数中尽量避免执行耗时操作，因为它会阻塞事件的传递，甚至有可能会引起ANR。
- MainThread：如果使用事件处理函数指定了线程模型为MainThread，那么不论事件是在哪个线程中发布出来的，该事件处理函数都会在UI线程中执行。该方法可以用来更新UI，但是不能处理耗时操作。
- BackgroundThread：如果使用事件处理函数指定了线程模型为BackgroundThread，那么如果事件是在UI线程中发布出来的，那么该事件处理函数就会在新的线程中运行，如果事件本来就是子线程中发布出来的，那么该事件处理函数直接在发布事件的线程中执行。在此事件处理函数中禁止进行UI更新操作。
- Async：如果使用事件处理函数指定了线程模型为Async，那么无论事件在哪个线程发布，该事件处理函数都会在新建的子线程中执行。同样，此事件处理函数中禁止进行UI更新操作。

为了验证以上四个方法，我写了个小例子。

```
@Subscribe(threadMode = ThreadMode.PostThread)
public void onMessageEventPostThread(MessageEvent messageEvent) {
    Log.e("PostThread", Thread.currentThread().getName());
}

@Subscribe(threadMode = ThreadMode.MainThread)
public void onMessageEventMainThread(MessageEvent messageEvent) {
    Log.e("MainThread", Thread.currentThread().getName());
}

@Subscribe(threadMode = ThreadMode.BackgroundThread)
public void onMessageEventBackgroundThread(MessageEvent messageEvent) {
    Log.e("BackgroundThread", Thread.currentThread().getName());
}

@Subscribe(threadMode = ThreadMode.Async)
public void onMessageEventAsync(MessageEvent messageEvent) {
    Log.e("Async", Thread.currentThread().getName());
}
```

分别使用上面四个方法订阅同一事件，打印他们运行所在的线程。首先我们在UI线程中发布一条MessageEvent的消息，看下日志打印结果是什么。

```
findViewById(R.id.send).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Log.e("postEvent", Thread.currentThread().getName());
            EventBus.getDefault().post(new MessageEvent());
        }
    });
```

打印结果如下:

```
2689-2689/com.lling.eventbusdemo E/postEvent： main
2689-2689/com.lling.eventbusdemo E/PostThread： main
2689-3064/com.lling.eventbusdemo E/Async： pool-1-thread-1
```

```
2689-2689/com.lling.eventbusdemo E/MainThread： main
2689-3065/com.lling.eventbusdemo E/BackgroundThread： pool-1-thread-2
```

从日志打印结果可以看出，如果在UI线程中发布事件，则线程模型为PostThread的事件处理函数也执行在UI线程，与发布事件的线程一致。线程模型为Async的事件处理函数执行在名字叫做pool-1-thread-1的新的线程中。而MainThread的事件处理函数执行在UI线程，BackgroundThread的时间处理函数执行在名字叫做pool-1-thread-2的新的线程中。

我们再看看在子线程中发布一条MessageEvent的消息时，会有什么样的结果。

```
findViewById(R.id.send).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    Log.e("postEvent", Thread.currentThread().getName());
                    EventBus.getDefault().post(new MessageEvent());
                }
            }).start();
        }
    });
```

打印结果如下：

```
3468-3945/com.lling.eventbusdemo E/postEvent： Thread-125
3468-3945/com.lling.eventbusdemo E/PostThread： Thread-125
3468-3945/com.lling.eventbusdemo E/BackgroundThread： Thread-125
3468-3946/com.lling.eventbusdemo E/Async： pool-1-thread-1
3468-3468/com.lling.eventbusdemo E/MainThread： main
```

从日志打印结果可以看出，如果在子线程中发布事件，则线程模型为PostThread的事件处理函数也执行在子线程，与发布事件的线程一致（都是Thread-125）。BackgroundThread事件模型也与发布事件在同一线程执行。Async则在一个名叫pool-1-thread-1的新线程中执行。MainThread还是在UI线程中执行。

上面一个例子充分验证了指定不同线程模型的事件处理方法执行所在的线程。

黏性事件

除了上面讲的普通事件外，EventBus还支持发送黏性事件。何为黏性事件呢？简单讲，就是在发送事件之后再订阅该事件也能收到该事件，跟黏性广播类似。具体用法如下：

订阅黏性事件：

```
EventBus.getDefault().register(StickyModeActivity.this);
```

黏性事件处理函数：

```
@Subscribe(sticky = true)
public void XXX(MessageEvent messageEvent) {
    ......
}
```

发送黏性事件：

```
EventBus.getDefault().postSticky(new MessageEvent("test"));
```

处理消息事件以及取消订阅和上面方式相同。

看个简单的黏性事件的例子，为了简单起见我这里就在一个Activity里演示了。

Activity代码：

```
public class StickyModeActivity extends AppCompatActivity {

    int index = 0;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_sticky_mode);
        findViewById(R.id.post).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                EventBus.getDefault().postSticky(new MessageEvent("test" + index++));
            }
        });
        findViewById(R.id.regist).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                EventBus.getDefault().registerSticky(StickyModeActivity.this);
            }
        });

        findViewById(R.id.unregist).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                EventBus.getDefault().unregister(StickyModeActivity.this);
            }
        });

    }

    @Subscribe(threadMode = ThreadMode.PostThread, sticky = true)
    public void onMessageEventPostThread(MessageEvent messageEvent) {
        Log.e("PostThread", messageEvent.getMessage());
    }

    @Subscribe(threadMode = ThreadMode.MainThread, sticky = true)
    public void onMessageEventMainThread(MessageEvent messageEvent) {
        Log.e("MainThread", messageEvent.getMessage());
    }

    @Subscribe(threadMode = ThreadMode.BackgroundThread, sticky = true)
    public void onMessageEventBackgroundThread(MessageEvent messageEvent) {
        Log.e("BackgroundThread", messageEvent.getMessage());
    }

    @Subscribe(threadMode = ThreadMode.Async, sticky = true)
    public void onMessageEventAsync(MessageEvent messageEvent) {
        Log.e("Async", messageEvent.getMessage());
    }

}
```

布局代码activity_sticky_mode.xml：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:paddingLeft="@dimen/activity_horizontal_
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
```

```
    android:orientation="vertical"
    tools:context="com.lling.eventbusdemo.StickyModeActivity">

    <Button
        android:id="@+id/post"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Post"/>

    <Button
        android:id="@+id/regist"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Regist"/>

    <Button
        android:id="@+id/unregist"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="UnRegist"/>

</LinearLayout>
```

代码很简单，界面上三个按钮，一个用来发送黏性事件，一个用来订阅事件，还有一个用来取消订阅的。首先在未订阅的情况下点击发送按钮发送一个黏性事件，然后点击订阅，会看到日志打印结果如下：

```
15246-15246/com.lling.eventbusdemo E/PostThread： test0
15246-15391/com.lling.eventbusdemo E/Async： test0
15246-15246/com.lling.eventbusdemo E/MainThread： test0
15246-15393/com.lling.eventbusdemo E/BackgroundThread： test0
```

这就是粘性事件，能够收到订阅之前发送的消息。但是它只能收到最新的一次消息，比如说在未订阅之前已经发送了多条黏性消息了，然后再订阅只能收到最近的一条消息。这个我们可以验证一下，我们连续点击5次POST按钮发送5条黏性事件，然后再点击REGIST按钮订阅，打印结果如下：

```
6980-6980/com.lling.eventbusdemo E/PostThread： test4
6980-6980/com.lling.eventbusdemo E/MainThread： test4
6980-7049/com.lling.eventbusdemo E/Async： test4
6980-7048/com.lling.eventbusdemo E/BackgroundThread： test4
```

由打印结果可以看出，确实是只收到最近的一条黏性事件。

好了，EventBus的使用暂时分析到这里，例子代码从这里获取。下一讲将讲解EventBus的源码。

本文首发：http://liuling123.com/2016/01/EventBus-explain.html

Lauren_Liuling Android开发工程师
总资产2 (约0.21元) 共写了5698字 获得211个赞 共114个粉丝

关注

被以下专题收入，发现更多相似内容

Android... Android知识 Android... 技术与人生 技术

Android... android安全 展开更多 ∨

推荐阅读

更多精彩内容 >

EventBus使用详解

前言：EventBus出来已经有一段时间了，github上面也有很多开源项目中使用了EventBus。所以抽空学习...

hi小波 阅读 705 评论 1 赞 2

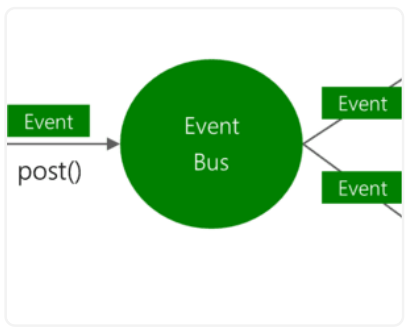
EventBus的使用之重复早轮子

概述 EventBus是Android和Java的发布/订阅事件总线。 简化了组件之间的通信；将事件发送者和接收者...

廉颇未老 阅读 81 评论 0 赞 0

EventBus使用详解

目录 1.概述 2.实战 1.基本框架搭建 2.新建一个类FirstEvent 3.在要接收消息的页面注册Even...



慕涵盛华 阅读 4,235 评论 2 赞 13

EventBus使用详解

EventBus这个开源框架出来已经很久了，深的很多开发者青睐，由greenrobot组织贡献(该组织还贡献了gr...

Scus 阅读 410 评论 0 赞 0

EventBus使用详解

前言 最近在公司做一个类似于手机工厂模式的一个项目，用来检测其他各个App是否正常工作，所以要求是尽可能的轻量级，...

Luckily_Liu 阅读 501 评论 2 赞 7