**Введение**

BroadcastReceiver — это компонент Android, который позволяет приложениям реагировать на широковещательные сообщения (broadcasts). Эти сообщения могут быть как системными (например, уведомления о состоянии батареи, сети или звонков), так и пользовательскими (созданными другими приложениями). BroadcastReceiver играет ключевую роль в архитектуре Android, позволяя приложениям быть более интерактивными и отзывчивыми, не расходуя ресурсы устройства на постоянную работу в фоновом режиме.

**История появления BroadcastReceiver**

До появления BroadcastReceiver в Android разработчики сталкивались с проблемой эффективного взаимодействия приложений с системными событиями. Ранее для подобных задач использовались сервисы, которые постоянно работали в фоне, что приводило к избыточному расходу ресурсов устройства. BroadcastReceiver стал решением этой проблемы, предоставив механизм, который позволяет приложениям "просыпаться" только при наступлении определённых событий. Это значительно улучшило управление энергопотреблением и производительность системы.

**Основные функции и возможности BroadcastReceiver**

BroadcastReceiver позволяет приложениям:

* Реагировать на системные события, такие как:
* Изменение состояния сети (Wi-Fi, мобильный интернет).
* Уведомления о входящих звонках или сообщениях.
* События, связанные с батареей (например, низкий заряд).
* Завершение загрузки устройства.
* Обрабатывать пользовательские события, отправленные через метод sendBroadcast().

BroadcastReceiver может быть зарегистрирован двумя способами:

1. **Статическая регистрация** в файле AndroidManifest.xml. Этот метод позволяет приёмнику работать даже тогда, когда приложение не запущено.
2. **Динамическая регистрация** в коде с использованием метода registerReceiver(). Этот способ используется для временного отслеживания событий, когда приложение активно.

**Пример статической регистрации:**

<receiver android:name=".CallReceiver">

<intent-filter>

<action android:name="android.intent.action.PHONE\_STATE" />

</intent-filter>

</receiver>

**Пример динамической регистрации:**

BroadcastReceiver receiver = new CallReceiver();

IntentFilter filter = new IntentFilter(TelephonyManager.ACTION\_PHONE\_STATE\_CHANGED);

context.registerReceiver(receiver, filter);

**Архитектура приложения и роль BroadcastReceiver**

Приложение состоит из нескольких ключевых компонентов:

* **CallReceiver** — класс, реализующий BroadcastReceiver, который отслеживает состояние телефонных звонков.
* **AccessibilityServiceKillSwitch** — сервис доступности, который реагирует на состояние звонков и выполняет действия в зависимости от текущего приложения на экране.
* **KillSwitchService** — сервис, запускающий AccessibilityServiceKillSwitch.
* **MainActivity** — основная активность приложения, предоставляющая интерфейс для управления функциями приложения.

BroadcastReceiver в данном случае играет роль связующего звена между системными событиями (изменение состояния звонков) и логикой приложения. Он обеспечивает передачу информации о текущем состоянии звонков в другие компоненты через SharedPreferences.

**Использование BroadcastReceiver в приложении KillSwitch**

В приложении KillSwitch BroadcastReceiver реализован в классе CallReceiver. Его основная задача — отслеживать состояние телефонных звонков и передавать эту информацию другим компонентам приложения. Рассмотрим реализацию более подробно.

package com.example.killswitch;

import android.content.BroadcastReceiver;

import android.content.Context;

import android.content.Intent;

import android.content.SharedPreferences;

import android.telephony.TelephonyManager;

import android.util.Log;

public class CallReceiver extends BroadcastReceiver {

private static final String TAG = "CallReceiver";

private static final String PREFS\_NAME = "KillSwitchPrefs";

private static final String PREF\_CALL\_ACTIVE = "call\_active";

@Override

public void onReceive(Context context, Intent intent) {

String state = intent.getStringExtra(TelephonyManager.EXTRA\_STATE);

Log.d(TAG, "Phone state changed: " + state);

SharedPreferences prefs = context.getSharedPreferences(PREFS\_NAME, Context.MODE\_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();

if (TelephonyManager.EXTRA\_STATE\_RINGING.equals(state) || TelephonyManager.EXTRA\_STATE\_OFFHOOK.equals(state)) {

Log.d(TAG, "Call active");

editor.putBoolean(PREF\_CALL\_ACTIVE, true);

} else if (TelephonyManager.EXTRA\_STATE\_IDLE.equals(state)) {

Log.d(TAG, "Call ended");

editor.putBoolean(PREF\_CALL\_ACTIVE, false);

}

editor.apply();

}

}

Описание работы:

1. При получении события (например, изменения состояния телефона) метод onReceive() извлекает информацию о текущем состоянии звонка из переданного Intent.
2. Состояние звонка (активен, завершён или входящий вызов) сохраняется в SharedPreferences для последующего использования другими компонентами приложения.
3. Логирование используется для отладки и мониторинга работы приёмника.

**Взаимодействие с AccessibilityServiceKillSwitch**

BroadcastReceiver передаёт информацию о состоянии звонков в AccessibilityServiceKillSwitch, который отвечает за закрытие браузеров, если звонок активен.

@Override

public void onAccessibilityEvent(AccessibilityEvent event) {

if (event.getEventType() == AccessibilityEvent.TYPE\_WINDOW\_STATE\_CHANGED) {

CharSequence packageName = event.getPackageName();

Log.d(TAG, "Foreground app: " + packageName);

SharedPreferences prefs = getSharedPreferences(PREFS\_NAME, MODE\_PRIVATE);

boolean isCallActive = prefs.getBoolean(PREF\_CALL\_ACTIVE, false);

Log.d(TAG, "Is call active: " + isCallActive);

if (isCallActive && isBrowserApp(packageName)) {

Log.d(TAG, "Attempting to close browser: " + packageName);

boolean backAction1 = performGlobalAction(GLOBAL\_ACTION\_BACK);

boolean backAction2 = performGlobalAction(GLOBAL\_ACTION\_BACK);

boolean backAction3 = performGlobalAction(GLOBAL\_ACTION\_BACK);

Log.d(TAG, "Back action results: " + backAction1 + ", " + backAction2 + ", " + backAction3);

}

}

}

Описание работы:

1. Сервис доступности отслеживает изменения состояния окон (например, открытие браузера).
2. Если звонок активен (информация берётся из SharedPreferences), сервис выполняет три действия "Назад", чтобы закрыть браузер.

**Преимущества использования BroadcastReceiver в данном приложении**

1. **Эффективное использование ресурсов**:  
   BroadcastReceiver активируется только при наступлении события (в данном случае — изменения состояния звонков), что позволяет избежать постоянной работы в фоне и экономить ресурсы устройства.
2. **Простота интеграции**:  
   Использование SharedPreferences для хранения состояния звонков позволяет другим компонентам приложения легко получать доступ к этой информации без необходимости прямого взаимодействия с BroadcastReceiver.
3. **Универсальность**:  
   BroadcastReceiver может быть легко модифицирован для обработки других системных событий, таких как изменения состояния сети или уровня заряда батареи.

**Ограничения BroadcastReceiver в данном приложении**

Несмотря на свои преимущества, использование BroadcastReceiver имеет некоторые ограничения:

* **Ограниченное время выполнения**: Метод onReceive() должен завершиться в течение 10 секунд, иначе система может завершить выполнение BroadcastReceiver.
* **Ограничения на статическую регистрацию**: В последних версиях Android (начиная с Android 8.0) статическая регистрация BroadcastReceiver ограничена для некоторых типов событий, что может потребовать дополнительных усилий для реализации.

**Альтернативы BroadcastReceiver**

В зависимости от требований приложения, вместо BroadcastReceiver могут быть использованы другие подходы для обработки системных событий. Основные альтернативы:

1. **JobScheduler**
   * **Описание**: API для выполнения фоновых задач, которые должны быть выполнены при определённых условиях (например, подключение к Wi-Fi или зарядное устройство).
   * **Преимущества**:
     + Подходит для длительных операций.
     + Может быть использован для задач, которые не требуют немедленного выполнения.
   * **Ограничения**:
     + Не подходит для задач, требующих немедленной реакции на события.
2. **WorkManager**
   * **Описание**: Современный инструмент для управления фоновой работой, который гарантирует выполнение задач даже при перезагрузке устройства.
   * **Преимущества**:
     + Поддерживает сложные условия выполнения задач.
     + Может работать в связке с другими API, такими как JobScheduler.
   * **Ограничения**:
     + Может быть избыточным для простых задач, таких как обработка системных событий.
3. **Foreground Service**
   * **Описание**: Сервис, который работает в фоне и отображает уведомление для пользователя.
   * **Преимущества**:
     + Подходит для длительных операций, требующих уведомления пользователя (например, воспроизведение музыки или запись звонков).
   * **Ограничения**:
     + Требует отображения постоянного уведомления, что может быть нежелательно для некоторых приложений.
4. **ContentObserver**
   * **Описание**: Используется для отслеживания изменений в данных, таких как изменения в базе данных или файлах.
   * **Преимущества**:
     + Подходит для задач, связанных с мониторингом данных.
   * **Ограничения**:
     + Не может быть использован для обработки системных событий, таких как изменения состояния звонков.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Критерий** | **BroadcastReceiver** | **JobScheduler** | **WorkManager** | **Foreground Service** | **ContentObserver** |
| Реакция на события | Немедленная | Отложенная | Отложенная | Немедленная | Зависит от данных |
| Энергопотребление | Низкое | Среднее | Среднее | Высокое | Низкое |
| Поддержка сложных условий | Ограниченная | Высокая | Высокая | Низкая | Низкая |
| Простота реализации | Высокая | Средняя | Средняя | Средняя | Средняя |