flutter–EventBus全局事件总线

season_26

EventBus通常用于解决跨页面或跨组件通信的需求，使得组件之间的耦合度更低，代码更易于维护和扩展。

功能：

1. **发布/订阅模式**：EventBus基于发布/订阅模式，组件可以订阅感兴趣的事件，当事件发生时，订阅者会收到通知并执行相应操作。

2. **跨组件通信**：EventBus可以实现跨页面、跨组件的通信，使得不同组件之间能够进行数据传递和交互。

用法：

1. **创建EventBus实例**：在应用程序中创建一个全局的EventBus实例，通常可以使用第三方库如event_bus来简化创建过程。

2. **定义事件类**：定义不同类型的事件类，通常是继承自一个基类，用于标识不同的事件类型。

3. **订阅事件**：在需要接收事件通知的组件中，通过EventBus实例的on方法订阅感兴趣的事件。

4. **发布事件**：在某个组件中发生事件时，通过EventBus

实例的fire方法发布该事件，触发订阅者的回调函数执行。

示例代码如下：

```
// 创建全局EventBus实例
EventBus eventBus = EventBus();
// 定义事件类
class MyEvent {
  String message;
  MyEvent(this.message);
}
// 订阅事件
eventBus.on<MyEvent>().listen((event) {
  print('Received event: ${event.message}');
});
// 发布事件
eventBus.fire(MyEvent('Hello EventBus!'));
```

知识补充：

单例模式

单例模式是一种设计模式，用于确保一个类只有一个实例，并提供全局访问点来访问该实例。单例模式通常用于需要在整个应用程序中共享相同实例的情况，以避免多个实例的创建和资源浪费。

特点：

1. **私有构造函数**：单例类的构造函数是私有的，防止外部

直接实例化该类。

2. **静态实例**：单例类内部会保存一个静态实例变量，用于存储唯一的实例。
3. **全局访问点**：提供一个静态方法来获取该实例，以便在整个应用程序中访问单例实例。

实现方式：

在Dart中，可以使用静态变量和工厂构造函数来实现单例模式，确保类只有一个实例，并通过静态方法来获取该实例。

1. **饿汉式**：在类加载时就创建单例实例，线程安全，但可能会造成资源浪费。

```
class Singleton {
  static final Singleton _instance = Singleton._();
  factory Singleton() => _instance;
  Singleton._();
}
```

1. 代码定义了一个静态、不可变的 `_instance` 变量，类型为 `Singleton`，并将其初始化为调用`Singleton._()` 构造函数创建的实例。使用了 Dart 中的静态成员和不可变变量的声明方式，确保`_instance` 只被初始化一次.
2. 通过工厂构造函数来获取实例，可以确保每次调用 `Singleton()` 都返回同一个实例，实现单例模式的要求。
3. 代码定义了一个私有的构造函数`Singleton._()`，用于创建 `Singleton` 类的实例。

为什么变量的类型可以为Singleton

在单例模式中，我们希望通过静态变量 `_instance` 来保存单

例实例，因此将 `_instance` 的类型设置为 `Singleton` 可以确保 `_instance` 只能存储 `Singleton` 类的实例，从而实现单例模式的要求。

在 Dart 中，类也是一种类型，因此可以将类作为变量的类型。这样做的好处是可以限制变量只能存储特定类型的实例，从而提高代码的类型安全性。

2. **懒汉式**：在首次获取实例时创建单例实例，延迟加载，线程不安全，需要考虑线程安全性。

```
class Singleton {
  static Singleton? _instance;

  factory Singleton() {
    if (_instance == null) {
      _instance = Singleton._();
    }
    return _instance!;
  }

  Singleton._();
}
```

使用：

```
Singleton singleton1 = Singleton();
Singleton singleton2 = Singleton();
print(identical(singleton1, singleton2)); //
Output: true
```

3. 使用`shared`方法。

```
class Singleton {
  static Singleton _instance;

  Singleton._();

  static Singleton shared() {
    if (_instance == null) {
      _instance = Singleton._();
    }
    return _instance;
  }

  void printMessage() {
    print('Hello from Singleton instance');
  }
}

void main() {
  Singleton singleton1 = Singleton.shared();
  Singleton singleton2 = Singleton.shared();

  print(identical(singleton1, singleton2)); //
Output: true

  singleton1.printMessage(); // Output: Hello from
```

```
Singleton instance
}
```

优点：

- **全局访问**：方便在应用程序的任何地方访问单例实例。
- **资源共享**：避免重复创建实例，节省资源。
- **数据共享**：多个模块之间共享数据，实现数据共享和通信。

注意事项：

- **线程安全**：在多线程环境下需要考虑线程安全性，避免多个线程同时创建实例。
- **内存泄漏**：需要注意单例实例的生命周期，避免出现内存泄漏问题。
- **滥用单例**：不应滥用单例模式，只有在确实需要全局唯一实例时才使用。

subscriptions字典

在 Dart 中，可以使用一个 `subscriptions` 字典来管理不同类型事件的订阅者，并且可以通过该字典来取消订阅。这种做法通常用于事件管理系统，其中不同类型的事件有不同的订阅者，并且需要能够灵活地取消订阅。

可以使用 `event_bus` 库来实现事件总线（Event Bus）功能，结合 `subscriptions` 字典来管理不同类型事件的订阅者。下面是一个示例，演示了如何结合 `subscriptions` 和 `event_bus` 使用：

首先，需要在 `pubspec.yaml` 文件中添加 `event_bus` 依赖：

```
dependencies:
  event_bus: ^5.0.1
```

然后可以创建一个 EventManager 类，结合 event_bus 和 subscriptions 字典来管理事件订阅者：

```
import 'package:event_bus/event_bus.dart';

class EventManager {
  EventBus eventBus = EventBus();
  Map<String, StreamSubscription> subscriptions =
{};

  void subscribe(String eventType, void
Function(dynamic) callback) {
    if (!subscriptions.containsKey(eventType)) {
      subscriptions[eventType] =
eventBus.on(eventType).listen(callback);
    }
  }

  void publish(String eventType, dynamic data) {
    eventBus.fire(eventType, data);
  }

  void unsubscribe(String eventType) {
    if (subscriptions.containsKey(eventType)) {
      subscriptions[eventType].cancel();
```

```
      subscriptions.remove(eventType);
    }
  }
}

void main() {
  EventManager eventManager = EventManager();

  eventManager.subscribe('login', (data) {
    print('User logged in: $data');
  });

  eventManager.publish('login', 'username');

  eventManager.unsubscribe('login');

  eventManager.publish('login', 'another
username'); // 这个事件将不会触发订阅者的回调函数
}
```

在上面的示例中，我们创建了一个`EventManager`类，其中包含了一个`eventBus`对象用于处理事件总线功能，以及一个 ``subscriptions```字典用于管理事件订阅者的流订阅。

在`subscribe`方法中，我们使用`eventBus.on(eventType).listen(callback)`来订阅特定类型的事件，并将回调函数注册到事件总线上。在`publish`

方法中，我们使用 `eventBus.fire(eventType, data)` 来发布特定类型的事件，并传递数据给订阅者。在 `unsubscribe` 方法中，我们取消特定类型事件的订阅。

上难度

```
/// EventBus的工具类
class WisdomEventBusUtils {
  // 单列模式
  static EventBus? _eventBus;

  static EventBus? shared() {
    _eventBus ??= EventBus();
    return _eventBus;
  }

  /// 订阅者
  static Map<Type, List<StreamSubscription?>?>
subscriptions = {};

  /// 添加监听事件
  /// [T] 事件泛型 必须要传
  /// [onData] 接受到事件
  /// [autoManaged] 自动管理实例，off 取消
  static StreamSubscription? on<T extends
Object>(void Function(T event) onData,
```

```
      {Function? onError,
      void Function()? onDone,
      bool? cancelOnError,
      bool autoManaged = true}) {
    StreamSubscription? subscription =
shared()?.on<T>().listen(onData,
        onError: onError, onDone: onDone,
cancelOnError: cancelOnError);
    if (autoManaged == true) {
      subscriptions ??= {};
      List<StreamSubscription?> subs =
subscriptions[T.runtimeType] ?? [];
      subs.add(subscription);
      subscriptions[T.runtimeType] = subs;
    }
    return subscription;
  }

  /// 移除监听者
  /// [T] 事件泛型 必须要传
  /// [subscription] 指定
  static void off<T extends
Object>({StreamSubscription? subscription}) {
    subscriptions = {};
    if (subscription != null) {
      // 移除传入的
```

```
      List<StreamSubscription?> subs =
subscriptions[T.runtimeType] ?? [];
      subs.remove(subscription);
      subscriptions[T.runtimeType] = subs;
    } else {
      // 移除全部
      subscriptions[T.runtimeType] = null;
    }
  }

  /// 发送事件
  static void fire(event) {
    shared()?.fire(event);
  }
}

/// EventBus的工具类
/// 有状态组件
mixin WisdomEventBusMixin<T extends StatefulWidget>
on State<T> {
  /// 需要定义成全局的,共用一个是实例
  EventBus? mEventBus =
WisdomEventBusUtils.shared();

  /// 订阅者
  List<StreamSubscription?> mEventBusSubscriptions
```

```
= [];

  /// 统一在这里添加监听者
  @protected
  void mAddEventBusListeners();

  /// 添加监听事件
  void mAddEventBusListener<T>(void Function(T
event) onData,
      {Function? onError, void Function()? onDone,
bool? cancelOnError}) {

mEventBusSubscriptions.add(mEventBus?.on<T>().liste
n(onData,
        onError: onError, onDone: onDone,
cancelOnError: cancelOnError));
  }

  /// 发送事件
  void mEventBusFire(event) {
    mEventBus?.fire(event);
  }

  @override
  @mustCallSuper
  void dispose() {
```

```
    super.dispose();
    debugPrint('dispose:WisdomEventBusMixin');
    if (mEventBusSubscriptions.isNotEmpty)
      for (StreamSubscription? subscription in
mEventBusSubscriptions) {
        subscription!.cancel();
      }
  }

  @override
  @mustCallSuper
  void initState() {
    super.initState();
    debugPrint('initState:WisdomEventBusMixin');
    mAddEventBusListeners();
  }
}
```

尾记：EventBus与Provider的区别

`EventBus` 和 `Provider` 是两种在 Flutter 中常用的状态管理工具，它们有不同的特点和适用场景：

1. **EventBus**：
    - **特点**：`EventBus` 是一个简单的事件总线库，用于在应用程序中进行事件的发布和订阅。它允许不同部分的代码之间进行解耦，通过事件的方式进行通信。
    - **适用场景**：适用于在应用程序中有多个模块或组件

之间需要进行通信的场景。比如在不同页面之间传递数据、触发特定事件等情况下使用 `EventBus` 可以方便实现。

2. **Provider**：
    - **特点**：`Provider` 是一个用于管理应用程序状态的工具，它提供了一种简单而强大的方式来共享和管理状态。通过 `Provider`，可以在应用程序中轻松地实现状态共享和更新。
    - **适用场景**：适用于需要在应用程序中共享状态、管理全局数据、以及实现状态变更通知的场景。比如在应用程序中共享用户登录状态、主题设置、购物车数据等信息时可以使用 `Provider`。

在什么应用场景下使用：

- 使用 `EventBus` 的场景：
    - 当需要在应用程序中实现不同模块或组件之间的解耦通信时，可以使用 `EventBus`。比如在一个复杂的应用程序中，不同模块需要相互通信但又不希望直接耦合在一起时，可以使用 `EventBus` 来进行事件的发布和订阅。
    - 当需要在应用程序中进行跨页面的数据传递或事件触发时，可以使用 `EventBus` 来简化通信过程。
- 使用 `Provider` 的场景：
    - 当需要在应用程序中共享和管理全局状态时，可以使用 `Provider`。比如在需要共享用户登录状态、

主题设置、语言设置等全局数据时，可以使用 `Provider` 来管理这些状态。

- 当需要在应用程序中实现状态变更通知、自动更新界面等功能时，可以使用 `Provider` 来管理状态，并通过 `Consumer` 或 `Provider.of` 来监听状态变化并更新界面。

总的来说，`EventBus` 适用于实现组件之间的解耦通信，而 `Provider` 适用于共享和管理全局状态。根据具体的应用场景和需求，可以选择合适的工具来实现状态管理和通信功能。

什么是解耦通信？什么叫直接耦合在一起？

解耦通信是指在软件设计中，将不同模块或组件之间的通信机制设计得相对独立，使得它们之间的关联性较低，以减少模块之间的依赖关系，提高系统的灵活性和可维护性。解耦通信可以通过事件驱动、消息队列、中介者模式等方式实现，从而使得系统中的各个部分能够独立开发、测试和维护。

直接耦合在一起则是指软件系统中的不同模块或组件之间的关系较为紧密，彼此之间直接依赖，一旦其中一个模块发生变化，可能会影响到其他模块的功能或实现。直接耦合在一起的设计会导致系统的可维护性较差，难以扩展和修改。

举例来说，假设一个应用程序中有两个模块 A 和 B，模块 A 负责用户登录，模块 B 负责显示用户信息。如果模块 A 和模块 B 之间直接耦合在一起，即模块 A 直接调用模块 B 的方法来显示用户信息，那么一旦模块 B 的显示逻辑发生变化，就

需要修改模块 A 的代码，这样就会导致模块 A 和模块 B 之间的紧密耦合。
相反，如果使用解耦通信的方式，比如通过事件总线或消息队列，模块 A 发布一个用户登录事件，模块 B 订阅该事件并显示用户信息，那么模块 A 和模块 B 之间就解耦了，彼此之间不直接依赖，修改模块 B 的显示逻辑不会影响到模块 A，系统的灵活性和可维护性会得到提升。