Flutter 中单例，shared_preferences 实现数据持久化，`IO 文件读写操作数据持久化

一巴掌拍出两坨脂肪

本章知识点主要介绍 Flutter 中的数据共享和持久化，很显然，本章知识点相当重要，数据共享和持久化操作基本是每个应用的必备操作。比如：一个应用里有很多界面都会用到我的的用户信息，而我的用户信息一开始就在登录时就返回了，我们把用户的基本信息存储在一个全局的变量里即可实现数据共享，让其他视图直接引用该变量值即可 。但是这样的存储只能达到内存级别的，应用关闭后这些数据就被丢失了。要实现关闭应用后再次重启应用我的用户信息而不丢失那该怎么办呢？ 这就是本章节的重点了------ 持久化

本章简要：

1、`Flutter 中单例介绍`

2、`shared_preferences 实现数据持久化`

3、`IO 文件读写操作数据持久化`

一、全局静态变量

Flutter 中的 Dart 语言如同 Java 语言一样，存在很多地方的相似性。在通过使用 `static` 关键字 为一个变量开辟内存地址

后，而其他需要引用到该变量值时， 只需要将内存地址指向该变量即可实现数据共享。

如： 我定义了一个 Global 类来管理全局的的静态资源：

```
class Global{

  //定义一个静态属性 name
  static String name = "张三";

}
```

在需要使用到该变量值的地方 通过 `Global.name` 方式获取该变量的值。

二、单例

单例模式是日常开发中最常用的设计模式之一。Dart 是单线程模型，因此在 Flutter 中实现单例 不需要像 Java 中加双重检查锁去考虑多线程的问题。

Dart 中的单例也有两种模式：

1、饿汉式

饿汉式比较好理解：在类加载时，就进行实例的创建。加载时获取实例速度较慢，运行时速度较快。通俗的讲："我管你吃不吃包子，反正我先把这坨包子蒸好放在这儿"

实例代码：

```
class UserHelper{

    // 单例公开访问点
    factory UserHelper() =>_userInstance();
```

a1

工厂构造函数


```
  static UserHelper get instance => _userInstance();

  //静态私有成员，没有初始化 静态私有成员，加载时就进行初始化
  static UserHelper _instance = UserHelper._();

  // 私有构造函数
  UserHelper._() {
    // 具体初始化代码
    print("------>初始化");
  }

  // 静态、同步、私有访问点
  static UserHelper _userInstance() {

    return _instance;
  }

  String getUserName(){

    return "张三";
  }


}
```

a3 b1 b3 a4 b4 a2 b2

饿汉式单例测试：

```
void main(){

  var userHelper = UserHelper();
  var userHelper1 = UserHelper();
  var userHelper2 =  UserHelper.instance;

  print("------------> 对象: '${userHelper.hashCode} 相等
'+${identical(userHelper, userHelper1)}"); //true
```

a0 b0

```
    print("------------> 对象: '${userHelper1.hashCode} 相等
'+${identical(userHelper, userHelper2)}"); //true
    print("------------> 对象: '${userHelper2.hashCode} 相等
'+${identical(userHelper1, userHelper2)}"); //true
  }
```

控制台输出：

```
    I/flutter: ------>初始化
    I/flutter: ------------> 对象: '1070008279 相等  '+true
    I/flutter: ------------> 对象: '1070008279 相等  '+true
    I/flutter: ------------> 对象: '1070008279 相等  '+true
```

可以看出 UserHelper 类有且只初始化了一次，且 hashCode 值也都相等，证明后面的无论 new UserHelper() 多少次拿到的都是同一对象。

2、懒汉式

在类加载时，不创建实例。加载时速度较快，运行时获取实例速度较慢。通俗的讲："我管你吃不吃包子，等你问我了我才开始给你做包子"

实例代码：

```
class UserHelper{

  // 单例公开访问点
a1  factory UserHelper() =>_userInstance();

  static UserHelper get instance => _userInstance();          b1

  // 静态私有成员，没有初始化 静态私有成员，加载时不进行初始化
a3  static UserHelper _instance;                              b3
```

a4

```
  // 私有构造函数
  UserHelper._() {
    // 具体初始化代码
    print("------>初始化");
  }

  // 静态、同步、私有访问点
  static UserHelper _userInstance() {
    if (_instance == null) {
      _instance = UserHelper._();
    }
    return _instance;
  }


}
```

b4

a2

b2

懒汉式单例测试：

a0

```
void main(){

  var userHelper = UserHelper();
  var userHelper1 = UserHelper();
  var userHelper2 =  UserHelper.instance;

  print("------------> 对象: '${userHelper.hashCode} 相等
'+${identical(userHelper, userHelper1)}"); //true
  print("------------> 对象: '${userHelper1.hashCode} 相等
'+${identical(userHelper, userHelper2)}"); //true
  print("------------> 对象: '${userHelper2.hashCode} 相等
'+${identical(userHelper1, userHelper2)}"); //true

}
```

b0

控制台输出：

```
    I/flutter ( 8120): ------>初始化
    I/flutter ( 8120): ------------> 对象: '537698073 相等
'+true
    I/flutter ( 8120): ------------> 对象: '537698073 相等
'+true
    I/flutter ( 8120): ------------> 对象: '537698073 相等
'+true
```

无论是通过饿汉式 还是懒汉式方式实现的单例始终拿到的都是同一对象，而同一对象中的属性值肯定是相等的，那么上面介绍的第一点通过静态实现全局共享 也可以换成单例的方式实现。

下面列举下使用单例的场景和好处：

1、对象需要频繁的实例化和销毁，此时考虑使用单例可以大幅度提高性能。

2、控制资源的使用。

3、控制实例产生的数量，达到节约资源的目的。

4、作为通信媒介使用，也就是数据共享。

三、shared_preferences 数据持久化

shared_preferences 是 Flutter 提供的 key-value 存储插件，它通过 Android 和 iOS 平台提供的机制来实现数据持久化到磁盘中。在 iOS 上封装的是 NSUserDefault（后缀 `.plist` 的文件中），在 android 上封装的是 SharedPreferences（后缀 `.xml` 文件中）。在使用上也是如同原生一样简单。

为工程添加 shared_preferences 插件：

1、在`pubspec.yaml`文件中添加依赖

```
    dependencies:
      fluttertoast: ^3.0.3
      flutter:
        sdk: flutter
      #添加持久化插件 sp
      shared_preferences: ^0.5.3+1
```

在 pubspec.yaml 添加依赖时特别需要注意缩进，多一个或少一个空格可能都将添加不上。

2、安装依赖库

执行 `flutter packages get` 命令；AS 开发工具直接右上角 `packages get` 也可。

3、在使用的文件中导入该库

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

shared_preferences 源码分析：

```
    class SharedPreferences {
      SharedPreferences._(this._preferenceCache);

      static const String _prefix = 'flutter.';
      static SharedPreferences _instance;
      static Future<SharedPreferences> getInstance() async
{
        if (_instance == null) {
          final Map<String, Object> preferencesMap =
              await _getSharedPreferencesMap();
          _instance = SharedPreferences._(preferencesMap);
        }
```

```
        return _instance;
      }

      /// The cache that holds all preferences.
      ///
      /// It is instantiated to the current state of the
SharedPreferences or
      /// NSUserDefaults object and then kept in sync via
setter methods in this
      /// class.
      ///
      /// It is NOT guaranteed that this cache and the
device prefs will remain
      /// in sync since the setter method might fail for
any reason.
      final Map<String, Object> _preferenceCache;

      /// Returns all keys in the persistent storage.
      Set<String> getKeys() =>
Set<String>.from(_preferenceCache.keys);

      /// Reads a value of any type from persistent
storage.
      dynamic get(String key) => _preferenceCache[key];

      /// Reads a value from persistent storage, throwing
an exception if it's not a
      /// bool.
      bool getBool(String key) => _preferenceCache[key];

      /// Reads a value from persistent storage, throwing
an exception if it's not
      /// an int.
      int getInt(String key) => _preferenceCache[key];
```

```
      /// Reads a value from persistent storage, throwing
an exception if it's not a
      /// double.
      double getDouble(String key) =>
_preferenceCache[key];

      /// Reads a value from persistent storage, throwing
an exception if it's not a
      /// String.
      String getString(String key) =>
_preferenceCache[key];

      /// Returns true if persistent storage the contains
the given [key].
      bool containsKey(String key) =>
_preferenceCache.containsKey(key);

      /// Reads a set of string values from persistent
storage, throwing an
      /// exception if it's not a string set.
      List<String> getStringList(String key) {
        List<Object> list = _preferenceCache[key];
        if (list != null && list is! List<String>) {
          list = list.cast<String>().toList();
          _preferenceCache[key] = list;
        }
        // Make a copy of the list so that later mutations
won't propagate
        return list?.toList();
      }

      /// Saves a boolean [value] to persistent storage in
the background.
```

```
      ///
      /// If [value] is null, this is equivalent to calling
[remove()] on the [key].
      Future<bool> setBool(String key, bool value) =>
_setValue('Bool', key, value);

      /// Saves an integer [value] to persistent storage in
the background.
      ///
      /// If [value] is null, this is equivalent to calling
[remove()] on the [key].
      Future<bool> setInt(String key, int value) =>
_setValue('Int', key, value);

      /// Saves a double [value] to persistent storage in
the background.
      ///
      /// Android doesn't support storing doubles, so it
will be stored as a float.
      ///
      /// If [value] is null, this is equivalent to calling
[remove()] on the [key].
      Future<bool> setDouble(String key, double value) =>
          _setValue('Double', key, value);

      /// Saves a string [value] to persistent storage in
the background.
      ///
      /// If [value] is null, this is equivalent to calling
[remove()] on the [key].
      Future<bool> setString(String key, String value) =>
          _setValue('String', key, value);

      /// Saves a list of strings [value] to persistent
```

```
storage in the background.
      ///
      /// If [value] is null, this is equivalent to calling
[remove()] on the [key].
      Future<bool> setStringList(String key, List<String>
value) =>
          _setValue('StringList', key, value);

      /// Removes an entry from persistent storage.
      Future<bool> remove(String key) => _setValue(null,
key, null);

      Future<bool> _setValue(String valueType, String key,
Object value) {
        final Map<String, dynamic> params = <String,
dynamic>{
          'key': '$_prefix$key',
        };
        if (value == null) {
          _preferenceCache.remove(key);
          return _kChannel
              .invokeMethod<bool>('remove', params)
              .then<bool>((dynamic result) => result);
        } else {
          if (value is List<String>) {
            // Make a copy of the list so that later
mutations won't propagate
            _preferenceCache[key] = value.toList();
          } else {
            _preferenceCache[key] = value;
          }
          params['value'] = value;
          return _kChannel
              .invokeMethod<bool>('set$valueType', params)
```

```
              .then<bool>((dynamic result) => result);
        }
      }

      /// Always returns true.
      /// On iOS, synchronize is marked deprecated. On
Android, we commit every set.
      @deprecated
      Future<bool> commit() async => await
_kChannel.invokeMethod<bool>('commit');

      /// Completes with true once the user preferences for
the app has been cleared.
      Future<bool> clear() async {
        _preferenceCache.clear();
        return await _kChannel.invokeMethod<bool>('clear');
      }

      /// Fetches the latest values from the host platform.
      ///
      /// Use this method to observe modifications that
were made in native code
      /// (without using the plugin) while the app is
running.
      Future<void> reload() async {
        final Map<String, Object> preferences =
            await
SharedPreferences._getSharedPreferencesMap();
        _preferenceCache.clear();
        _preferenceCache.addAll(preferences);
      }

      static Future<Map<String, Object>>
_getSharedPreferencesMap() async {
```

```
        final Map<String, Object> fromSystem =
            await _kChannel.invokeMapMethod<String,
Object>('getAll');
        assert(fromSystem != null);
        // Strip the flutter. prefix from the returned
preferences.
        final Map<String, Object> preferencesMap = <String,
Object>{};
        for (String key in fromSystem.keys) {
          assert(key.startsWith(_prefix));
          preferencesMap[key.substring(_prefix.length)] =
fromSystem[key];
        }
        return preferencesMap;
      }

      /// Initializes the shared preferences with mock
values for testing.
      ///
      /// If the singleton instance has been initialized
already, it is automatically reloaded.
      @visibleForTesting
      static void setMockInitialValues(Map<String, dynamic>
values) {
        _kChannel.setMockMethodCallHandler((MethodCall
methodCall) async {
          if (methodCall.method == 'getAll') {
            return values;
          }
          return null;
        });
        _instance?.reload();
      }
    }
```

从源码上看非常简单，首先 SharedPreferences 使用的一个单例模式，且是异步的。它在数据存储上提供了：setInt、setBool、setString 和 setStringList 方法来设置特定类型的数据。在数据读取上提供了：getString 、getInt 、getDouble、getStringList方法来获取数据。此外还提供了如下几个工具API:

1、是否包含有该 key 值从存储：containsKey(String key)

2、移除指定的 key 值存储：remove(String key)

3、清除全部的持久化数据：clear()

4、提交存储数据：commit()，该方法已被废弃。

综合运用：

首先我们在 全局类 Global 中初始化 SharedPreferences ，使用静态方式方便其他地方使用。

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

class Global{

  //定义一个全局的 sp
  static SharedPreferences preferences;

  //初始化
  static void initPreferences() async{
    preferences= await SharedPreferences.getInstance();
  }

}
```

特别注意：SharedPreferences单例的获取是一个异步方法所以需要 await 、async 关键字。

其次：在程序启动时就初始化

```
void main(){

   runApp(MyApp());

   //初始化 sp
   Global.initPreferences();
 }
```

下面就可以在使用的地方直接引用即可，例子：

```
 import 'package:flutter/material.dart';
 import 'package:flutter_learn/util/Global.dart';
 import 'package:flutter_learn/util/ToastUtil.dart';
 import 'package:shared_preferences/
shared_preferences.dart';

 //使用初始化好的静态资源
 SharedPreferences sharedPreferences = Global.preferences;
 //定义一个内存级别的变量
 int count = 0;

 class PersistenPage extends StatefulWidget {
   PersistenPage({Key key}) : super(key: key);

   _PersistenPageState createState() =>
_PersistenPageState();
 }

 class _PersistenPageState extends State<PersistenPage> {
   @override
   Widget build(BuildContext context) {

     count = sharedPreferences.getInt("count") == null
```

```
          ? 0
          : sharedPreferences.getInt("count");

      return Scaffold(
          appBar: AppBar(
            title: Text("持久化"),
          ),
          body: Container(
            color: Colors.blue,
            width: double.infinity,
            child: Column(
              children: <Widget>[
                SizedBox(height: 20),
                Text('ShardPreferences方式持久化',
                    style: TextStyle(color: Colors.white)),
                SizedBox(height: 20),
                Text("当前累计数据：$count"),
                Row(
                  children: <Widget>[
                    SizedBox(width: 5),
                    RaisedButton(
                      color: Colors.deepPurple,
                      elevation: 20,
                      focusElevation: 40,
                      child: Text('自增', style:
TextStyle(color: Colors.white)),
                      onPressed: () {
                        setState(() {
                          count++;

sharedPreferences?.setInt("count", count);
                        });
                      },
                    ),
                    SizedBox(width: 5),
```

```
                    RaisedButton(
                      color: Colors.deepPurple,
                      elevation: 20,
                      focusElevation: 40,
                      child: Text('清除', style:
TextStyle(color: Colors.white)),
                      onPressed: () {
                        setState(() {
                          count = 0;
                          sharedPreferences.clear();
                          //
sharedPreferences.remove("count");
                        });
                      },
                    ),
                    SizedBox(width: 5),
                    RaisedButton(
                      color: Colors.deepPurple,
                      elevation: 20,
                      focusElevation: 40,
                      child: Text('是否包含', style:
TextStyle(color: Colors.white)),
                      onPressed: () {
                        setState(() {
                          count = 0;
                          bool flag =
sharedPreferences.containsKey("count");
                          ToastUtil.show("是否包含:  $flag");
                        });
                      },
                    ),
                    SizedBox(width: 5),
                  ],
                ),
                SizedBox(height: 10),
```

```
            ],
          ),
        ));
  }
}
```

案例效果图：

从上面可以看出，无论是退出当前界面，还是关闭应用后，再次回到持久化界面任然可以读取到存储到文件中的值，也就是实现了数据本地持久化。

四、IO 文件读写操作数据持久化

Flutter 同原生一样支持对文件的读和写。Flutter 中 IO 其实是 Dart 中的一部分，由于 Dart VM 是运行在 PC 或服务器操作系统下，而 Flutter 是运行在移动操作系统中，他们的文件系统所以会有一些差异。为此 PathProvider 插件提供了一种平台透明的方式来访问设备文件系统上的常用位置。该类当前支持访问两个文件系统位置：

1、`临时文件夹`：可以使用 getTemporaryDirectory() 来获取临时目录；在 iOS 上对用的是 NSCachesDirectory 在 Android 对用的是 getCacheDir()。

2、`应用的Documents 目录`：可以使用 getApplicationDocumentsDirectory 来获取 Documents 目录；在 iOS 对应的是 NSDocumentDirectory ，在 Android 上对应的是 AppData 目录。

【特别注意：】临时文件夹在执行系统的清空缓存时会清空该文件夹，documents 目录只有在删除应用时才会清空。

PathProvider 和 SharedPreferences 一样需要引入；具体步骤如下：

1、在 `pubspec.yaml` 文件中添加声明：

```
  dependencies:
    fluttertoast: ^3.0.3
    flutter:
      sdk: flutter
    #添加持久化插件 sp
    shared_preferences: ^0.5.3+1
    #添加文件库
    path_provider: ^1.2.0
```

2、安装依赖库

执行 `flutter packages get` 命令； AS 开发工具直接右上角 `packages get`也可。

3、在使用的文件中导入该库

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

File类的代码比较多，这里就不贴内部源码了，下面只列出开发中常用的 `API` :

1、`create()` 创建文件,多级目录 recursive 为真则递归创建
2、`createSync()` 同步方式创建文件

3、`rename()` 文件重命名
4、`renameSync()` 同步设置文件重命名

5、`copy()` 复制文件
6、`copySync()` 同步复制文件

7、`length()` 获取文件长度
8、`lengthSync()` 同步获取文件长度

9、`absolute` 获取绝对路径文件

10、lastAccessed() 获取最后一次的访问时间
11、lastAccessedSync() 同步获取最后一次的访问时间

12、setLastAccessed() 设置最后一次的访问时间
13、setLastAccessedSync() 同步设置最后一次的访问时间

14、lastModified() 获取最后一次修改时间
15、lastModifiedSync() 同步获取最后一次修改时间

16、setLastModified() 设置最后一次修改时间
17、setLastModifiedSync() 同步设置最后一次修改时间

18、open() 打开文件有多种方式和文件的访问模式，详情参阅源码

19、readAsBytes() 读取字节
20、readAsBytesSync() 同步读取字节

21、readAsString() 读取字符串
22、readAsStringSync() 同步读取字符串

23、readAsLines() 读取一行
24、readAsLinesSync() 同步读取一行

25、writeAsBytes() 写出字节数组
26、writeAsBytesSync() 同步写出字节数组

27、writeAsString() 写出字符串
28、writeAsStringSync() 同步写出字符串

对 File 的操作有太多 API 了，更多内容只有自己去查

阅源码了。下面我们来将之前的计数器实例来改造下，使用文件的方式。

首先我们在 全局类 Global 中创建文件 ， 使用静态方式方便其他地方使用。

```
import 'dart:io';

import 'package:path_provider/path_provider.dart';
import 'package:shared_preferences/
shared_preferences.dart';

class Global{

  //定义一个全局的 sp
  static SharedPreferences preferences;

  static File file;

  //初始化
  static void initPreferences() async{
    preferences= await SharedPreferences.getInstance();
  }

  //初始化一个文件，方便使用
  static void initFile() async{

    final directory = await
getApplicationDocumentsDirectory();
    if (!(file is File)) {
      final String path = directory.path;
      file = File('$path/myInfo.txt');
     if( !file.existsSync()){
       // 不存在则创建文件
```

```
      file.createSync(recursive: true);
    }
   }
  }

}
```

其次：在程序启动时就初始化

```
  void main(){

    runApp(MyApp());

    //初始化 sp
    Global.initPreferences();

    //初始化文件
    Global.initFile();
  }
```

下面就可以在使用的地方直接引用即可，例子：

```
  import 'dart:io';

  import 'package:flutter/material.dart';
  import 'package:flutter_learn/util/Global.dart';
  import 'package:flutter_learn/util/ToastUtil.dart';
  import 'package:shared_preferences/
shared_preferences.dart';

  //使用初始化好的静态资源
  SharedPreferences sharedPreferences = Global.preferences;

  //使用初始化好的本地文件
  File localFile = Global.file;
```

```dart
class PersistenPage extends StatefulWidget {
  PersistenPage({Key key}) : super(key: key);

  _PersistenPageState createState() =>
_PersistenPageState();
}

class _PersistenPageState extends State<PersistenPage> {
  @override
  Widget build(BuildContext context) {

    return Scaffold(
        appBar: AppBar(
          title: Text("持久化"),
        ),
        body: SingleChildScrollView(
          child: Container(
              color: Colors.blue,
              height: 1000,
              child: Column(
                children: <Widget>[

                  // shardPreferences(),

                  IoReadWirte()
                ],
              ),
          ),
        ));
  }


  //文件的读写
  Widget IoReadWirte(){
    //定义一个内存级别的变量
```

```
      String info = localFile.readAsStringSync();
      int count =int.parse(info??0);
      return Column(
        children: <Widget>[
          SizedBox(height: 20),
          Text('文件操作方式持久化',
              style: TextStyle(color: Colors.white)),
          SizedBox(height: 20),
          Text("当前累计数据：$count"),
          Row(
            children: <Widget>[
              SizedBox(width: 5),
              RaisedButton(
                color: Colors.deepPurple,
                elevation: 20,
                focusElevation: 40,
                child: Text('自增', style: TextStyle(color:
Colors.white)),
                onPressed: () {
                  setState(() {
                    count++;

localFile.writeAsStringSync(count.toString());

                  });
                },
              ),
              SizedBox(width: 5),
              RaisedButton(
                color: Colors.deepPurple,
                elevation: 20,
                focusElevation: 40,
                child: Text('自减', style: TextStyle(color:
Colors.white)),
                onPressed: () {
```

```
                  setState(() {
                    count--;

localFile.writeAsStringSync(count.toString());
                  });
                },
              ),
            ],
          ),
          SizedBox(height: 10),
        ],
      );
    }
  }
```

为了方便起见和减少文章篇幅，这里只贴出了 File 操作的全部代码，shardPreferences 中的代码暂时删除，文末的工程实例中拥有全部代码。

效果如下：

无论是用 SharedPreferences 还是文件操作 方式实现数据持久化都是非常简单，在开发中 如果只是存储一些简单数据 其实用 SharedPreferences 就非常好了，如果涉及到一些文件（如：图片，文档等）才会去使用 File。

上面两点都只是对一些单一数据的持久化，而要持久化关系

型数据就显现的捉衿见肘了，对于有关系的数据我们首先想到的肯定是通过数据库去存储，而数据库在开发中也是重中之中，涉及到篇幅问题这里就不介绍了，这也是为下一篇文章做铺垫吧

本章中很多都是通过静态的方式去存值，其实在开发中应该减少这种方式的使用 即在需要使用的时候才去获取，避免不必要的内存消耗！我这里只是为了方便演示才这样写。

好了本章节到此结束，又到了说再见的时候了，如果你喜欢请留下你的小红星，创作真心也不容易；你们的支持才是创作的动力，如有错误，请热心的你留言指正, 谢谢大家观看，下章再会 O(∩_∩)O

实例源码地址： https://github.com/zhengzaihong/flutter_learn