Flutter之shared_preferences的使用、源码分析(二)

卢叁

shared_preferences源码分析

实例化对象源码分析

接下来我们来对shared_preferences进行分析，我们在使用的时候需要通过getInstance实例化一个对象，接下来我们看下这里面它都做了什么操作。

静态变量分析

我们先来看下它定义了三个静态变量:

_prefix: 设置持久化数据和读取持久化数据时统一设置前缀（flutter.）

_completer： 持久化数据异步通知，就是当shared_preferences实例化完成后通过completer.future来返回结果

_manualDartRegistrationNeeded： 是否需要手动注册，因为涉及到Linux、Windows、Mac Os的持久化数据时，是需要手动进行注册的，默认为true

```
static const String _prefix = 'flutter.';
static Completer<SharedPreferences>? _completer;
static bool _manualDartRegistrationNeeded = true;
```

`getInstance()`源码分析

当我们获取实例化对象时先判断`_completer`是否为空，如果不为空则直接返回它的的`future`结果，否则它会实例化一个`SharedPreferences`的`Completer`对象，然后通过`_getSharedPreferencesMap`来获取持久化的`map`对象，获取到`map`对象后，通过`completer.complete(SharedPreferences._(preferencesMap))`将`Map`结果返回出去，代码如下：

```
static Future<SharedPreferences> getInstance() async {
  if (_completer == null) {
    final completer = Completer<SharedPreferences>();
    try {
      final Map<String, Object> preferencesMap =
        await _getSharedPreferencesMap();

completer.complete(SharedPreferences._(preferencesMap));
    } on Exception catch (e) {
      // If there's an error, explicitly return the future
with an error.
      // then set the completer to null so we can retry.
      completer.completeError(e);
      final Future<SharedPreferences> sharedPrefsFuture =
completer.future;
      _completer = null;
      return sharedPrefsFuture;
    }
    _completer = completer;
  }
  return _completer!.future;
```

```
}
```

`_getSharedPreferencesMap()`源码分析

在我们调用`getInstance()`方法里，会调用`_getSharedPreferencesMap()`来获取持久化的Map数据，我们接下来看看它是如何获取的，首先它通过`_store.getAll()`就可以直接获取到本地的所有持久化数据，当我们调用`_store`时，它会判断是否需要手动注册

不需要手动注册时：
在`iOS`、`Android`等平台中使用不需要手动注册，所以它直接就返回的对应的实例对象

需要手动注册时：
先判断是否是`web`，如果是就返回`localStorage`，否则判断是`Linux`还是`Windows`，然后根据平台的不同返回其对应的实例。

```
static Future<Map<String, Object>>
_getSharedPreferencesMap() async {
  final Map<String, Object> fromSystem = await
_store.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;
}

static SharedPreferencesStorePlatform get _store {
  // TODO(egarciad): Remove once auto registration lands on
Flutter stable.
  // https://github.com/flutter/flutter/issues/81421.
  if (_manualDartRegistrationNeeded) {
    // Only do the initial registration if it hasn't
already been overridden
    // with a non-default instance.
    if (!kIsWeb &&
        SharedPreferencesStorePlatform.instance
        is MethodChannelSharedPreferencesStore) {
      if (Platform.isLinux) {
        SharedPreferencesStorePlatform.instance =
SharedPreferencesLinux();
      } else if (Platform.isWindows) {
        SharedPreferencesStorePlatform.instance =
SharedPreferencesWindows();
      }
    }
    _manualDartRegistrationNeeded = false;
  }

  return SharedPreferencesStorePlatform.instance;
}
```

`_setValue()`源码分析

不管我们是存储什么内容的数据，最终都会调用`_setValue()`

来进行存储，

首先它会检查存入的`value`是否为空，如果为空就抛出异常，否则就用`_prefix + key`来作为存入的`key`值。

判断存入的值是不是`List<String>`,如果是先把`value`通过`toList()`方法转换，然后在存入，否则直接存入，这步存入操作只是存入缓存中，当应用程序退出时将消失
最后通过`_store`来异步写入到磁盘中

```
Future<bool> _setValue(String valueType, String key, Object
value) {
  ArgumentError.checkNotNull(value, 'value');
  final String prefixedKey = '$_prefix$key';
  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;
  }
  return _store.setValue(valueType, prefixedKey, value);
}
```

本篇主要讲了`shared_preferences`的源码，下篇讲讲项目中使用`shared_preferences`的封装部分。