1. 背景

SharedPreferences 是用于在本地存储中存储键值对的常用方法。如果没有以适当的方式使用，将会有以下的缺点：

- 在 UI 线程上调用是不安全的，尽管它们具有异步 API，在 UI 线程上调用看起来是安全的，但实际上执行可能导致问题的磁盘 I/O 操作

- 运行时异常并不安全，因为它们会引发解析错误

- 无法发出错误信号

- 它不提供类型安全

1. 简介

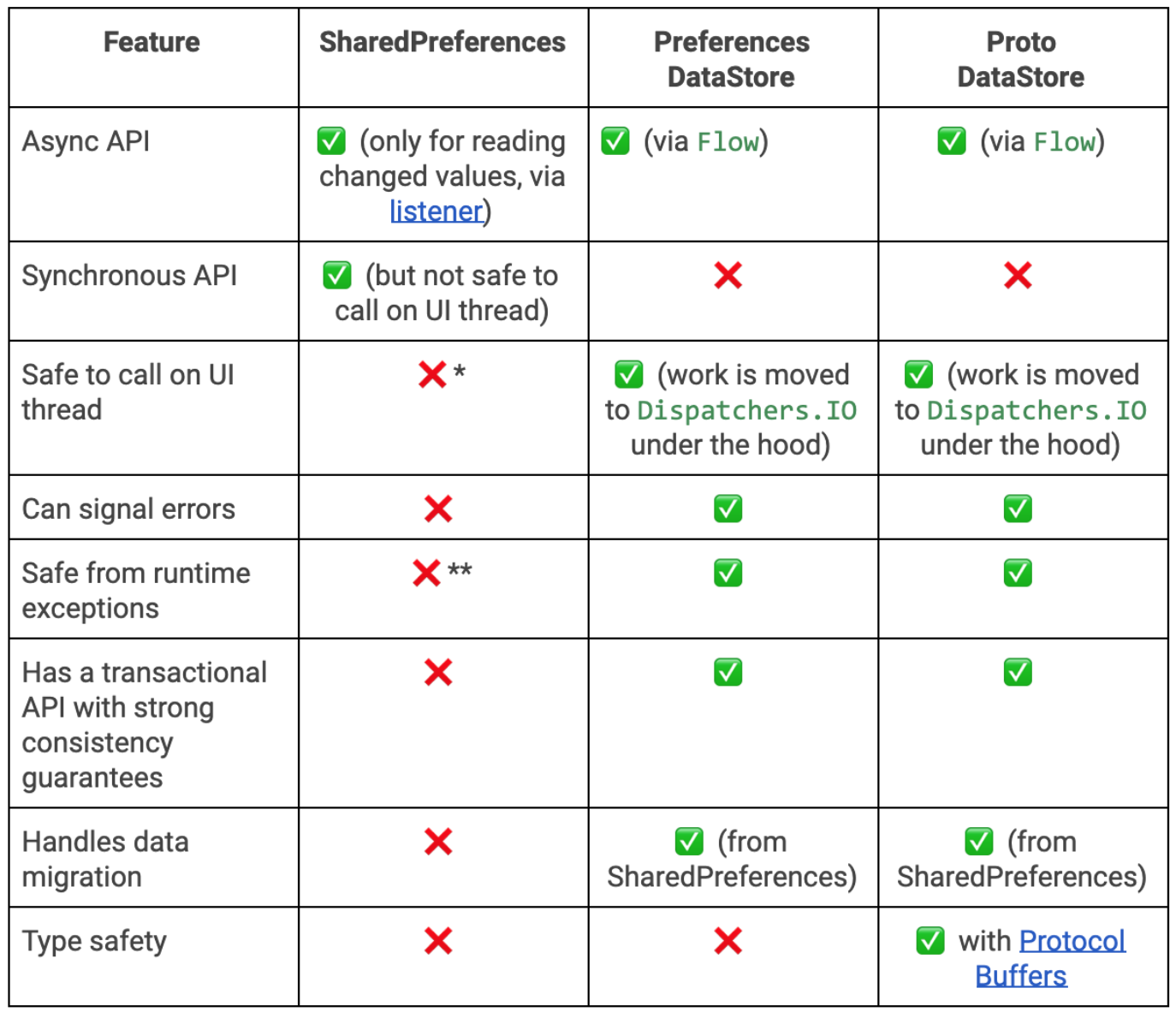
DataStore 是 Jetpack 提供的一种高级数据存储解决方案，用于替代 SharedPreferences。它是使用 Kotlin 协程和 Flow 构建的，以异步、一致和事务方式存储数据。

使用 DataStore 有两种实现方式：

- Preferences DataStore：将数据存储在类似于 Shared Preferences 的键值对中。它不提供任何类型安全。不需要预定义的模式。

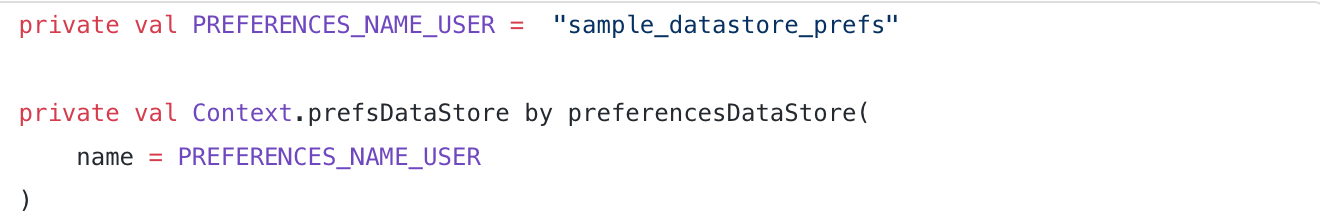
- Proto DataStore：使用协议缓冲区将数据存储为具有预定义模式的自定义数据类型的实例。它提供类型安全。

要知道在这三个选项中使用哪一个，我们应该基本上了解它们提供什么以及它们缺少什么。让我们检查一下差异：



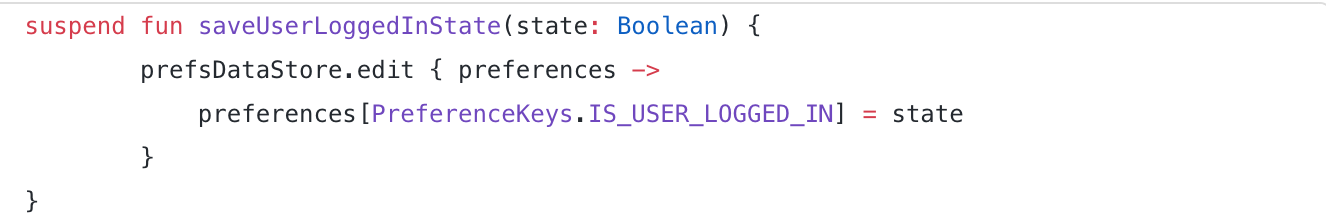
1、PreferencesDataStore

preferencesDataStore 为单个进程 DataStore 创建一个属性委托。这应该只在文件中调用一次，并且 DataStore 的所有用法都应该使用对同一实例的引用。

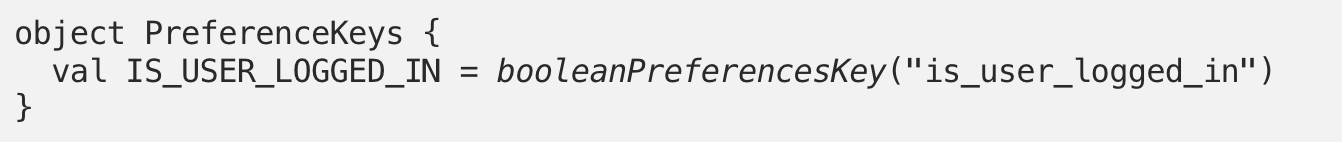


注意：由于我们需要使用相同的实例，因此在初始化时尝试将其保留在 Singleton 类中，并在需要时访问相同的实例

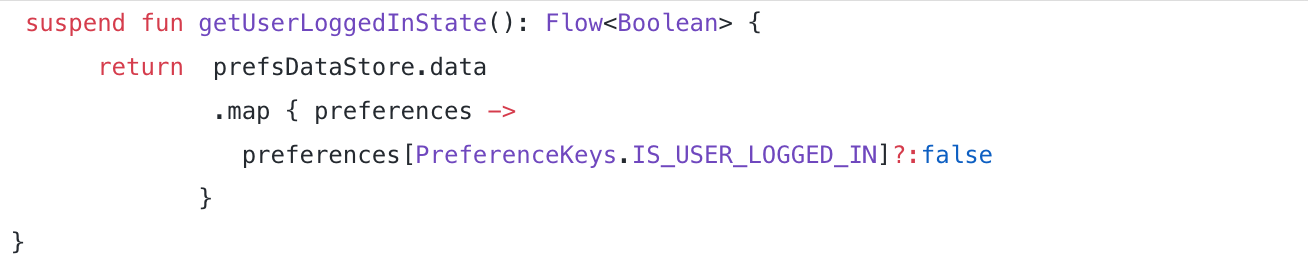
由于我们已经创建了一个 preferenceDataStore 实例的实例，现在让我们看看如何更新其中的数据



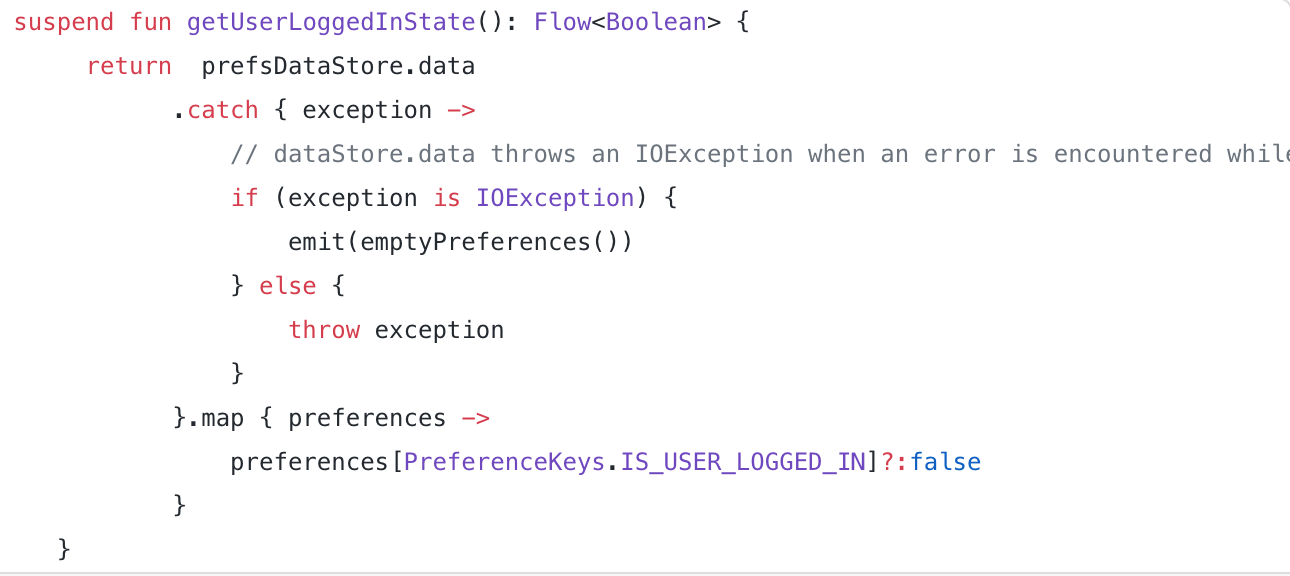
PreferenceKeys 只不过是一个对象类，我们在其中定义与 Preferences 相关的常量



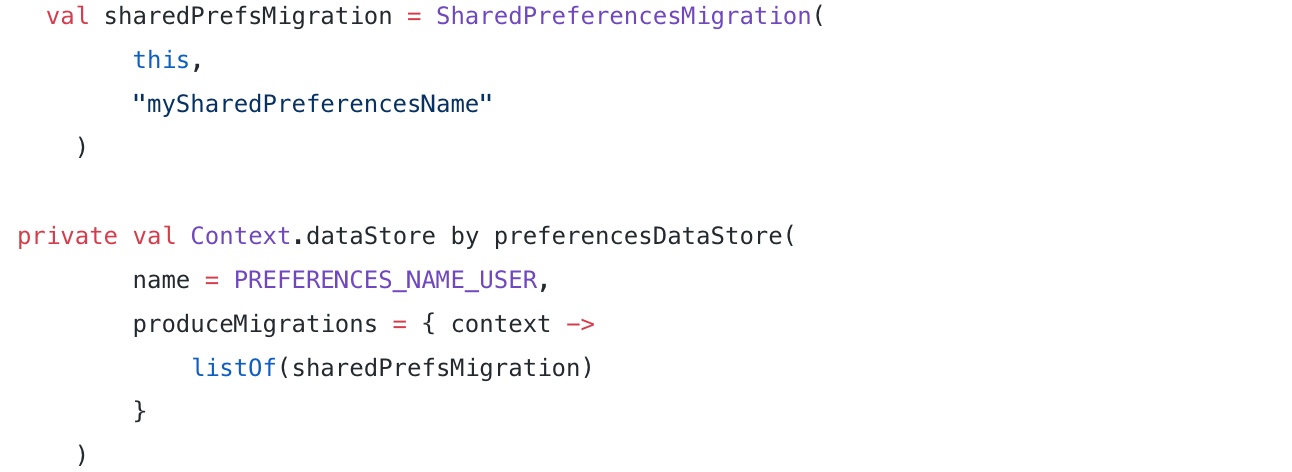
如何从 preferenceDataStore 读取或检索数据



从preferenceDataStore 读取数据时发生错误时可能会抛出IOExceptions。在应用 map 运算符之前使用 catch 处理这些异常是一个最佳实践。如果遇到 IOException，我们会抛出 emptyPreferences，否则我们会重新抛出引发的异常。现在我们的读取块看起来像

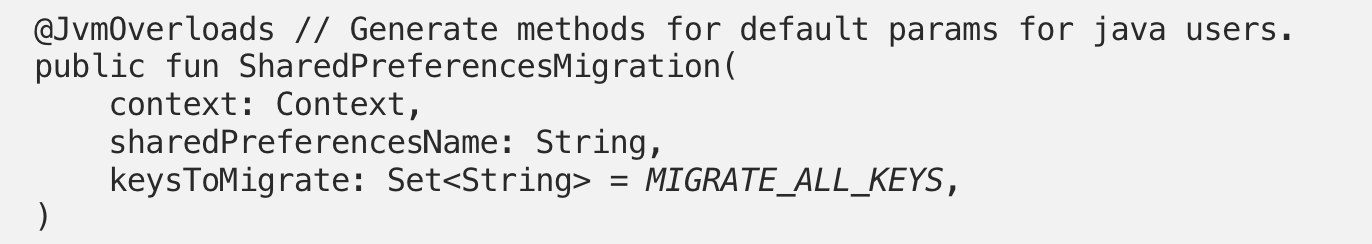


DataStore 为从 SharedPreferences 迁移提供了一种更简单的方法。 SharedPreferencesMigration 是一个 DataMigration 实例，用于从 SharedPreferences 迁移到 DataStore。在构建数据存储时，我们只需要将 SharedPreferences 名称传递给 SharedPreferencesMigration。并且 SharedPreferencesMigration 实例需要添加到 Datastore builder。



注意：key仅从 SharedPreferences 迁移一次，因此一旦代码迁移到 DataStore，您应该停止使用旧的 SharedPreferences

但是，迁移应该在 DataStore 中发生任何数据访问之前运行。我们的迁移必须在使用 DataStore.data 读取任何数据之前以及在使用 DataStore.updateData() 更新任何数据之前成功我们可以指定要迁移的键。这是一个可选参数，如果我们指定集合，那么只有那些键将被迁移，否则它是 MIGRATE\_ALL\_KEYS。 SharedPreferencesMigration 的默认签名。

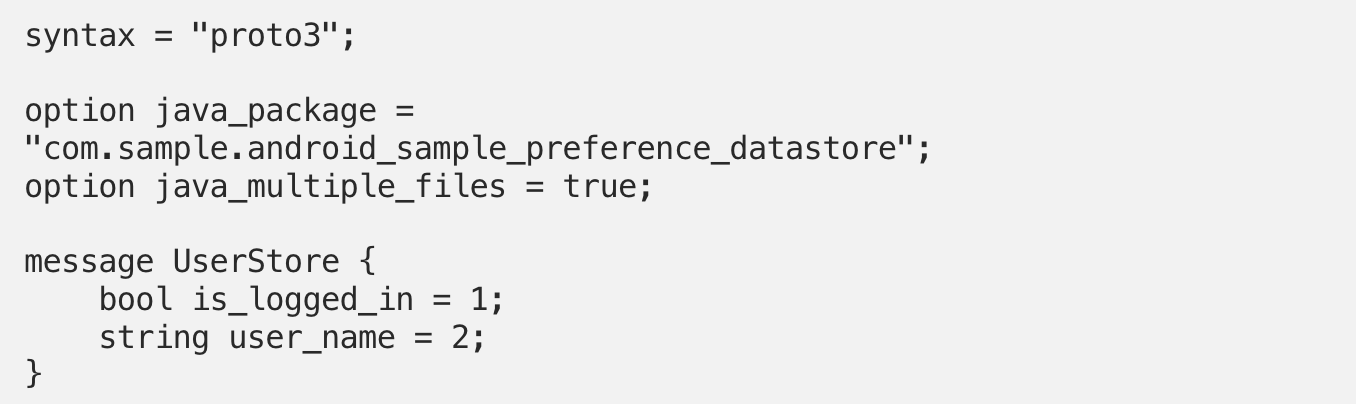


注意：此迁移仅支持基本 SharedPreferences 类型，如 boolean、float、int、long、string 和 string set。如果 getAll 的结果包含其他类型，它们将被忽略。

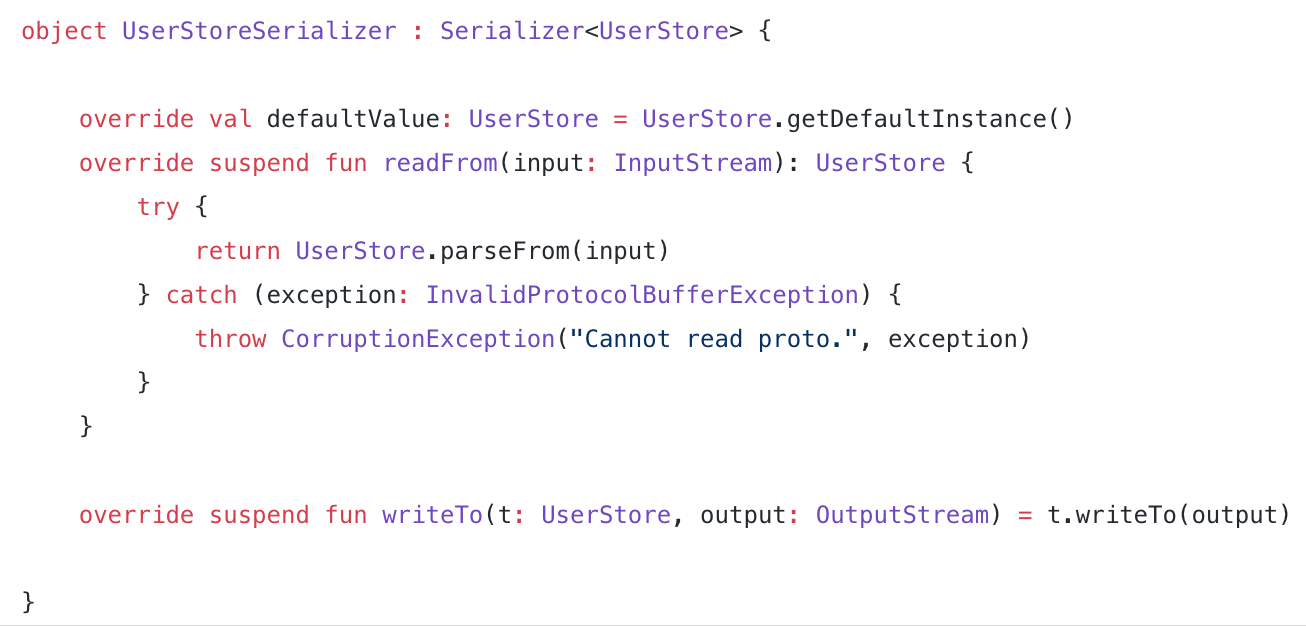
1. ProtoDataStore

协议缓冲区通常被称为 Protobuf，是一种与语言和平台无关的序列化数据机制。 Protobufs 最适合需要通过网络进行更快通信或存储数据的场景。 Google 在 2008 年创建了 ProtoBuf 格式。它是 JSON、XML 的替代解决方案，用于尽可能快地序列化和反序列化数据。

在这种机制中，我们使用带有 .proto 扩展名的 proto 文件，我们在其中写入要序列化的数据。 proto 文件包含我们定义数据的消息类型。

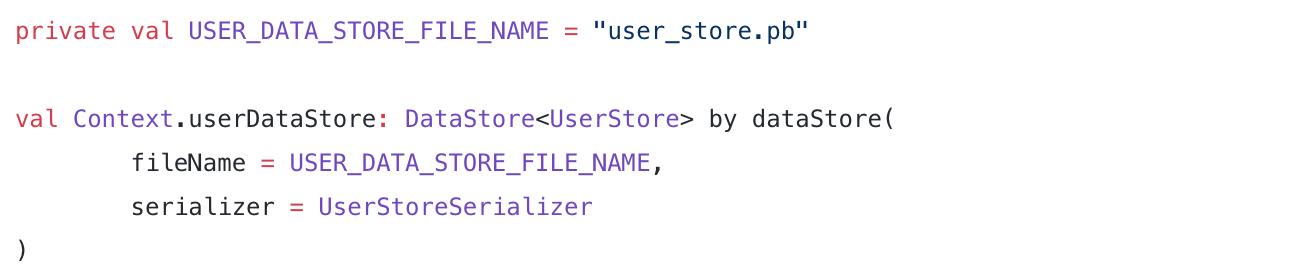


现在是时候创建序列化器了。让我们创建一个实现 Serializer<T> 的类，其中 T 是在 proto 文件中定义的消息类型。这个序列化器告诉数据存储如何读取和写入我们在 proto 文件中定义的数据类型。

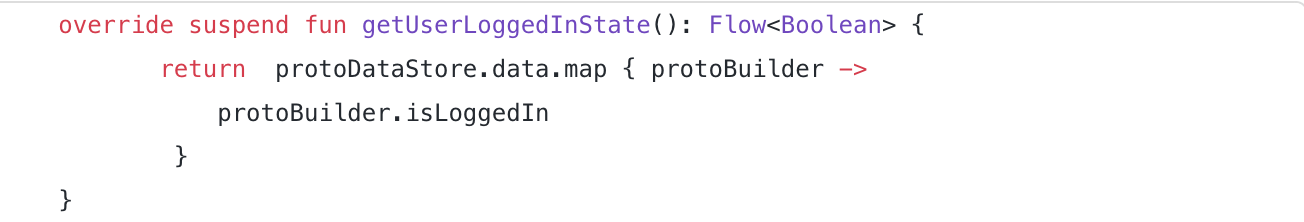


注意：最好在 withContext(Dispatchers.IO) { } 中运行 readFrom 和 writeTo 块并在这两种情况下处理异常，因为方法 readFrom 和 writeTo 可能会抛出 IO 异常。如果有多种消息类型，我们需要定义多个序列化器。

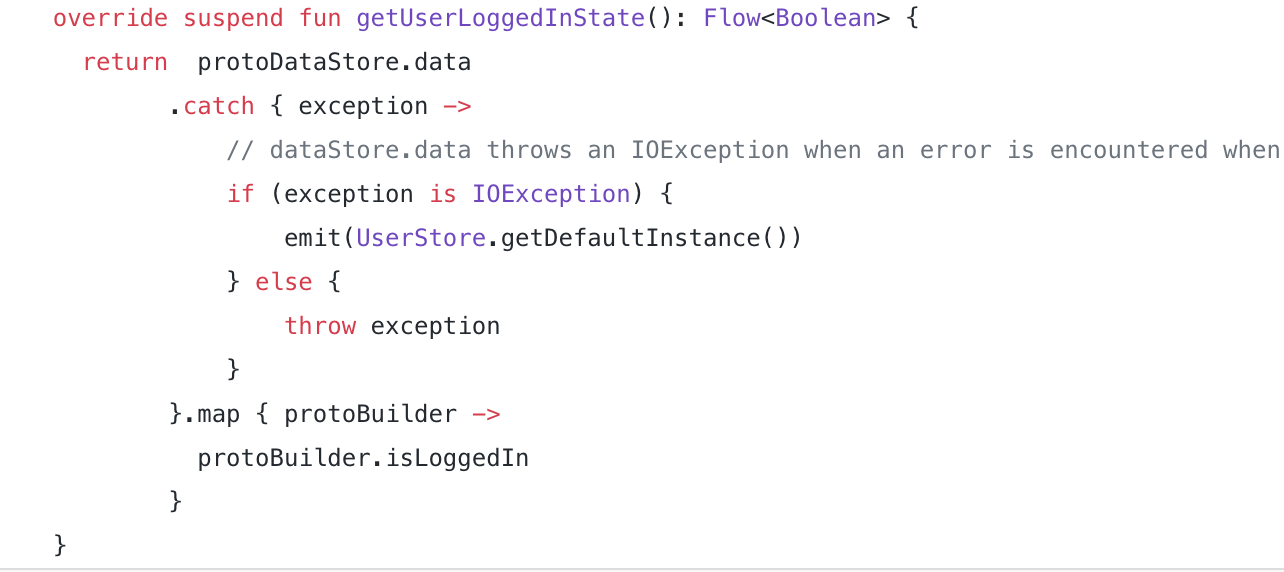
可以使用 dataStore 委托来创建 Datastore 实例。委托需要 2 个强制性输入，它们是名称和序列化程序。dataStore 委托确保我们在应用程序中有一个具有该名称的 DataStore 实例。



与我们在 Preference Datastore 中所做的一样，我们可以使用 DataStore.data 从存储的实例状态中公开特定属性的 Flow。



使用 catch 块读取 wrap 时处理异常



我们有 updatedata() 函数，可以在原子读取-修改-写入操作中以事务方式更新数据。我们获取属性的当前状态，然后在其上写入并保存。

