蚂蚁科技产品手册

统一存储

产品版本： V20210430

文档版本： V20210430

蚂蚁科技技术文档

商标声明

及其他蚂蚁科技服务相关的商标均为蚂蚁科技所有。
本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁科技保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁科技授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁科技授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

目录

1 统一存储简介

mPaaS 提供的统一存储组件是支付宝客户端持久化存储的完整解决方案。该方案的 SDK 在不同平台分别提供了多样化的存储方式以满足不同的存储需求。

功能特性

根据 App 的不同操作平台，mPaaS 的统一存储功能具备以下特性：

- 接入 **Android 客户端**：
  - 支持 SDK 数据库加密。
  - 基于 OrmLite （Object Relational Mapping Lite）框架重构，提供 DAO （Data Access Objects）支持，开发简单易用。
  - 支持基于 SharePreferences 的键值对存储。
  - 支持文件加密存储。
- 接入 **iOS 客户端**：
  - 减少 NSUserDefaults 的使用，不将较大数据和有隐私性数据存储在 NSUserDefaults 里，存取效率相对使用 NSUserDefaults 有大幅提升。
  - 减少业务自动维护文件的情况，减少 Documents、Library 目录下的杂乱文件。
  - 统一存储按存储空间划分为：与用户无关的空间，当前用户的存储空间。业务层无需关注用户切换，并且不需要使用 userId 来获取当前用户数据。
  - 基于 sqlite，提供 DAO （Data Access Objects） 支持，相比 CoreData 更加灵活。通过配置文件将数据库操作封装起来并与业务隔离。业务层使用接口存取数据、操作数据库表。
  - 底层提供数据加密支持。
  - 提供多样化的存储方式，满足不同需求，并提供高效的内存缓存。

2 接入 Android

2.1 快速开始

统一存储支持 **原生 AAR 接入**、**mPaaS Inside 接入** 和 **组件化接入** 三种接入方式。根据不同的业务需求，可实现数据库存储、键值对存储和文件存储多种存储方式。

前置条件

- 若采用原生 AAR 方式接入，需先完成 将 mPaaS 添加到您的项目中 的前提条件和后续相关步骤。
- 若采用 mPaaS Inside 方式接入，需先完成 mPaaS Inside 接入流程 。
- 若采用组件化方式接入，需先完成 组件化接入流程 。

添加 SDK

原生 AAR 方式

参考 AAR 组件管理 ，通过 **组件管理（AAR）** 在工程中安装 **存储** 组件。

mPaaS Inside 方式

在工程中通过 **组件管理** 安装 **存储** 组件。
更多信息，参考 管理组件依赖 。

组件化方式

在 Portal 和 Bundle 工程中通过 **组件管理** 安装 **存储** 组件。
更多信息，参考 管理组件依赖 。

初始化 mPaaS

如果使用原生 AAR 方式 / mPaaS Inside 方式，需要初始化 mPaaS。
在 Application 中添加以下代码：

```
public class MyApplication extends Application {
@Override
protected void attachBaseContext(Context base) {
super.attachBaseContext(base);
// mPaaS 初始化回调设置
QuinoxlessFramework.setup(this, new IInitCallback() {
@Override
public void onPostInit() {
// 此回调表示 mPaaS 已经初始化完成，mPaaS 相关调用可在这个回调里进行
}
});
}
@Override
public void onCreate() {
super.onCreate();
// mPaaS 初始化
QuinoxlessFramework.init();
}
}
```

使用存储

如需使用数据库存储相关内容，参考 数据库存储 。

如需使用键值对存储相关内容，参考 键值对存储 。

如需使用文件存储相关内容，参考 文件存储 。

代码示例

参考 代码示例，获取示例代码。

2.2 进阶指南

数据库存储

关于数据库存储的使用方法，参考 数据库存储 。

键值对存储

关于键值对存储的使用方法，参考 键值对存储 。

文件存储

关于文件存储的使用方法，参考 文件存储 。

3 接入 iOS

统一存储支持 **基于 mPaaS 框架接入**、**基于已有工程且使用 mPaaS 插件接入** 和 **基于已有工程且使用 CocoaPods 接入** 三种接入方式。根据不同的业务需求，可实现多种数据存储方案。包括：APDataCenter、KV 存储、DAO 存储、LRU 存储、自定义存储、数据清理。

本文档介绍如何使用 Xcode 添加统一存储模块到工程中。

前置条件

您已经接入工程到 mPaaS。更多信息，请参见以下内容：

- 基于 mPaaS 框架接入
- 基于已有工程且使用 mPaaS 插件接入
- 基于已有工程且使用 CocoaPods 接入

添加 SDK

根据您采用的接入方式，请选择相应的添加方式。

- 使用 mPaaS Xcode Extension。
  此方式适用于采用了 **基于 mPaaS 框架接入** 或 **基于已有工程且使用 mPaaS 插件接入** 的接入方式。
  - 点击 Xcode 菜单项 **Editor** > **mPaaS** > **编辑工程**，打开编辑工程页面。
  - 选择 **统一存储**，保存后点击 **开始编辑**，即可完成添加。

- 使用 cocoapods-mPaaS 插件。此方式适用于采用了 **基于已有工程且使用 CocoaPods 接入** 的接入方式。
    - 在 Podfile 文件中，使用 mPaaS_pod"mPaaS_DataCenter" 添加统一存储组件依赖。
    - 执行 pod install 即可完成接入。

使用 SDK

参考 统一存储 官方 Demo 在 10.1.60 及以上版本的基线中使用统一存储组件 SDK。

4 存储类型

4.1 存储类型简介

Android 存储类型

接入 Android 客户端的统一存储组件提供以下持久化存储方案：

- 数据库存储 ：基于 OrmLite 架构，提供了数据库底层加密能力。
- 键值对存储 ：基于 Android 原生的 SharedPreferences，同时进行了一定的包装，提升了易用性。
- 文件存储 ：基于 Android 原生 File，提供了文件加密能力。

iOS 存储类型

接入 iOS 客户端的统一存储组件提供以下持久化存储方案：

- APDataCenter ：统一存储的入口类。
- KV 存储 ：提供接口存储，简化客户端持久化对象的复杂度。
- DAO 存储 ：当业务有 sqlite 访问需要时，可由统一存储的 DAO 功能进行简化和封装。
- LRU 存储 ：提供内存缓存和磁盘缓存的存储方法。
- 自定义存储 ：提供 APCustomStorage 存储、APAsyncFileArrayService 存储、APObjectArrayService 存储等自定义存储方式。
- 数据清理 ：创建自动维护容量的缓存目录、提供清理缓存的实现类。

相关的公开类说明，如下表所示：

类名	功能
APDataCenter	单例类，统一存储入口类。
APSharedPreferences	对应一个数据库文件，提供 Key-Value 存储接口，同时容纳 DAO 建表。
APDataCrypt	对称加密结构体。
APLRUDiskCache	支持 LRU 淘汰规则的磁盘缓存。
APLRUMemoryCache	支持 LRU 淘汰规则的内存缓存，线程安全。
APObjectArrayService	基于 DAO，可以分业务对支持 NSCoding 的对象提供持久化，支持加密、容量限制与内存缓存。
APAsyncFileArrayService	基于 DAO，对二进制数据提供持久化，支持加密、容量限制与内存缓存。
APCustomStorage	自定义存储空间，同时在这个空间内提供完整的用户管理，Key-Value、DAO 存储功能。
APDAOProtocol	接口描述，为 DAO 对象支持的接口。

4.2 Android 存储类型

4.2.1 数据库存储

mPaaS 提供的数据库存储基于 OrmLite 架构，提供了数据库底层加密能力。数据库的增、删、改、查可以使用 com.j256.ormlite.dao.Dao 的接口来调用。

重要：在使用数据库时，请不要对原有数据库直接进行加密，否则会引发 native 层解密崩溃。建议您先创建新的加密数据库，再将原有数据库内容拷贝至新建的加密数据库中。

使用示例

- 生成数据表
- 创建 OrmLiteSqliteOpenHelper
- 查询数据
- 插入数据
- 删除数据

生成数据表

```
// 数据库表名，默认为类名
@DatabaseTable
public class User {
// 主键
@DatabaseField(generatedId = true)
public int id;
// name字段唯一
@DatabaseField(unique = true)
public String name;
@DatabaseField
public int color;
@DatabaseField
public long timestamp;
}
```

创建 OrmLiteSqliteOpenHelper

自定义一个 DemoOrmLiteSqliteOpenHelper 继承自 OrmLiteSqliteOpenHelper。通过OrmLiteSqliteOpenHelper，可以创建数据库并对数据库加密。

```
public class DemoOrmLiteSqliteOpenHelper extends OrmLiteSqliteOpenHelper {

/**
* 数据库名称
*/
private static final String DB_NAME ="com_mpaas_demo_storage.db";

/**
* 当前数据库版本
*/
private static final int DB_VERSION = 1;

/**
* 数据库加密密钥，mPaaS 支持数据库加密，使数据在设备上更安全，若为 null 则不加密。
```

```
* 注意：密码只能设置一次，不提供修改密码的 API；不支持对未加密的库设置密码进行加密（会导致闪退）。
*/
private static final String DB_PASSWORD ="mpaas";

public DemoOrmLiteSqliteOpenHelper(Context context) {
super(context, DB_NAME, null, DB_VERSION);
setPassword(DB_PASSWORD);
}

/**
* 数据库创建时的回调函数
*
* @param sqLiteDatabase 数据库
* @param connectionSource 连接
*/
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase, ConnectionSource connectionSource) {
try {
// 创建User表
TableUtils.createTableIfNotExists(connectionSource, User.class);
} catch (SQLException e) {
e.printStackTrace();
}
}

/**
* 数据库更新时的回调函数
*
* @param database 数据库
* @param connectionSource 连接
* @param oldVersion 旧数据库版本
* @param newVersion 新数据库版本
*/
@Override
public void onUpgrade(SQLiteDatabase database, ConnectionSource connectionSource, int oldVersion, int
newVersion) {
try {
// 删除旧版User表，忽略错误
TableUtils.dropTable(connectionSource, User.class, true);
} catch (SQLException e) {
e.printStackTrace();
}
try {
// 从新创建User表
TableUtils.createTableIfNotExists(connectionSource, User.class);
} catch (SQLException e) {
e.printStackTrace();
}
}
}
```

查询数据

这里是查询User表的全部数据并按照timestamp字段进行升序排列。

```
/**
* 初始化DB数据
*/
private void initData() {
mData.clear();
try {
mData.addAll(mDbHelper.getDao(User.class).queryBuilder().orderBy("timestamp", true).query());
} catch (SQLException e) {
e.printStackTrace();
}
}
```

插入数据

```
/**
* 插入用户信息
*
* @param user 用户信息
*/
private void insertUser(User user) {
if (null == user) {
return;
}
try {
// mDbHelper = new DemoOrmLiteSqliteOpenHelper(this); 更多信息，请参见上文 创建 OrmLiteSqliteOpenHelper
mDbHelper.getDao(User.class).create(user);
} catch (SQLException e) {
e.printStackTrace();
}
}
```

删除数据

```
/**
* 删除用户信息
*
* @param user 用户信息
*/
private void deleteUser(User user) {
try {
// mDbHelper = new DemoOrmLiteSqliteOpenHelper(this); 更多信息，请参见上文 创建 OrmLiteSqliteOpenHelper
mDbHelper.getDao(User.class).delete(user);
} catch (SQLException e) {
e.printStackTrace();
}
}
```

4.2.2 键值对存储

mPaaS 提供的键值对存储类似 Android 原生的 SharedPreferences，提供了类似的接口，底层是 mPaaS 自主实现的键值对存储系统。

使用示例

- 创建 APSharedPreferences
- 查询数据
- 插入数据
- 删除数据

创建 APSharedPreferences

```
// context为Android上下文，GROUP_ID可以理解为SharedPreferences的文件名
APSharedPreferences mAPSharedPreferences = SharedPreferencesManager.getInstance(context, GROUP_ID);
```

查询数据

```
/**
* 初始化键值对数据
*/
private void initData() {
try {
// 获取所有键值对信息
aMap.putAll((Map<String, String>) mAPSharedPreferences.getAll());
} catch (Exception e) {
e.printStackTrace();
}
}
```

插入数据

```
/**
* 插入键值对
*
* @param key key
* @param value value
*/
private void insertKeyValue(String key, String value) {
mAPSharedPreferences.putString(key, value);
mAPSharedPreferences.commit();
}
```

删除数据

```
/**
* 删除键值对
*
* @param key key
*/
private void deleteKeyValue(String key) {
mAPSharedPreferences.remove(key);
```

```
mAPSharedPreferences.commit();
}
```

4.2.3 文件存储

mPaaS 提供的文件存储基于 Android 原生 File，提供了加密能力。

说明：由于文件加密采用无线保镖提供的加密功能，请确保无线保镖加密图片已正确生成。

文件存储类型

- **ZFile**：该文件类型存储在 data/data/package_name/files 下。
- **ZExternalFile**：该文件类型存储在 sdcard/Android/data/package_name/files 下。
- **ZFileInputStream/ZFileOutputStream**：文件存储输入输出流，使用该流则不进行加解密。
- **ZSecurityFileInputStream/ZSecurityFileOutputStream**：文件存储安全输入输出流，使用该流则会进行加解密。

使用示例

- 文件转文本
- 文本转文件
- 插入文件
- 删除文件

文件转文本

```
/**
* 文件转文本
* @param file 文件
* @return 文本
*/
public String file2String(File file) {
InputStreamReader reader = null;
StringWriter writer = new StringWriter();
try {
// 使用解密输入流ZSecurityFileInputStream
// 如果不使用加解密功能，则请使用ZFileInputStream
reader = new InputStreamReader(new ZSecurityFileInputStream(file, this));
//将输入流写入输出流
char[] buffer = new char[DEFAULT_BUFFER_SIZE];
int n = 0;
while (-1 != (n = reader.read(buffer))) {
writer.write(buffer, 0, n);
}
} catch (Exception e) {
e.printStackTrace();
return null;
} finally {
if (reader != null)
```

```
try {
reader.close();
} catch (IOException e) {
e.printStackTrace();
}
}
//返回转换结果
if (writer != null) {
return writer.toString();
} else {
return null;
}
}
```

文本转文件

```
/**
* 文本转文件
* @param res 文本
* @param file 文件
* @return true表示成功，反之失败
*/
public boolean string2File(String res, File file) {
boolean flag = true;
BufferedReader bufferedReader = null;
BufferedWriter bufferedWriter;
try {
bufferedReader = new BufferedReader(new StringReader(res));
// 使用加密输出流ZSecurityFileOutputStream
// 如果不使用加解密功能，则请使用ZFileOutputStream
bufferedWriter = new BufferedWriter(new OutputStreamWriter(new ZSecurityFileOutputStream(file, this)));
//字符缓冲区
char buf[] = new char[DEFAULT_BUFFER_SIZE];
int len;
while ((len = bufferedReader.read(buf)) != -1) {
bufferedWriter.write(buf, 0, len);
}
bufferedWriter.flush();
bufferedReader.close();
bufferedWriter.close();
} catch (Exception e) {
e.printStackTrace();
flag = false;
return flag;
} finally {
if (bufferedReader != null) {
try {
bufferedReader.close();
} catch (IOException e) {
e.printStackTrace();
}
}
}
return flag;
}
```

插入文件

```
/**
* 插入文件
*
* @param file 文件
*/
private void insertFile(BaseFile file) {
if (null == file) {
return;
}
StringBuilder sb = new StringBuilder();
String content = sb.append(file.getName())
.append(' ')
.append(SIMPLE_DATE_FORMAT.format(new Date(System.currentTimeMillis()))).toString();
string2File(content, file);
try {
if (!file.exists()) {
file.createNewFile();
}
mData.add(file);
} catch (Exception e) {
e.printStackTrace();
}
}
```

删除文件

```
/**
* 删除文件
*
* @param file 文件
*/
private void deleteFile(BaseFile file) {
try {
file.delete();
mData.remove(file);
} catch (Exception e) {
e.printStackTrace();
}
}
```

4.3 iOS 存储类型

4.3.1 APDataCenter

APDataCenter 为统一存储的入口类，为一个单例，可在代码任意地方调用。

```
[APDataCenter defaultDataCenter]
```

也可以使用宏。

```
#define APDefaultDataCenter [APDataCenter defaultDataCenter]
```

接口介绍

宏定义

```
#define APDefaultDataCenter [APDataCenter defaultDataCenter]
#define APCommonPreferences [APDefaultDataCenter commonPreferences]
#define APUserPreferences [APDefaultDataCenter userPreferences]
#define APCurrentVersionStorage [APDefaultDataCenter currentVersionStorage]
```

常量定义

以下几个事件通知业务代码通常无须关注，但是统一存储会抛出这些通知。

```
/**
* 前一个用户的数据库文件将要关闭的事件通知
*/
extern NSString* const kAPDataCenterWillLastUserResign;

/**
* 用户状态已经发生切换的通知。有可能是 user 变为 nil 了，具体 userId 可以用 currentUserId 来获取。
* 这个通知附加的 object 是个字典，如果不为 nil，里面@"switched"这个键值返回@YES 表示确实发生了用户切换事件。
*/
extern NSString* const kAPDataCenterDidUserUpdated;

/**
* 用户并没切换，APDataCenter 重新收到登入事件。会抛这个通知。
*/
extern NSString* const kAPDataCenterDidUserRenew;
```

接口与属性

void APDataCenterLogSwitch(BOOL on);

打开或关闭统一存储的控制台 log，默认为打开。

@property (atomic, strong, readonly) NSString* currentUserId;

获取当前登录用户的 userId。

(NSString*)preferencesRootPath;

获取存储 commonPreferences 和 userPreferences 数据库文件夹的路径。

(void)setCurrentUserId:(NSString*)currentUserId;

设置当前登录的用户 ID，业务代码请不要调用，需要由登录模块调用。设置用户 ID 后，userPreferences 会指向这个用户的数据库。

(void)reset;

完全重置整个统一存储目录，请谨慎使用。

(APSharedPreferences*)commonPreferences;

与用户无关的全局存储数据库。

(APSharedPreferences*)userPreferences;

当前登录用户的存储数据库。不是登录态时，取到的是 nil。

(APSharedPreferences*)preferencesForUser:(NSString*)userId;

返回指定用户 id 的存储对象，业务层通常使用 userPreferences 方法即可。当有异步存储需要时，防止窜数据，可以使用该方法取特定用户的存储数据库。

(APPreferencesAccessor*)accessorForBusiness:(NSString*)business;

根据 business 名生成一个存取器，业务层需要自行持有这个对象。使用这个存取器后，访问 KV 存储就不需要再传 business 了。

```
APPreferencesAccessor* accessor = [[APDataCenter defaultDataCenter] accessorForBusiness:@"aBiz"];
[[accessor commonPreferences] doubleForKey:@"aKey"];

// 等价于

[[[APDataCenter defaultDataCenter] commonPreferences] doubleForKey:@"aKey"business:@"aBiz"];
```

(APCustomStorage*)currentVersionStorage;

统一存储会维护一个当前版本的数据库，当版本发生升级时，这个数据库会重置。

(id<APDAOProtocol>)daoWithPath:(NSString*)filePath userDependent:(BOOL)userDependent;

从一个配置文件生成 DAO 访问对象。

参数说明

参数	说明
filePath	DAO 配置文件的文件路径。在 main bundle 里的文件使用下面方式： NSString* filePath = [[NSBundle mainBundle] pathForResource:@"file"ofType:@"xml"];
userDependent	指定这个 DAO 对象操作哪个数据库。 如果 userDependent=NO，表示与用户无关，那么 DAO 会在 commonPreferences 的数据库文件中建表。 如果 userDependent=YES，那么 DAO 对象会在 userPreferences 的数据库文件中建表。 当切换用户后，后续的 DAO 操作会自动在更换后的用户文件中进行，业务无须关心用户切换。

返回值

DAO 对象。业务不用关心它的类名，只需要使用业务自己定义的 id<AProtocol> 强制转换一下即可。返回的 DAO 对象，在需要时也可以使用 id<APDAOProtocol> 进行转换，调用默认提供的方法。所以自定义的 AProtocol 不要含有 APDAOProtocol 里定义的方法。

(id<APDAOProtocol>)daoWithPath:(NSString*)filePath databasePath:(NSString*)databasePath;

创建一个维护自己独立数据库文件的 DAO 访问对象，而不使用 APSharedPreferences。使用 daoWithPath:userDependent: 接口创建的 DAO 对象，操作的是 commonPreferences 或 userPreferences。这个接口会创建一个 DAO 对象，并且操作的是 databasePath 指定的特定数据库文件，若文件不存在则会创建此文件。可以创建多个 DAO 对象，指定相同的 databasePath。

参数说明

参数	说明
filePath	同 daoWithPath:userDependent: 接口。
databasePath	DAO 数据库文件的位置，可以传绝对路径，也可传 Documents/XXXX.db 或 Library/Movie/XXX.db 这样的相对路径。

返回值

DAO 对象。

4.3.2 KV 存储

客户端许多场景下使用 Key-Value 存储就能很好的满足需求，通常我们会使用 NSUserDefaults，但 NSUserDefaults 不支持加密，持久化速度慢。

统一存储的 Key-Value 存储提供接口存储：NSInteger、long long（在 64 位系统上与 NSInteger 相同）、BOOL、double、NSString 等类型的 PList 对象，支持 NSCoding 的对象，以及可通过反射转换成 JSON 表达的 Objective-C 对象，极大简化客户端持久化对象的复杂度。

关于 Key-Value 存储中大部分接口的说明，请参考 APSharedPreferences.h 头文件方法描述。

存储基本类型

统一存储提供下列接口存储基本类型。

```
- (NSInteger)integerForKey:(NSString*)key business:(NSString*)business;
- (NSInteger)integerForKey:(NSString*)key business:(NSString*)business defaultValue:(NSInteger)defaultValue; // 当数据不存在时返回默认值
- (void)setInteger:(NSInteger)value forKey:(NSString*)key business:(NSString*)business;

- (long long)longLongForKey:(NSString*)key business:(NSString*)business;
- (long long)longLongForKey:(NSString*)key business:(NSString*)business defaultValue:(long long)defaultValue; // 当数据不存在时返回默认值
- (void)setLongLong:(long long)value forKey:(NSString*)key business:(NSString*)business;

- (BOOL)boolForKey:(NSString*)key business:(NSString*)business;
- (BOOL)boolForKey:(NSString*)key business:(NSString*)business defaultValue:(BOOL)defaultValue; // 当数据不存在时返回默认值
- (void)setBool:(BOOL)value forKey:(NSString*)key business:(NSString*)business;

- (double)doubleForKey:(NSString*)key business:(NSString*)business;
- (double)doubleForKey:(NSString*)key business:(NSString*)business defaultValue:(double)defaultValue; // 当数据不存在时返回默认值
- (void)setDouble:(double)value forKey:(NSString*)key business:(NSString*)business;
```

其中 defaultValue 参数为数据不存在时返回的默认值。

存储 Objective-C 对象

接口说明

统一存储提供下列接口存储 Objective-C 对象。

```
- (NSString*)stringForKey:(NSString*)key business:(NSString*)business;
- (NSString*)stringForKey:(NSString*)key business:(NSString*)business extension:(APDataCrypt*)extension;
- (void)setString:(NSString*)string forKey:(NSString*)key business:(NSString*)business;
- (void)setString:(NSString*)string forKey:(NSString*)key business:(NSString*)business
extension:(APDataCrypt*)extension;

- (id)objectForKey:(NSString*)key business:(NSString*)business;
- (id)objectForKey:(NSString*)key business:(NSString*)business extension:(APDataCrypt*)extension;

- (void)setObject:(id)object forKey:(NSString*)key business:(NSString*)business;
- (void)setObject:(id)object forKey:(NSString*)key business:(NSString*)business extension:(APDataCrypt*)extension;
- (BOOL)setObject:(id)object forKey:(NSString*)key business:(NSString*)business extension:(APDataCrypt*)extension
options:(APDataOptions)options;

- (void)archiveObject:(id)object forKey:(NSString*)key business:(NSString*)business;
- (void)archiveObject:(id)object forKey:(NSString*)key business:(NSString*)business
extension:(APDataCrypt*)extension;
- (BOOL)archiveObject:(id)object forKey:(NSString*)key business:(NSString*)business
extension:(APDataCrypt*)extension options:(APDataOptions)options;

- (void)saveJsonObject:(id)object forKey:(NSString*)key business:(NSString*)business;
- (void)saveJsonObject:(id)object forKey:(NSString*)key business:(NSString*)business
extension:(APDataCrypt*)extension;
- (BOOL)saveJsonObject:(id)object forKey:(NSString*)key business:(NSString*)business
extension:(APDataCrypt*)extension options:(APDataOptions)options;
```

setString & stringForKey

保存 NSString 时，推荐使用 setString，stringForKey 接口，名称上更有解释性。

如果数据未加密，使用这个接口存储的字符串，可以通过 SQLite DB 查看器看到，更直观。使用 setObject 保存的字符串会首先通过 Property List 转成 NSData 再保存到数据库里。

setObject

保存 Property List 对象时，建议使用 setObject，这样效率最高。

Property List 对象： NSNumber、NSString、NSData、NSDate、NSArray、NSDictionary，NSArray 和 NSDictionary 里的子对象也只能是 PList 对象。

使用 setObject 保存 Property List 后，使用 objectForKey 取到的对象是 Mutable 的。下面代码里拿到的 savedArray 是 NSMutableArray。

```
NSArray* array = [[NSArray alloc] initWithObjects:@"str", nil];
[APCommonPreferences setObject:array forKey:@"array"business:@"biz"];
```

```
NSArray* savedArray = [APCommonPreferences objectForKey:@"array"business:@"biz"];
```

archiveObject

对于支持 NSCoding 协议的 Objective-C 对象，统一存储调用系统的 NSKeyedArchiver 将对象转成 NSData 并进行持久化。

Property List 对象也可以使用这个接口，但效率比较低，不推荐。

saveJsonObject

当一个 Objective-C 对象既不是 Property List 对象，也不支持 NSCoding 协议时，可以使用这个方法进行持久化。

该方法通过运行时动态反射，将 Objective-C 对象映射成 JSON 字符串。但不是所有 Objective-C 对象都可以使用该方法保存，比如含有 C 结构体指针属性的 Objective-C 对象，互相引用的 Objective-C 对象，属性里有字典或数组的 Objective-C 对象。

objectForKey

统一存储保存 Objective-C 对象数据时，会同时记录它的归档方式，取对象统一使用 objectForKey。

说明：使用 setString 保存的字符串需要使用 stringForKey 获取。

数据加密

使用默认加密

带 extension 的接口支持加密，传入 APDataCrypt 结构体。

APDefaultEncrypt 为默认加密方法，为 AES 对称加密。

APDefaultDecrypt 为默认解密方法，与 APDefaultEncrypt 使用相同密钥。

通常情况下，使用统一存储提供的默认加密即可，如下所示：

```
[APUserPreferences setObject:aObject forKey:@"key"business:@"biz"extension:APDefaultEncrypt()];

id obj = [APUserPreferences objectForKey:@"key"business:@"biz"extension:APDefaultDecrypt()];
// or
id obj = [APUserPreferences objectForKey:@"key"business:@"biz"];
```

因为使用的是默认加密，所以取数据的接口可以省略 extension 参数。

使用自定义加密方法

如果业务有更高的加密安全要求，可以自己实现 APDataCrypt 结构体，并指定加密、解密的函数指针。但请保证加密与解密方法对应，这样才能正确保存、还原数据。

基础类型加密

如果想加密存储 BOOL、NSInteger、double、long long 类型的对象，可以把它们转成字符串，或放到 NSNumber 里，再调用 setString、setObject 接口即可。

指定 options

```
typedef NS_OPTIONS (unsigned int, APDataOptions)
{
// 这两个选项不要在接口中使用，是标识数据的加密属性的，请使用 extension 来传递加密方法
APDataOptionDefaultEncrypted = 1 << 0, // 这个选项不要传，传了也没效果，统一存储会使用接口里的 extension 来做加密的判断，而不是 options
APDataOptionCustomEncrypted = 1 << 1, // 这个选项不要传，传了也没效果，统一存储会使用接口里的 extension 来做加密的判断，而不是 options

// 标识该数据在清理缓存时可被清除，这里用 unsinged int 强转 1，为因为某些编译选项下，右边的 1<<31 不能按照 unsigned int 来计算，导致赋值失败
APDataOptionPurgeable = (unsigned int)1 << 31,
};
```

可以在 setObject、archiveObject、saveJsonObject 三个方法中指定 options。

APDataOptionPurgeable 指该数据可以在数据清理时自动清除，详见 数据清理 。

4.3.3 DAO 存储

文档解析出错,请参照金融科技上面的文档，文档编号为(53148),也请把括号中的文档编号发给我们查找问题,谢谢!

4.3.4 LRU 存储

根据 LRU 淘汰规则，LRU 存储提供两种存储方法：

- 内存缓存（APLRUMemoryCache）：提供内存 LRU 淘汰算法的缓存，缓存 id 对象。APLRUMemoryCache 是线程安全的，同时 LRU 算法基于链表实现，效率较高。
- 磁盘缓存（APLRUDiskCache）：提供持久化到数据库的 LRU 淘汰算法缓存，缓存支持 NSCoding 的对象。使用数据库相比文件会更容易维护，也使磁盘更整洁。

内存缓存

@property (nonatomic, assign) BOOL handleMemoryWarning; // default NO

设置是否处理系统内存警告，默认为 NO。如果设置为 YES，当有内存警告时会清空缓存。

(id)initWithCapacity:(NSInteger)capacity;

初始化，指定容量。

(void)setObject:(id)object forKey:(NSString*)key;

将对象存入缓存，如果 object 为 nil，会删除对象。

(void)setObject:(id)object forKey:(NSString*)key expire:(NSTimeInterval)expire;

将对象存入缓存，并指定一个过期时间戳。

(id)objectForKey:(NSString*)key;

取对象。

(void)removeObjectForKey:(NSString*)key;

删除对象。

(void)removeAllObjects;

删除所有对象。

(void)addObjects:(NSDictionary*)objects;

批量添加数据，无法单独设置每个对象的 expire 时间，默认都是永不过期的对象。

(void)removeObjectsWithRegex:(NSString*)regex;

批量删除数据，数据的 key 匹配 regex 的正则表达式。

(void)removeObjectsWithPrefix:(NSString*)prefix;

批量删除具有某个前缀的所有数据。

(void)removeObjectsWithSuffix:(NSString*)suffix;

批量删除具有某个后缀的所有数据。

(void)removeObjectsWithKeys:(NSSet*)keys;

批量删除所有 keys 指定的数据。

(NSArray*)peekObjects:(NSInteger)count fromHead:(BOOL)fromHead;

将缓存对象读取到一个数组里，但不做 LRU 缓存策略处理。fromHead 为 YES 时，从头开始遍历，否则对尾开始遍历。

(BOOL)objectExistsForKey:(NSString*)key;

快速判断某个 key 的对象是否存在，不会影响 LRU。

(void)resetCapacity:(NSInteger)capacity;

更新容量，如果新容量比原先的小，会删除部分缓存。

磁盘缓存

(id)initWithName:(NSString*)name capacity:(NSInteger)capacity userDependent:(BOOL)userDependent crypted:(BOOL)crypted;

创建一个持久化的 LRU 缓存，存入的对象需要支持 NSCoding 协议。

参数

参数	说明
name	缓存的名字，用做数据库的表名。
capacity	容量，实际容量会比这个大一些，解决缓存满时添加数据的性能问题。
userDependent	是否与用户相关，如果与用户相关，当 APDataCenter.currentUserId 为空时缓存无法操作；当切换用户后，缓存会自动指向当前用户的表，业务无须关心这个事件。
crypted	数据是否加密。

返回值

缓存实例，业务需要持有。

(void)setObject:(id)object forKey:(NSString*)key;

缓存一个对象，expire 默认为 0，也就是永不过期。

(void)setObject:(id)object forKey:(NSString*)key expire:(NSTimeInterval)expire;

缓存一个对象，并指定过期的时间戳。

参数

参数	说明
object	对象，如果为 nil 会删除指定 key 的对象。
key	key
expire	过期时间戳，指定一个相对于 1970 的绝对时间戳。可以使用[date timeIntervalSince1970]。

(id)objectForKey:(NSString*)key;

取对象，如果对象读取出来时，指定的 expire 时间戳已经达到，会返回 nil，并删除对象。如果调用 objectForKey 前，有其它 setObject 操作写数据库未完成，会等待。

(void)removeObjectForKey:(NSString*)key;

删除对象。

(void)removeAllObjects;

删除所有对象。

(void)addObjects:(NSDictionary*)objects;

批量添加数据。

(void)removeObjectsWithSqlLike:(NSString*)like;

批量删除数据，数据的 key 使用 sqlite 的 like 语句匹配。

(void)removeObjectsWithKeys:(NSSet*)keys;

批量删除所有 keys 指定的数据。

4.3.5 自定义存储

APDataCenter 对应的默认存储空间为应用沙箱的 /Documents/Preferences 目录。若业务比较独立或数据量比较多，可以自定义存储空间。统一存储模块提供三种自定义存储方式：

- APCustomStorage 存储
- APAsyncFileArrayService 存储
- APObjectArrayService 存储

APCustomStorage 存储

您可以使用 APCustomStorage 创建一个自己的存储目录。在这个目录里，您可以使用统一存储提供的所有服务，类似 APDataCenter。比如：

```
APCustomStorage* storage = [APCustomStorage storageInDocumentsWithName:@"Contact"];
```

就会创建 Documents/Contact 目录。这个目录里同样有存储公共数据的commonPreferences和与用户相关数据的userPreferences。APCustomStorage 与 APDataCenter 类似，业务同样无须关注用户切换。

接口说明

(instancetype)storageInDocumentsWithName:(NSString*)name;

创建路径为 /Documents/name 的自定义存储

(id)initWithPath:(NSString*)path;

在任意指定路径创建自定义存储，通常不需要使用这个方法，使用 storageInDocumentsWithName 即可。使用此接口创建的 APCustomStorage，业务需要自己持有，并且当多个 APCustomStorage 的 path 相同时，会出错。

(APBusinessPreferences*)commonPreferences;

与用户无关的全局存储对象，使用 key-value 方式存取数据。与 APDataCenter 的区别是：在业务的自定义存储空间里，存储 key-value 数据时不需要 business 参数，只需要 key 即可。

(APBusinessPreferences*)userPreferences;

当前登录用户的存储对象，使用 key-value 方式存取数据。不是登录态时，取到的是 nil。与 APDataCenter 的区别是：在业务的自定义存储空间里，存储 key-value 数据时不需要 business 参数，只需要 key 即可。

(id)daoWithPath:(NSString*)filePath userDependent:(BOOL)userDependent;

参考 APDataCenter 的同名接口

(APAsyncFileArrayService*)asyncFileArrayServiceWithName:(NSString*)*name userDependent:(BOOL)userDependent capacity:(NSInteger)capacity crypted:(BOOL)crypted;

创建一个异步的文件阵列管理服务，用于存储互相类似的多条文件记录。会根据服务名单独在数据库文件里建表存储，不放在 key-value 存储的表里。

@param name 服务名，不能为空
@param userDependent 是否与用户相关
@param capacity 容量，超过 capacity 条数据后，会自动清除最早的。capacity <= 0 表示不做数量限制。不是字节容量，是条数。
@param crypted 文件内容是否加密

@return 返回 service 对象，业务自行持有

- **(APObjectArrayService*)objectArrayServiceWithName:(NSString*)*name userDependent:(BOOL)userDependent capacity:(NSInteger)capacity cacheCapacity:(NSInteger)cacheCapacity crypted:(BOOL)crypted;**

创建一个 id 对象存储的阵列管理服务，用于存储类似的 id 对象，支持内存缓存与数据加密。

@param name 服务名，不能为空
@param userDependent 是否与用户相关
@param capacity 容量，超过 capacity 条数据后，会自动清除最早的。capacity <= 0 表示不做数量限制。不是字节容量

，是条数。
@param cacheCapacity 内存缓存的条目数容量，cacheCapacity <= 0 时不使用内存缓存。
@param crypted id 对象是否加密

@return 返回 service 对象，业务需要自行持有

APAsyncFileArrayService 存储

- APAsyncFileArrayService 这个服务是利用数据库提供文件阵列存储功能。当有大量类似文件（比如聊天里的语音）需要存储时，可以使用这个服务。
- 这个服务需要使用 APCustomStorage 来创建。您需要先创建 APCustomStorage，指定自己的工作目录，再用 APCustomStorage 创建一个 APAsyncFileArrayService，然后使用这个 service 就可以在自己的目录内的数据库文件里写入文件阵列。

接口说明

对于传入了 completion 的异步接口，completion 一定会在主线程回调，可为 nil。

(void)writeFile:(NSData*)*data name:(NSString*)name completion:(void(\^)(BOOL result))completion;

写入文件，文件名是 name，当异步写入完成时，会回调 completion。result 返回是否成功。

(void)readFile:(NSString*)*name completion:(void(\^)(NSData* data))completion;

异步读取文件，文件名是 name，完成时会将文件数据在 completion 返回。

(NSData*)*readFileSync:(NSString*)name;

同步读文件

(void)readFilesLike:(NSString*)*pattern completion:(void(\^)(NSDictionary* result))completion;

使用 sqlite 的 like 功能读取批量数据，result 为文件名与文件数据的字典，异步接口。

(void)renameFile:(NSString*)*name newName:(NSString*)newName completion:(void(\^)(BOOL result))completion;

异步重命名文件

(void)removeFile:(NSString*)name;

异步删除文件

(void)removeFilesLike:(NSString*)pattern;

使用 sqlite 的 like 功能批量删除数据

(void)removeAllFiles;

异步删除所有文件

(BOOL)fileExists:(NSString*)name;

同步接口，文件是否存在

(NSArray*)allFileNames;

同步接口，取所有文件名

(NSInteger)fileCount;

同步接口，文件数目

APObjectArrayService 存储

- APObjectArrayService 基于 APAsyncFileArrayService 实现，存储支持 NSCoding 协议的 Objective-C 对象，相比 APAsyncFileArrayService，这个服务提供了内存缓存功能。
- 这个服务同样需要使用 APCustomStorage 来创建。您需要先创建 APCustomStorage，指定自己的工作目录，再用 APCustomStorage 创建一个 APObjectArrayService，然后使用这个 service 就可以在自己的目录内的数据库文件里写入对象阵列。

接口说明

(void)setObject:(id)object forKey:(NSString*)key; // 写 IO 是异步的

设置对象

(id)objectForKey:(NSString*)key; // 如果缓存里没有，同步去数据库中读取

取对象

(void)objectForKey:(NSString*)key completion:(void(^)(id object))completion;

异步读取，读取成功后在主线程调用 completion 返回。

(void)objectsForKeyLike:(NSString*)pattern completion:(void(^)(NSDictionary result))completion;

使用 sqlite 的 like 功能读取批量数据，data 为对象名与对象的字典。不检查缓存。

(void)removeObjectForKey:(NSString*)key;

删除数据

(void)removeObjectsForKeyLike:(NSString*)pattern;

使用 sqlite 的 like 功能批量删除数据

(void)removeAllObjects;

删除所有数据

(BOOL)objectExistsForKey:(NSString*)key;

判断某键值数据是否存在

(NSArray*)allKeys;

缓存中所有的 Key

(NSInteger)objectCount;

缓存对象数目

4.3.6 数据清理

自动清理的缓存目录

创建一个可以自动维护容量的缓存目录，通过 APPurgeableType 指定清空的逻辑，通过 size 指定缓存目录的大小。应用每次启动会在后台进程检查目录状态，并按需求删除文件。如果一个目录设置了容量上限，当达到上限时，会删除其中创建时间最早的文件，使目录恢复到 1/2 容量上限的使用情况。

```
#import <Foundation/Foundation.h>

typedef NS_ENUM(NSUInteger, APPurgeableType)
{
APPurgeableTypeManual = 0, // 当用户手动清除缓存时清空
APPurgeableTypeThreeDays = 3, // 自动删除三天前的数据
APPurgeableTypeOneWeek = 7, // 自动删除一周前的数据
APPurgeableTypeTwoWeeks = 14, // 自动删除两周前的数据
APPurgeableTypeOneMonth = 30, // 自动删除一个月前的数据
};

#ifdef __cplusplus
extern"C"{
#endif // __cplusplus

/**
* 根据用户的输入返回一个可被清理的存储路径，同时会自动判断目录是否存在，如果不存在会创建。
*
* @param userPath 用户指定的路径，比如之前使用"Documents/SomePath"来拼接，现在使用
APPurgeableStoragePath(@"Documents/SomePath")获得路径即可。
* @param type 指定自动清空的类型，可以是用户手动或每周，或每三天。
* @param size 指定当尺寸达到多大时，清空较老的数据，单位MB。0表示不设置上限。
*
* @return 目标路径
*/
NSString* APPurgeablePath(NSString* path);
NSString* APPurgeablePathType(NSString* path, APPurgeableType type);
NSString* APPurgeablePathTypeSize(NSString* path, APPurgeableType type, NSUInteger size /* MB */);

/**
* 清空并重置所有注册的目录
*/
void ResetAllPurgeablePaths();

#ifdef __cplusplus
}
#endif // __cplusplus
```

缓存清理接口

统一存储提供清理缓存的实现类，这个类从 PurgeableCache.plist 中读取清理任务，这个文件需要放在应用的 Main Bundle 里。清理器会异步执行。回调函数总会在主线程调用，可以在里面进行 UI 展示与处理。

```
#import <Foundation/Foundation.h>

typedef NS_ENUM(NSUInteger, APCacheCleanPhase)
{
APCacheCleanPhasePreCalculating = 0, // 执行前扫描沙箱大小
APCacheCleanPhaseCleaning, // 正在清理
APCacheCleanPhasePostCalculating, // 执行完成扫描沙箱大小
APCacheCleanPhaseDone, // 完成
};
```

```
@interface APUserCacheCleaner : NSObject

/**
* 异步执行清理。必须传一个回调方法。
* 当phase返回
APCacheCleanPhasePreCalculating，APCacheCleanPhaseCleaning，APCacheCleanPhasePostCalculating时
，progress代表真实的进度。最大为1.0。
* 当phase为APCacheCleanPhaseDone时，progress返回清理了多少MB的数据。
*
* @param callback 回调方法
*/
+ (void)execute:(void(^)(APCacheCleanPhase phase, float progress))callback;

@end
```

在 PurgeableCache.plist 中可以定义两种类型的清理任务。

▼ Root	Array	(4 items)
▼ Item 0	Dictionary	(2 items)
Class	String	APUserCacheCleaner
▼ Selectors	Array	(2 items)
Item 0	String	cleanDefaultPreferences
Item 1	String	cleanPurgeablePaths
▶ Item 1	Dictionary	(2 items)
▶ Item 2	Dictionary	(2 items)
▼ Item 3	Dictionary	(2 items)
Path	String	Library/Caches
▼ Entries	Array	(10 items)
Item 0	String	*.localstorage
Item 1	String	alumonitorlog
Item 2	String	TBSDKNetworkSDK_Cache*
Item 3	String	AutoNaviMapKitCache
Item 4	String	com.alipay.downloads
Item 5	String	com.alipay.iphoneclient
Item 6	String	LogFiles
Item 7	String	*.cache
Item 8	String	*.txt
Item 9	String	file

Path 文件或目录的路径，只需要沙箱内的相对路径即可。

Entries 当 Path 为一个目录时，删除它下面的哪些子文件或目录，支持 * 进行通配。

Class 指定一个回调方法的定义类。

Selectors 调用 Class 类的哪些类方法。注意，必须为类方法，不能为实例方法。

5 常见问题

iOS 常见问题

如何设置统一存储用户态？

解答：接入 mPaaS 的应用会使用自己的账号体系，如果需要使用统一存储来管理用户态数据，请第一时间通知统一存储，让统一存储进行用户数据库的切换，再通知其它业务层。

```
[[APDataCenter defaultDataCenter] setCurrentUserId:userId];
```

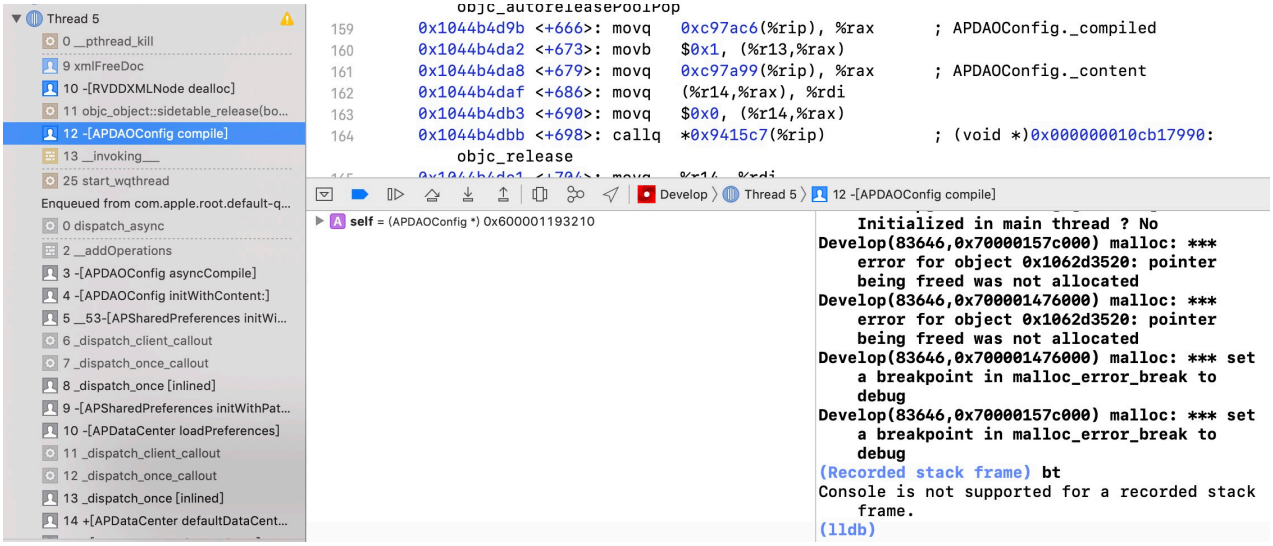
当用户登出时，可以不调用 setCurrentUserId 方法，统一存储会继续打开上一个用户的数据库，不会产生影响。

统一存储是线程安全的吗？

解答：是的，统一存储的数据存储接口都考虑了线程安全性问题，可以在任意线程进行调用。

如何解决与百度地图 SDK 的冲突？

描述：当与某一版本的百度地图 SDK 集成时，可能出现如下 crash。



解答：您需要在 App 初始化时进行如下设置（10.1.32 及以上版本支持）。

```
#import <MPDataCenter/APDataCenter.h>
// App 初始化方法中设置
APDataCenter.compatibility = YES;
```

archiveObject 是如何存储和读取变量的？

解答：请参考以下代码：

- 对象持久化存储：

```
MPCodingData *obj = [MPCodingData new];
obj.name = @"Amelia"; :
obj.age = 1;
[APUserPreferences archiveObject:obj forKey:@"archObjKey"business:dataBusiness];
```

统一存储中- 读取变量：

```
MPCodingData *encodeObj = [APUserPreferences objectForKey:@"archObjKey"business:dataBusiness];
```


蚂蚁集团
ANT GROUP