蚂蚁科技

统一存储
使用指南

文档版本：20231226

蚂蚁集团
ANT GROUP

法律声明

商标声明

蚂蚁集团 ANT GROUP及其他蚂蚁集团相关的商标均为蚂蚁集团所有。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。蚂蚁集团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，并在蚂蚁集团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集团授权渠道下载、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。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	**单击设置> 网络> 设置网络类型。**
粗体	表示按键、菜单、页面名称等UI元素。	**在结果确认页面，单击确定。**
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.统一存储

1.1. 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。

1.2. 接入 Android

1.2.1. 快速开始

统一存储支持 **原生 AAR 接入** 和 **组件化接入** 两种接入方式。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。

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 方式添加 SDK，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，参见 数据库存储。
- 如需使用键值对存储相关内容，参见 键值对存储。
- 如需使用文件存储相关内容，参见 文件存储。

代码示例

单击此处，获取示例代码。

1.2.2. 进阶指南

数据库存储

关于数据库存储的使用方法，请参考 数据库存储。

键值对存储

关于键值对存储的使用方法，请参考 键值对存储。

文件存储

关于文件存储的使用方法，请参考 文件存储。

1.3. 接入 iOS

本文介绍如何将统一存储组件接入到 iOS 客户端。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。

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。
  i. 点击 Xcode 菜单项 **Editor** > **mPaaS** > **编辑工程**，打开编辑工程页面。
  ii. 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。
  i. 在 Podfile 文件中，使用 `mPaaS_pod "mPaaS_DataCenter"` 添加统一存储组件依赖。
  ii. 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。

1.4. 存储类型

1.4.1. 存储类型简介

接入 Android 客户端的统一存储组件提供以下持久化存储方案：

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。

类名	功能
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。

1.4.2. Android 存储类型

1.4.2.1. 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、查可以使用以下接口来调用：

- 10.2.3 及以上基线：`com.alibaba.j256.ormlite.dao.Dao`
- 10.1.68 及以下基线：`com.j256.ormlite.dao.Dao`

> **说明**
>
> 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。

- 10.2.3 及以上基线：

```
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(MPSQLiteDatabase 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(MPSQLiteDatabase 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();
        }
    }
}
```

- 10.1.68 及以下基线：

```
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).que
ry());
        } 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();
        }
    }
```

1.4.2.2. 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();
}
```

1.4.2.3. 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();
        }
    }
```

1.4.3. iOS 存储类型

1.4.3.1. 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。

1.4.3.2. 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:(NS
Integer)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:(l
ong 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:(APData
Crypt*)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 extensi
on:(APDataCrypt*)extension;
- (BOOL)setObject:(id)object forKey:(NSString*)key business:(NSString*)business extensi
on:(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 ext
ension:(APDataCrypt*)extension;
- (BOOL)archiveObject:(id)object forKey:(NSString*)key business:(NSString*)business ext
ension:(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 ex
tension:(APDataCrypt*)extension;
- (BOOL)saveJsonObject:(id)object forKey:(NSString*)key business:(NSString*)business ex
tension:(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

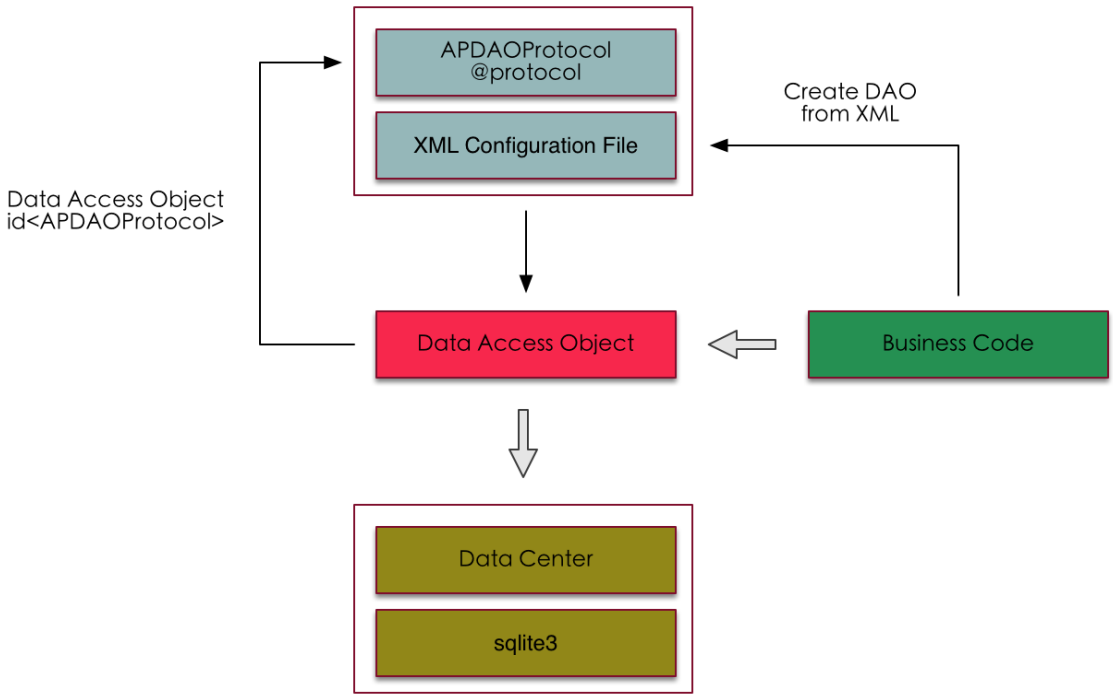
```
typedef NS_OPTIONS (unsigned int, APDataOptions)
{
    // 这两个选项不要在接口中使用，是标识数据的加密属性的，请使用 extension 来传递加密方法
    APDataOptionDefaultEncrypted     = 1 << 0,        // 这个选项不要传，传了也没效果，统一存储会使用接口里的 extension 来做加密的判断，而不是 options
    APDataOptionCustomEncrypted      = 1 << 1,        // 这个选项不要传，传了也没效果，统一存储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

1.4.3.3. DAO 存储

普通 KV 存储只能存储简单数据类型或封装好的 OC 对象，且不支持搜索。当业务有 sqlite 访问需求时，可由统一存储的 DAO 功能进行简化和封装。基本工作原理如下：



1. 定义一个 `xml` 格式的配置文件，描述各 sqlite 操作的函数、返回的数据类型、需要加密的字段等。
2. 定义一个 DAO 对象的接口 `DAOInterface（@protocol）` ，接口方法名、参数与配置文件里的描述一致。
3. 业务将 `xml` 配置文件传给 `APDataCenter` 的 `daoWithPath` 方法，生成 DAO 访问对象。该对象直接强转为 `id<DAOInterface>` 。
4. 接下来业务就可以直接调用 DAO 对象的方法，统一存储会将该方法转换为配置文件里描述的数据库操作。

示例

- 配置文件第一行定义了默认表名和数据库版本，以及初始化方法。
- `insertItem` 和 `getItem` 是插入和读取数据的两个方法，接收参数并格式化到 SQL 表达式里。
- `createTable` 方法会在底层被默认调用一次。

```
<module name="Demo" initializer="createTable" tableName="demoTable" version="1.0">
<update id="createTable">
  create table if not exists ${T} (index integer primary key, content text)
</update>

<insert id="insertItem" arguments="content">
  insert into ${T} (content) values(#{content})
</insert>

<select id="getItem" arguments="index" result="string">
  select * from ${T} where index = #{index}
</select>
</module>
```

- DAO 接口定义：

```
@protocol DemoProtocol <APDAOProtocol>
- (APDAOResult*)insertItem:(NSString*)content;
- (NSString*)getItem:(NSNumber*)index;
@end
```

- 创建 DAO 代理对象，假设配置文件名为 `demo_config.xml`，在 Main Bundle 中。
- 用 `insertItem` 方法写入一个数据，获取它的索引，再用该索引把写入的数据读出来。

```
NSString* filePath = [[NSBundle mainBundle] pathForResource:@"demo_config" ofType:@"x
ml"];
id<DemoProtocol> proxy = [[APDataCenter defaultDataCenter] daoWithPath:filePath userD
ependent:YES];
[proxy insertItem:@"something"];
long long lastIndex = [proxy lastInsertRowId];
NSString* content = [proxy getItem:[NSNumber numberWithInteger:lastIndex]];
NSLog(@"content %@", content);
```

- 其中 `lastInsertRowId` 是 `APDAOProtocol` 的一个方法，用来取最后插入的行的 `rowId`。想让 DAO 对象支持该方法，只要在声明 `DemoProtocol` 时继承自 `APDAOProtocol` 即可。

关键字

module

```
<module name="demo" initializer="createTable" tableName="tableDemo" version="1.5" reset
OnUpgrade="true" upgradeMinVersion="1.2">
```

- `initializer` 参数可选。对于 `initializer` 指定的 `update` 方法，DAO 认为是数据库建表方法，会在第一次 DAO 请求时默认执行一次。

- `tableName` 指定下面方法里默认操作的表名，在 SQL 语句里可以用 `${T}或${t}` 代替，不用每次都写表名。建议每个配置文件只操作一张表。`tableName` 可空，应对同一配置文件操作相同格式的多张表的情况。比如聊天消息分表处理，可以调用 DAO 对象的 `setTableName` 方法设置需要操作的表名。
- `version` 是配置文件的版本号，格式为 `x.x`，创建 table 后，会以 `tableName` 作为 key，把表的版本存到数据库文件的 `TableVersions` 表里，配合 `upgrade` 块进行表的更新。
- `resetOnUpgrade`，如果为 TRUE 或 YES，当 `version` 更新后，会删除原表，而不是调用 `ungrade` 块。无此参数为默认为 false。
- `upgradeMinVersion`，如果不为空，对于小于这个版本的数据库文件，直接重置，否则执行升级操作。

const

```
<const table_columns="(id, time, content, uin, read)"/>
```

- 定义一个字符串类型的常量，`table_columns` 是常量的名字，等号（=）后方为常量值，在配置文件里可以使用 `${常量名}` 来引用。

select

```
<select id="find" arguments="id, time" result=":messageModelMap">
  select * from ${T} where id = #{id} and time > @{time}
</select>
```

`@arguments`：

- 参数名的列表，用 `,` 分隔。调用者传进来的参数依赖 `arguments` 里的描述依次命名。DAO 对象的 selector 调用时不会携带参数名，所以必须在这里按顺序命名。
- 参数名前面有 `$` 符号时，表示这个参数不接受 nil 值。业务的调用 DAO 接口，是允许传 nil 参数的，如果某个参数前面有 `$` 符号，当调用者不小心传入了 nil 值，DAO 调用会自动失败。防止发生不可预知的问题。
- 关于如何引用参数，参见下文中的 引用方式。

例如，在上方的代码中，对应的 selector 为：

```
- (MessageModel*)find:(NSNumber*)id time:(NSNumber*)time;
```

如果 DAO 对象调用 `[daoProxy find:@1234 time:@2014]`，那么拼接好后的 SQL 语句是：

```
select * from tableDemo where id = ? and time > 2014
```

并且 `@1234` 这个 `NSNumber` 会交给 sqlite 绑定。

`@result`：

`result` 里可以填写 DAO 方法的返回值，用 `[]` 括起来表示返回数组类型，会对 select 执行的返回一直进行迭代，直到 sqlite 无结果返回。如果不用 `[]` 括起来，表示只返回一个结果，对 select 执行的返回只迭代一次，类似 FMDB 库里 `FMResultSet` 只调用一次 next 方法。

返回类型：

- `int`：只有一个结果，返回 `[NSNumber numberWithInt]` 类型，注意溢出的可能。

- `long long` ：只有一个结果，返回 `[NSNumber numberWithLongLong]` 类型。
- `bool` ：只有一个结果，返回 `[NSNumber numberWithBool]` 类型。
- `double` ：只有一个结果，返回 `[NSNumber numberWithDouble]` 类型。
- `string` ：只有一个结果，返回 `NSString*` 类型。
- `binary` ：只有一个结果，返回 `NSData*` 类型。
- `[int]` ：返回数组，数组里为 `[NSNumber numberWithInt]` 。
- `[long long]` ：返回数组，数组里为 `[NSNumber numberWithLongLong]` 。
- `[bool]` ：返回数组，数组里为 `[NSNumber numberWithBool]` 。
- `[double]` ：返回数组，数组里为 `[NSNumber numberWithDouble]` 。
- `[string]` ：返回数组，数组里为 `NSString*` 。
- `[binary]` ：返回数组，数组里为 `NSData*` 。
- `[{}]` ：返回数组，数组里是列名 > 列值的 map。
- `[AType]` ：返回数组，数组里是填好的自定义类。
- `{}` ：只有一个结果，返回列名 > 列值的 map。
- `AType` ：只有一个结果，返回填好的自定义类。
- `[:AMap]` ：返回数组，数组里是使用 `xml` 里定义的 AMap 映射出来的对象。
- `:AMap` ：只有一个结果，使用配置文件里定义的 AMap 来描述对象。

例如，上面的例子中，返回类型为 `:messageModelMap` 。具体返回的 `Objective-C` 类型，以及需要特殊映射的列都会在 `messageModelMap` 里定义。参考 map 关键字。

`@foreach` ：select 也支持 `foreach` 字段，用法下文介绍的 `insert` 、 `update` 、 `delete` 里的相似。不同的是， `select` 方法如果指定了 `foreach` 参数，那么会执行 N 次 SQL 的 select 操作，并把结果放到数组里返回。所以如果 DAO 的 `select` 方法是 `foreach` 的，它的返回值在 protocol 里一定要定义成 `NSArray*` 。

insert/update/delete

```
<insert id="addMessages" arguments="messages" foreach="messages.model">
    insert or replace into ${T} (id, content, uin, read) values(#{model.msgId}, #{model
.content}, #{model.uin}, #{model.read})
</insert>

<update id="createTable">
  <step>
    create table if not exists ${T} (id integer primary key, content text, uin integer,
read boolean)
  </step>
  <step>
    create index if not exists uin_idx on ${T} (uin)
  </step>
</update>

<delete id="remove" arguments="msgId">
  delete from ${T} where msgId = #{msgId}
</delete>
```

- insert、update、delete 方法格式相同，参数的拼接与引用和 select 方法相同。
- insert 和 delete 关键字是为了更好的区分方法用途。你完全可以把一个 delete from table 的操作写在 update 函数中。
- 在 DAO 接口中，需要执行 insert、update 或 delete 的方法，返回值为 APDAOResult*，标识执行是否成功。

`@foreach` ：

- 当 insert、update、delete 方法后面有 `foreach` 字段时，这个方法被调用时，会依次对参数数组里的每个元素执行一次 SQL。
- 要求 foreach 字段遵循格式： collectionName.item。其中 collectionName 必须对应 arguments 里的一个参数，指定循环的容器对象，并且调用时必须为一个 NSArray 或 NSSet 类型； item 表示从容器里取出的对象，用作循环变量。不能和 arguments 里的任何参数重名，这个 item 可以当作一个普通参数用在 SQL 语句里。

例如代理方法为：

```
- (void)addMessages:(NSArray*)messages;
```

messages 为 MessageModel 数组，那么对于 messages 里的每个 model，都会执行一次 SQL 调用，这样就能实现把数组里的元素一次性插入数据库而上层不需要关心循环的调用。底层会把这次操作合并为一个事务而提升效率。

step

```
<upgrade toVersion='3.2'>
    <step resumable="true">alter table ${T} add column1 text</step>
    <step resumable="true">alter table ${T} add column2 text</step>
</upgrade>
```

- 在 insert, update, delete, upgrade 方法里，可能遇到这种情况：执行一个 DAO 方法，但是需要调用多次 SQL update 操作执行多条 SQL 语句。比如用户建表后，又要创建一些索引。在函数里

  包裹的语句，都会当作一次 SQL update 被单独执行，底层会把所有操作合并为一次事务。比如上面的 createTable 方法。
- 如果一个函数里面有 step 了，那么在 step 外面不允许有文本了。step 内不能再含其它 step。

`@resumable` ：

- 当这条 step 语句生成的 SQL 执行失败时，是否继续执行下面的语句。无此参数默认为 false。也就是顺序执行 step，当发生失败的情况，整个 DAO 方法认为失败。

map

```
<map id="messageModelMap" result="MessageModel">
  <result property="msgId" column="id"/>
</map>
```

- 定义一个名为 `messageModelMap` 的映射，实际生成的 Objective-C 对象是 MessageModel 类。
- Objective-C 对象的 msgId 属性，对应表里名为 id 的列值；未列的属性认为和表的列名相同，可以省略。

upgrade

```
<upgrade toVersion="1.1">
  <step>
    alter table...
  </step>
  <step>
    alter table...
  </step>
</upgrade>
```

- 随着版本升级，数据库可能有升级需求；处理升级的 SQL 语句写在这里。比如最初，配置文件 module 的版本是 1.0，升级后，配置文件的版本为 1.1。那么新版本第一次运行 DAO 方法时，会检测当前表的版本与配置文件的版本，当发现不一致时，会依次执行升级方法。这个方法会由底层自动调用，并在升级完成后再执行 DAO 方法。
- upgrade 按照 SQL update 来执行，如果 SQL 不止一个，可以用 <step> 括起来，类似 <update id="createTable"> 的实现。
- 如果需要使用 upgrade 块定义的操作来升级表，那么 module 里的 resetOnUpgrade 必须设置为 false。

crypt

```
<crypt class="MessageModel" property="content"/>
```

描述 class 这个类的 property 属性进行加密处理，当向数据库写入时从这个属性里取出的值会进行加密；当数据库读出时，生成对象向这个属性里赋值会先解密再赋值。

比如执行 DAO 方法，model 是 MessageModel 类。因为取了 model 的 content 属性，所以会加密后再写入数据库。

```
<insert id="insertMessage" arguments="model">
     insert into ${T} (content) values(#{model.content})
</insert>
```

执行这个 select 方法时，返回对象是 MessageModel 类。底层从数据库里取出数据向 MessageModel 的实例写入 content 属性时，会将数据先解密再写入。最后返回处理好的 MessageModel 对象。

```
<select id="getMessage" arguments="msgId" result="MessageModel">
     select * from ${T} where msgId = #{msgId}
</select>
```

加密方式的设置方法定义在 APDAOProtocol 中，如下：

```
/**
 *  设置加密器，用于加密表里标记为需要加密的列的数据。向表里写数据时，碰到这个列，会调用进行加密。
 *
 *  @param crypt 加密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，需要外部进行 free。如果是
APDefaultEncrypt()，不需要进行释放。
 */
- (void)setEncryptMethod:(APDataCrypt*)crypt;

/**
 *  设置解密器，用于解密表里标记为需要加密的列的数据。从表里读数据时，碰到这个列，会调用进行解密。
 *
 *  @param crypt 解密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，需要外部进行 free。如果是
APDefaultDecrypt()，不需要进行释放。
 */
- (void)setDecryptMethod:(APDataCrypt*)crypt;
```

如果不进行设置，会使用 APDataCenter 的默认加密，见 KV 存储。如果一个 DAO 代理对象是 `id<DAOProtocol>` ，并且 `DAOProtocol` 是 `@protocol<APDAOProtocol>` ，那么可以直接用 DAO 代理对象调用 `setEncryptMethod` 和 `setDecryptMethod` 来设置加密、解密方法。

if

```
<insert id="addMessages" arguments="messages, onlyRead" foreach="messages.model">
    <if true="model.read or (not onlyRead)">
    insert or replace into ${T} (msgId, content, read) values(#{model.msgId}, #{model.c
ontent}, #{model.read})
    </if>
</insert>
```

- 在 insert、update、delete、select 方法里，可以嵌套使用 if 条件判断语句。当 if 条件满足时，会把 if 块内的文本拼接到最终的 SQL 语句里。
- if 后可以接 true="expr"，也可以接 false="expr"； expr 为表达式，可以使用方法的参数，并且可以使用"."来链式访问参数对象的属性。
- 表达式支持的运算符如下：

  (): 括号

  +: 正号

  -: 负号

  +: 加号

  -: 减号

  *: 乘号

  /: 除号

  \: 整除

  %: 取模

  >: 大于

  <: 小于

  >=: 大于等于

  <=: 小于等于

==：等于

!=：不等于

and：逻辑与，不区分大小写

or：逻辑或，不区分大小写

not：逻辑非，不区分大小写

xor：异或，不区分大小写

- 大于号、小于号字符需要使用转义。
- 里面的参数就是外部调用传入的参数名，但是不要像 SQL 块里使用 #{} 或 @{} 包裹。
- nil 含义同 Objective-C 里的 nil。
- 表达式里的字符串使用单引号起止，不支持转义字符，但是支持 \'代表一个单引号。
- 当参数为 Objective-C 对象时，支持用 `.` 来访问它的属性，比如上面例子 model.read，如果参数是数组或字典，可以用 `参数名.count` 取元素数。

一个较复杂的表达式如下：

```
<if true="(title != nil and year > 2010) or authors.count >= 2">
```

title，year，authors 都是调用者传来的参数，调用时 title 是可以传 nil 的；那么上面含义为"当书名不为空，并且出版年份大于 2010 年，或作者数大于 2。

choose

```
<choose>
  <when true="title != nil">
    AND title like #{title}
  </when>
  <when true="author != nil and author.name != nil">
    AND author_name like #{author.name}
  </when>
  <otherwise>
    AND featured = 1
  </otherwise>
</choose>
```

- 实现类似 switch 的语法，表达式要求类似 if 语句； when 后面也可以接 true="expr"或 false="expr"。
- 只会执行第一个符合条件的 when 或者 otherwise；可以没有 otherwise。

foreach

```
<foreach item="iterator" collection="list" open="(" separator="," close=")"
reverse="yes">
  @{iterator}
</foreach>
```

- open、separator、close、reverse 可以省略。
- item 表示循环变量，collection 表示循环数组参数名。

比如一个方法从外部接收字符串数组参数，list 内容为 `@[@"1", @"2", @"3"]` ，有另一个参数是 `prefix=@"abc"` ，使用 `()` 包裹，使用 `,` 分隔。那么执行结果为： `(abc1,abc2,abc3)` 。

```
<update id="proc" arguments="list, prefix">
   <foreach item="iterator" collection="list" open="(" separator="," close=")">
       {prefix}{iterator}
   </foreach>
</update>
```

foreach 语句通常用于拼接 select 语句里的 in 块，比如：

```
select * from ${T} where id in
<foreach item="id" collection="idList" open="(" separator="," close=")">
#{id}
</foreach>
```

where/set/trim

```
<where onVoid="quit">
  <if true="state != nil">
    state = #{state}
  </if>
  <if true="title != nil">
    AND title like #{title}
  </if>
  <if true="author != nil and author.name != nil">
    AND author_name like #{author.name}
  </if>
</where>
```

where 会处理多余的 AND、OR（大小写无所谓），并在任何条件都不符合时连 where 都不返回。用于在 SQL 语句里拼接有大量判断条件的 where 子句。比如上例，如果只有最后一个判断成立，该语句会正确返回 where author_name like XXX，而不是 where AND author_name like XXX。

```
<set>
  <if false="username != nil">username=#{username},</if>
  <if false="password != nil">password=#{password},</if>
  <if true="email != nil">email=#{email},</if>
  <if true="bio != nil">bio=#{bio},</if>
</set>
```

set 会处理结尾多余的 `,` ，并在任何条件都不符合时什么都不返回。与 where 语句类似，只是它处理的是后缀的逗号。

```
<trim prefix="WHERE" prefixOverrides="AND | OR | and | or " onVoid="ignoreAfter">
</trim>
<!--
  等价于<where>
-->

<trim prefix="SET" suffixOverrides=",">
</trim>
<!--
  等价于<set>
-->
```

- where 和 set 语句可以使用 trim 替换。Trim 语句定义了语句的整体前缀，以及对每个子句需要处理的多余的前缀与后缀列表（用 |划分）。
- onVoid 参数可以出现在 where、set、trim 里，有两个取值 ignoreAfter 和 quit。分别代表当这个 trim 语句里任何子句都不成立，导致生成一个空串时，采取什么逻辑。ignoreAfter 代码忽略下面的格式化语句，直接返回当前生成的 SQL 语句执行，quit 代表不再执行这条 SQL 语句，但会返回成功。

sql

```
<sql id="userColumns"> id,username,password </sql>
<select id="selectUsers" result="{}">
  select ${userColumns}
  from some_table
  where id = #{id}
</select>
```

定义可重用的 SQL 代码段，在其它语句中使用 ${name} 从原文引入进来。name 不能为 T 或 t，因为 ${T} 和 ${t} 代表默认的表名了。SQL 块里面可以再引用别的 SQL 块。

try except

```
<insert id="insertTemplates" arguments="templates" foreach="templates.model">
    <try>
        insert into ${T} (tplId, tplVersion, time, data, tag) values(%{'#{model.*}', tp
lId, tplVersion, time, data, tag})
        <except>
            update ${T} set %{'* = #{model.*}', data, tag} where tplId = #{model.tplId}
and tplVersion = #{model.tplVersion}
        </except>
    </try>
</insert>
```

有时，同一个 model 可能多次插入数据库，用 insert or replace 会导致当 model 主键冲突（同主键的 model 已经在数据库存在）时，原先的数据被删除掉，重新 insert。这样会导致同条数据的 rowid 发生变化。用 try except 语句块可以解决这个问题（当然不仅限于解决这种问题）。try except 只能出现在 DAO Method 定义里，前后不能再有其它语句。try 和 except 里面可以包含其它语句块。

当这条 DAO 方法执行时，如果 try 里面的语句执行失败，会自动尝试执行 except 里的语句。如果都失败，这次 DAO 调用才会返回失败。

引用方式

@ 引用

@{something}，用于方法参数，参数名为 something，在格式化 SQL 语句时会把对象内容拼到 SQL 语句中；因为参数都为 id 类型，所以默认使用 [NSString stirngWithFormat:@"%@", id] 来格式化；@{something or ""} 这种格式，表示传入的参数如果为 nil，会转成一个空字符串而不是 NULL。

不建议使用 @{} 的方式来引用参数，效率比较低，有 SQL 注入风险。如果参数对象是个 NSString，拼接进去后，会自动添加引号将字符串括起来，保证 SQL 语句格式的正确性。如果用户在配置文件里自己写了引号，底层不会自动添加引号了。

使用 @{something no ""}，这种格式，可以强制不添加引号。

```
<select id="getMessage" arguments="id" result="[MessageModel]">
    select * from ${T} where id = @{id}
</select>
```

比如上例，id 参数传进来是个 NSString，上面的写法是正确的，生成的 SQL 会自动把 id 格式化进去，并且在前后添加引号。

引用

#{something}，用于方法参数，参数名为 something，在格式化 SQL 语句时会转为一个 `?` ，然后将对象绑定给 sqlite；建议书写 SQL 时尽量使用这种方式，效率更高。 #{something or “”} 这种格式，表示传入的参数如果为 nil，会转成一个空字符串而不是 NULL。

$ 引用

${something}，用来引用配置文件里的内容，比如引用默认表名 ${T} 或 ${t}、引用配置文件里定义的常量和 SQL 代码块。

链式访问

对于 @ 和 # 引用，可以使用 `.` 来访问参数对象的属性。比如传入的参数名是 model，并且是一个 MessageModel 类型，它有属性 NSString* content。那么 @{model.content}，会取出其 content 属性的值。内部实现为 [NSObject valueForKey:]，所以如果参数是一个字典（字典的 valueForKey 等价于 dict[@""]），那么也可以使用 #{adict.aaa} 引用 adict[@"aaa"] 值。

代理方法

每个生成的 DAO 对象代理对象都支持 APDAOProtocol。

```
@protocol MyDAOOperations <APDAOProtocol>
- (APDAOResult*)insertMessage:(MyMessageModel*)model;
@end
```

具体方法见代码的函数注释。

```
#import <Foundation/Foundation.h>
#import <sqlite3.h>
#import "APDataCrypt.h"
#import "APDAOResult.h"
#import "APDAOTransaction.h"

@protocol APDAOProtocol;

typedef NS_ENUM (NSUInteger, APDAOProxyEventType)
{
    APDAOProxyEventShouldUpgrade = 0,    // 即将升级
    APDAOProxyEventUpgradeFailed,        // 表升级失败
    APDAOProxyEventTableCreated,         // 表被创建
    APDAOProxyEventTableDeleted,         // 表被删除
};
```

```
typedef void(^ APDAOProxyEventHandler)(id<APDAOProtocol> proxy, APDAOProxyEventType eve
ntType, NSDictionary* arguments);

/**
 *  这个 Protocol 定义的方法每个 DAO 代理对象都支持，使用时使用 id<APDAOProtocol>对 DAO 对象进行
转换。
 */
@protocol APDAOProtocol <NSObject>

/**
 *  配置文件里 module 可以设置表名，如果想实现配置文件作为一个模板，操作不同的表，可以生成 DAO 对象后
手工向 DAO 对象设计表名。
 *  比如要对与每个 id 的对话消息进行分表的情况。
 */
@property (atomic, strong) NSString* tableName;

/**
 *  返回这个 proxy 操作的表所在数据库文件的路径
 */
@property (atomic, strong, readonly) NSString* databasePath;

/**
 *  获取这个 proxy 操作的数据库文件的句柄
 */
@property (atomic, assign, readonly) sqlite3* sqliteHandle;

/**
 *  注册全局变量参数，这些参数配置文件里的所有方法都可以使用，在配置文件里使用#{name}和@{name}来访问
。
 */
@property (atomic, strong) NSDictionary* globalArguments;

/**
 *  这个 proxy 的事件回调，业务自己来设置。回调线程不确定。
    注意循环引用的问题，业务对象持有 proxy，这个 handler 方法里不要访问业务对象或 proxy，proxy 可以
在回调的第一个参数里拿到。
 */
@property (atomic, copy) APDAOProxyEventHandler proxyEventHandler;

/**
 *  设置加密器，用于加密表里标记为需要加密的列的数据。向表里写数据时，碰到这个列，会调用进行加密。
 *
 *  @param crypt 加密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，需要外部进行 free。如果是
APDefaultEncrypt()，不需要进行释放。
 */
@property (atomic, assign) APDataCrypt* encryptMethod;

/**
 *  设置解密器，用于解密表里标记为需要加密的列的数据。从表里读数据时，碰到这个列，会调用进行解密。
 *
 *  @param crypt 解密结构体，会被拷贝一份。如果传入的 crypt 是外部创建的，需要外部进行 free。如果是
APDefaultDecrypt()，不需要进行释放。
 */
```

```
@property (atomic, assign) APDataCrypt* decryptMethod;

/**
 *  返回 sqlite 的最后一条 rowId
 *
 *  @return sqlite3_last_insert_rowid()
 */
- (long long)lastInsertRowId;

/**
 *  获取配置文件定义的所有方法列表
 */
- (NSArray*)allMethodsList;

/**
 *  删除配置文件里定义的表，可以用于特殊情况下的数据还原。删除表后，DAO 对象仍可以正常使用，再次调用其它方法，会重新创建表。
 */
- (APDAOResult*)deleteTable;

/**
 *  删除符合某个正则规则的所有表，请确保只删除本 Proxy 操作的表，否则可能发生异常
 *
 *  @param pattern     正则表达式
 *  @param autovacuum 删除完成是否自动调用 vacuum 清理数据库空间
 *  @param progress    进度回调，可传 nil，回调不保证主线程。为百分之后的结果
 *
 *  @return 操作是否成功
 */
- (APDAOResult*)deleteTablesWithRegex:(NSString*)pattern autovacuum:(BOOL)autovacuum progress:(void(^)(float progress))progress;

/**
 *  调用自己的数据库链接执行 vacuum 操作
 */
- (void)vacuumDatabase;

/**
 *  DAO 对象可以自己把操作放在事务里提升速度，实际调用的是该 DAO 对象操作的数据库文件 APSharedPreferences 的 daoTransaction 方法。
 */
- (APDAOResult*)daoTransaction:(APDAOTransaction)transaction;

/**
 *  创建一个数据库副连接，为接下来可能发生的并发select 操作加速使用。可以调用多次，创建多个数据库连接待用。
 *  这些创建的链接会自动关闭，业务层无须处理。

 *  @param autoclose 在空闲状态多少秒后自动关闭，0 表示使用系统值
 */
- (void)prepareParallelConnection:(NSTimeInterval)autoclose;

@end
```

1.4.3.4. 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。

  参数如下表：

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。

  参数如下表：

参数	说明
object	对象，如果为 nil 会删除指定 key 的对象。
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。

1.4.3.5. 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。

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` 。例如：

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

执行以上代码就会创建 `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，一般情况下无需使用这个方法，使用 `storageInDocumentsWithName` 即可。使用此接口创建的 `APCustomStorage` 业务需要自己持有，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。

1.4.3.6. 数据清理

本文介绍的是如何自动清理缓存目录。

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，单位为 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, APCacheCleanPhasePostCalculati
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。

> **重要**
>
> 必须为类方法，不能为实例方法。

1.5. iOS 常见问题

本文介绍的是接入 iOS 过程中常见的问题及相应的解决方案。

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];
```

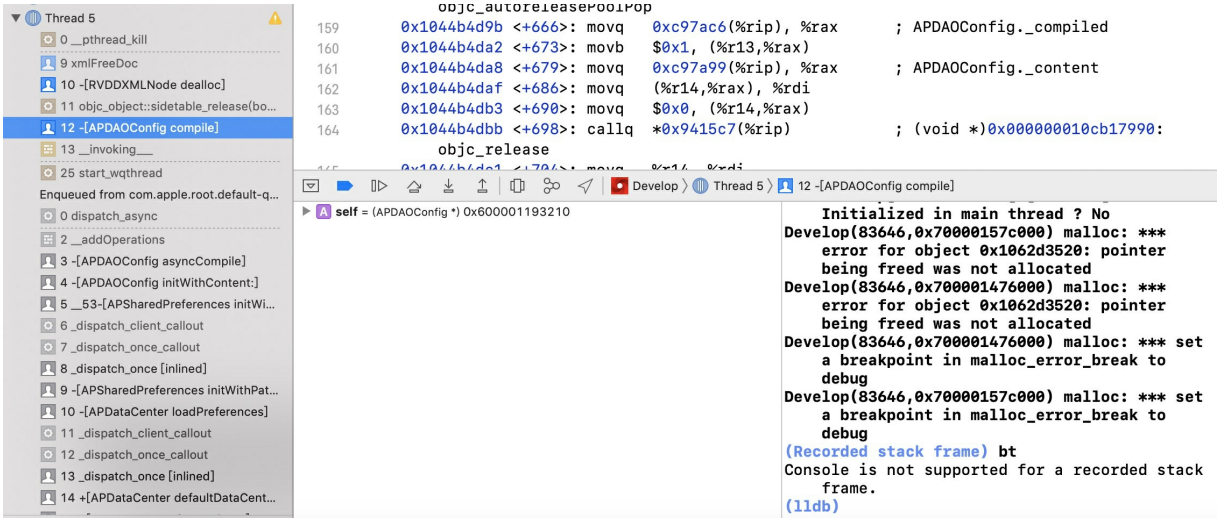
当用户登出时，可以不调用 `setCurrentUserId` 方法，统一存储会继续打开上一个用户的数据库，不会产生影响。

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];
```

- **在统一存储中读取变量：**

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