Ant Technology

Datacenter
User Guide

Document Version: 20260303

蚂蚁集团
ANT GROUP

Legal disclaimer

Ant Group all rights reserved©2022.

Trademark statement

蚂蚁集团 ANT GROUP and other trademarks related to Ant Group are owned by Ant Group. The third-party registered trademarks involved in this document are owned by the right holder according to law.

Disclaimer

The content of this document may be changed due to product version upgrades, adjustments, or other reasons. Ant Group reserves the right to modify the content of this document without notice and the updated versions of this document will be occasionally released through channels authorized by Ant Group. You must pay attention to the version changes of this document as they occur and download and obtain the latest version of this document from Ant Group's authorized channels. Ant Group does not assume any responsibility for direct or indirect losses caused by improper use of documents.

Document conventions

Style	Description	Example
Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	**Danger:** Resetting will result in the loss of user configuration data.
Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	**Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	**Notice:** If the weight is set to 0, the server no longer receives new requests.
Note	A note indicates supplemental instructions, best practices, tips, and other content.	**Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
Courier font	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

1.Datacenter

1.1. About Datacenter

Datacenter provided by mPaaS is a complete solution for persisting storage on mPaaS client. Datacenter SDK provides diversified storage methods to meet different storage requirements.

Features

The features of mPaaS Datacenter vary by platforms.

- **Android platform**:
  - Support SDK database encryption.
  - Reconstructed based on OrmLite (Object Relational Mapping Lite) framework, provide Data Access Objects (DAO) support, simple and easy to use.
  - Support SharePreferences-based key-value pair storage.
  - Support encrypting files before storing them.
- **iOS platform**:
  - Reduce the use of `NSUserDefaults`, and store the large-size data and private data in other places than `NSUserDefaults`, with relatively higher access efficiency compared with using NSUserDefaults.
  - Reduce the cases that the business system automatically maintains files, and reduce the messy files in `Documents` and `Library` directories.
  - Datacenter is divided into storage space irrelevant to users and storage space of the current users by storage space. The business layer doesn't have to be concerned with user switch, and doesn't have to use userId to obtain the current user data.
  - Based on sqlite, Datacenter provides DAO (Data Access Objects) support and is more flexible than CoreData. It encapsulates the database operations by using the configuration file and isolates them from business. The business layer users interfaces to access data and operate database tables.
  - The underlayer supports data encryption.
  - Provide diversified storage methods to meet different storage requirements, and providing memory cache.

1.2. Integrate Android SDK

1.2.1. Quick start

You can integrate Datacenter to your project through Native AAR or Portal & Bundle mode. Multiple storage methods such as database storage, key-value pair storage, and file storage can be implemented to meet different business requirements.

Prerequisites

- If you want to integrate the component to the mPaaS based on the native AAR mode, you need to first complete the prerequisites and the subsequent steps. For more information, see Add mPaaS to your project.
- If you want to integrate the component to the mPaaS based on components, you need to first complete the Component-based integration procedure.

Add the SDK

Native AAR mode

You can use the **component management (AAR)** function to install the **Storage** component in your project. For more information, see AAR component management.

Component-based mode

In your Portal and Bundle projects, install the **Storage** component on the **Component Management** page.

For more information, see Manage component dependencies.

mPaaS initialization

In the native AAR mode, you must initialize the mPaaS.

Add the following code to the

class:

```
public class MyApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        // mPaaS initialization
        MP.init(this);
    }
}
```

For more details, see Initialize mPaaS.

Using storage

If you need to use the database to store related content, see Database storage.

If you need to use the key-value pair to store related content, see Key-value pair storage.

If you need to use the file to store related content, see File storage.

Sample code

See Sample code to obtain sample code.

1.2.2. Advanced guide

Database storage

For information about how to use database storage, see Database storage.

Key-value pair storage

For information about how to use key-value pair storage, see Key-value pair storage.

File storage

For information about how to use file storage, see File storage.

1.3. Integrate iOS SDK

Currently, Datacenter supports integrating based on native framework and using Cocoapods. For different service requirements, various storage solutions are available, including APDataCenter, Key-value storage, DAO storage, LRU storage, custom storage and data cleaning.

This guide introduces how to add the Datacenter SDK to a project in Xcode.

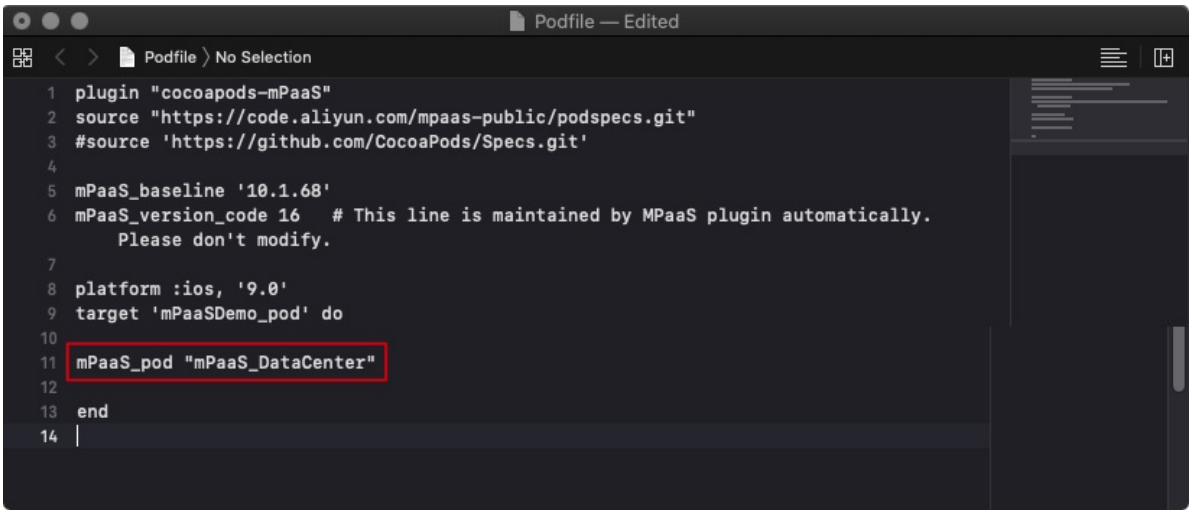
Prerequisite

You have integrated mPaaS to your project. For more information, refer to Integrate based on native framework and using Cocoapods.

Procedure

Use the cocoapods-mPaaS plugin to add the Datacenter SDK to a project.

1. In the **Podfile** file, use `mPaaS_pod "mPaaS_DataCenter"` to add the dependency.



2. Execute `pod install` to complete integrating the SDK.

What to do next

Use the SDK by referring to Datacenter Demo in baseline 10.1.60 and later.

1.4. Storage type

1.4.1. Storage types

Datacenter component for Android client provides the following persistent storage solutions:

Android storage types

Datacenter component for Android client provides the following persistent storage solutions:

- Database storage: Provide the capability of encrypting database underlayer based on OrmLite architecture.
- Key-value pair storage: Do some wrapping based on Android native `SharedPreferences` to improve the usability.
- File storage: Based on Android native `File`, provide file encryption capability.

iOS storage types

Datacenter component for iOS client provides the following persistent storage solutions:

- APDataCenter: The entrance class for Datacenter

- KV storage: Provide interface storage, and simplify the complexity of client-side persistent objects.
- DAO storage: When sqlite access is necessary for business, you can use the DAO function of Datacenter to simplify and encapsulate.
- LRU storage: Provide the storage methods of memory cache and disk cache.
- Custom storage: Provide `APCustomStorage` storage, `APAsyncFileArrayService` storage, `APObjectArrayService` storage, and other custom storage methods.
- Data cleaning: Create a cache directory that can automatically maintain the capacity, and provide the implementation class for cleaning cache.

Instructions on relevant public classes are as follows:

Class name	Function
APDataCenter	Singleton class, entrance class for Datacenter
APSharedPreferences	This class corresponds to a database file, provides a Key-Value storage interface, and contains DAO tables.
APDataCrypt	Symmetric encryption structure
APLRUDiskCache	The disk cache that supports LRU elimination rule
APLRUMemoryCache	The memory cache that supports LRU elimination rule, which is thread-safe
APObjectArrayService	Based on DAO, this class can persist the objects that support NSCoding by business, and supports encryption, capacity limitation and memory cache.
APAsyncFileArrayService	Based on DAO, this class can persist binary data, and supports encryption, capacity limitation and memory cache.
APCustomStorage	Customize storage space, and provide complete user management, Key-Value and DAO storage in this space.
APDAOProtocol	Describe the interfaces that are supported by DAO objects.

1.4.2. Android storage types

1.4.2.1. Database storage

Database storage provided by mPaaS provides the capability of encrypting database underlayer based on OrmLite architecture. You can call the following interface to implement data addition, deletion, modification and query in databases.

- 10.2.3 and later baselines: `com.alibaba.j256.ormlite.dao.Dao`
- 10.1.68 and earlier baseline: `com.j256.ormlite.dao.Dao`

> **Note**
>
> When using the database, please do not directly encrypt the original database, otherwise it will cause the native layer decryption crash. It is recommended that you create a new encrypted database first, and then copy the contents of the original database to the newly created encrypted database.

Examples

- Generate tables
- Create OrmLiteSqliteOpenHelper
- Query data
- Insert data
- Delete data

Generate tables

```
// Database table name, it is the class name by default
@DatabaseTable
public class User {
    // Primary key
    @DatabaseField(generatedId = true)
    public int id;
    // The value of name field must be unique
    @DatabaseField(unique = true)
    public String name;
    @DatabaseField
    public int color;
    @DatabaseField
    public long timestamp;
}
```

Create OrmLiteSqliteOpenHelper

Customize a `DemoOrmLiteSqliteOpenHelper` which inherits from `OrmLiteSqliteOpenHelper`.

With `OrmLiteSqliteOpenHelper`, a database can be created and encrypted.

- 10.2.3 and later baselines:

```
public class DemoOrmLiteSqliteOpenHelper extends OrmLiteSqliteOpenHelper {

    /**
     * Database name
     */
    private static final String DB_NAME = "com_mpaas_demo_storage.db";

    /**
     * Current database version
     */
    private static final int DB_VERSION = 1;
```

```
    /**
     * Database encryption key. mPaaS supports encrypting databases to make the data sa
fer on devices. If it is null, the databases will not be encrypted.
     * Note: The password can only be set once, and there is no API for changing the pa
ssword; encryption of the unencrypted library setting password is not supported (it wil
l cause a crash).
     */
    private static final String DB_PASSWORD = "mpaas";

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

    /**
     * Callback function upon database creation
     *
     * @param sqLiteDatabase: Database
     * @param connectionSource: Connection
     */
    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase, ConnectionSource
connectionSource) {
        try {
            // Create User table
            TableUtils.createTableIfNotExists(connectionSource, User.class);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /**
     * Callback function upon database update
     *
     * @param database: Database
     * @param connectionSource: Connection
     * @param oldVersion: Old database version
     * @param newVersion: New database version
     */
    @Override
    public void onUpgrade(SQLiteDatabase database, ConnectionSource connectionSource, i
nt oldVersion, int newVersion) {
        try {
            // Delete the old version of the User table, and ignore errors
            TableUtils.dropTable(connectionSource, User.class, true);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        try {
            // Rereate User table
            TableUtils.createTableIfNotExists(connectionSource, User.class);
        } catch (SQLException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

- 10.1.68 and earlier baseline:

```
Database encryption key. mPaaS supports encrypting databases to make the data safer on
devices. If it is null, the databases will not be encrypted. Database encryption key. m
PaaS supports encrypting databases to make the data safer on devices. If it is null, th
e databases will not be encrypted.public class DemoOrmLiteSqliteOpenHelper extends OrmL
iteSqliteOpenHelper {

    /**
     * Database name
     */
    private static final String DB_NAME = "com_mpaas_demo_storage.db";

    /**
     * Current database version
     */
    private static final int DB_VERSION = 1;

    /**
     * Database encryption key. mPaaS supports encrypting databases to make the data sa
fer on devices. If it is null, the databases will not be encrypted.
     * Note: The password can only be set once, and there is no API for changing the pa
ssword; encryption of the unencrypted library setting password is not supported (it wil
l cause a crash).
     */
    private static final String DB_PASSWORD = "mpaas";

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

    /**
     * Callback function upon database creation
     *
     * @param sqLiteDatabase   Database
     * @param connectionSource Connection
     */
    @Override
    public void onCreate(SQLiteDatabase sqLiteDatabase, ConnectionSource
connectionSource) {
        try {
            // Create User table
            TableUtils.createTableIfNotExists(connectionSource, User.class);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    /**
```

```
     * Callback function upon database update
     *
     * @param database         Database
     * @param connectionSource Connection
     * @param oldVersion       Old database version
     * @param newVersion       New database version
     */
    @Override
    public void onUpgrade(SQLiteDatabase database, ConnectionSource connectionSource, i
nt oldVersion, int newVersion) {
        try {
            // Delete the old version of the User table, and ignore errors
            TableUtils.dropTable(connectionSource, User.class, true);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        try {
            // Rereate User table
            TableUtils.createTableIfNotExists(connectionSource, User.class);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Query data

Assume that you are to query all data in the `User` table, and sort the data by `timestamp` field in an ascending order.

```
    /**
     * Initialize database data
     */
    private void initData() {
        mData.clear();
        try {

mData.addAll(mDbHelper.getDao(User.class).queryBuilder().orderBy("timestamp", true).que
ry());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

Insert data

```
    /**
     * Insert user information
     *
     * @param user: User information
     */
    private void insertUser(User user) {
        if (null == user) {
            return;
        }
        try {
            // Insert data, mDbHelper is the DemoOrmLiteSqliteOpenHelper that you
customized
            mDbHelper.getDao(User.class).create(user);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

Delete data

```
    /**
     * Delete user information
     *
     * @param user: User information
     */
    private void deleteUser(User user) {
        try {
            // Delete data, mDbHelper is the DemoOrmLiteSqliteOpenHelper that you
customized
            mDbHelper.getDao(User.class).delete(user);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

Related links

OrmLite introduction

1.4.2.2. Key-value pair storage

The key-value pair storage provided by mPaaS is similar with native Android `SharedPreferences`, providing a similar interface. The underlying is the key-value-pair storage system implemented by mPaaS.

Examples

- Create APSharedPreferences
- Query data
- Insert data
- Delete data

Create APSharedPreferences

```
// The context is Android context; GROUP_ID can be regarded as the file name of SharedP
references
APSharedPreferences mAPSharedPreferences =
SharedPreferencesManager.getInstance(context, GROUP_ID);
```

Query data

```
    /**
     * Initialize the data of key-value pairs
     */
    private void initData() {
        mData.clear();
        try {
            // Get the information of all key-value pairs
            mData.putAll((Map<String, String>) mAPSharedPreferences.getAll());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Insert data

```
    /**
     * Insert key-value pairs
     *
     * @param key   key
     * @param value value
     */
    private void insertKeyValue(String key, String value) {
        mAPSharedPreferences.putString(key, value);
        mAPSharedPreferences.commit();
    }
```

Delete data

```
    /**
     * Delete key-value pairs
     *
     * @param key key
     */
    private void deleteKeyValue(String key) {
        mAPSharedPreferences.remove(key);
        mAPSharedPreferences.commit();
    }
```

1.4.2.3. File storage

The file storage provided by mPaaS provides file encryption capability based on Android native `File`.

Important: Because the file encryption uses the encryption function provided by Mobile Security Guard, you must ensure that the encryption images in Mobile Security Guard have been correctly generated.

File types

- **ZFile**: This type of files are stored in `data/data/package_name/files`.
- **ZExternalFile**: This type of files are stored in `sdcard/Android/data/package_name/files`.
- **ZFileInputStream/ZFileOutputStream**: File storage input/output stream, the files will not be encrypted if you use this stream.
- **ZSecurityFileInputStream/ZSecurityFileOutputStream**: File storage input/output stream, the files will be encrypted if you use this stream.

Examples

- Convert files to text
- Convert text to files
- Insert files
- Delete files

Convert files to text

```
    /**
     * Convert files to text
     * @param file: File
     * @return Text
     */
    public String file2String(File file) {
        InputStreamReader reader = null;
        StringWriter writer = new StringWriter();
        try {
            // Use decryption input stream ZSecurityFileInputStream
            // If you don't use encryption/decryption function, then use
ZFileInputStream
            reader = new InputStreamReader(new ZSecurityFileInputStream(file, this));
            //Write the input stream into the output stream
            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();
                }
        }
        //Return the coversion result
        if (writer != null) {
            return writer.toString();
        } else {
            return null;
        }
    }
```

Convert text to files

```
    /**
     * Convert text to files
     * @param res: Text
     * @param file: File
     * @return true means successful while false means failed
     */
    public boolean string2File(String res, File file) {
        boolean flag = true;
        BufferedReader bufferedReader = null;
        BufferedWriter bufferedWriter;
        try {
            bufferedReader = new BufferedReader(new StringReader(res));
            // Use decryption output stream ZSecurityFileOutputStream
            // If you don’t use encryption/decryption function, then use
ZFileOutputStream
            bufferedWriter = new BufferedWriter(new OutputStreamWriter(new
ZSecurityFileOutputStream(file, this)));
            //Character buffers
            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;
    }
```

Insert files

```
    /**
     * Insert files
     *
     * @param file: File
     */
    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();
        }
    }
```

Delete files

```
    /**
     * Delete files
     *
     * @param file: File
     */
    private void deleteFile(BaseFile file) {
        try {
            file.delete();
            mData.remove(file);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

1.4.3. iOS storage types

1.4.3.1. APDataCenter

APDataCenter is a unified storage entry class. It is singleton and can be called anywhere in the code.

```
[APDataCenter defaultDataCenter]
```

You can also use macros.

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

This initializes `APDataCenter`.

API description

Macro definitions

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

Constants

These event notifications usually require no attention from the business-level codes, but Datacenter will throw these notifications.

```
/**
 *  Event notification on that the database file of the previous user is about to be cl
osed.
 */
extern NSString* const kAPDataCenterWillLastUserResign;

/**
 *  Notification on that the user status has been switched. It is possible that the use
r has changed to nil. The specific userId can be acquired with the currentUserId functi
on.
 *  The auxiliary object to this notification is a dictionary. If it is not nil, the @"
switched" key value in it returns `@YES and it indicates that user switch event did hap
pen.
 */
extern NSString* const kAPDataCenterDidUserUpdated;

/**
 *  The user didn't switch, and `APDataCenter` receives the sign-in event again. This n
otification will be thrown.
 */
extern NSString* const kAPDataCenterDidUserRenew;
```

APIs and properties

void APDataCenterLogSwitch(BOOL on);

Enable or disable the console log of Datacenter. It is enabled by default.

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

The userId of the user currently logged in.

(NSString*)preferencesRootPath;

Get the path where the `commonPreferences` and `userPreferences` database folders are stored.

(void)setCurrentUserId:(NSString*)currentUserId;

Set the user ID of the user currently logged in. Do not call it in the business-level codes as it will be called by the login module. Once the user ID is set, `userPreferences` will point to the database of this user.

(void)reset;

Fully reset the Data Center directories. Please use it with caution.

(APSharedPreferences*)commonPreferences;

The user independent global storage database.

(APSharedPreferences*)userPreferences;

The storage database that the user currently logged in. When the user is not logged in, nil will be returned.

(APSharedPreferences)preferencesForUser:(NSString)userId;

Return the storage object of the specified user ID. The business layer usually uses `userPreferences` method for this purpose. When there is a need for asynchronous storage, this method can be used to acquire the storage database of a specified user to avoid data mess-up.

(APPreferencesAccessor)accessorForBusiness: (NSString)business;

Generate an data accessor based on the business name. The business layer needs to hold the object by itself. Once this data accessor is used, the business value will be no longer required for accessing the KV storage.

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

// Equal to

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

(APCustomStorage*)currentVersionStorage;

Datacenter will maintain a database of the current version. When the version is upgraded, the database will be reset.

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

Generate a DAO access object from a configuration file.

Parameter description

Parameter	Description

filePath	The path of the DAO configuration file. For files in the main bundle, use the method below: `NSString* filePath = [[NSBundle mainBundle] pathForResource:@"file" ofType:@"xml"];`
userDependent	Specifies the database operated by the DAO object. If `userDependent=NO`, it indicates that it is not user-specific, and DAO object will create tables in the database files of `commonPreferences`. If `userDependent=YES`, DAO object will create tables in the database files of `userPreferences`. After the user is switched, the subsequent DAO operations will automatically switch to the files of the new user, and the business layer does not need to care about the user switch.

Return value

The DAO object. The business layer does not need to care about its class name but only needs to enforce the switch by using the custom `id<AProtocol>`. The DAO object returned can be switched with `id<APDAOProtocol>` as necessary. The method provided by default will be called. So the custom `AProtocol` should not contain methods defined in `APDAOProtocol`.

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

Create a DAO access object maintaining its own independent database files without using `APSharedPreferences`. The DAO object created using `daoWithPath:userDependent:` interface operates on `commonPreferences` or `userPreferences`. This interface will create a DAO object operating on the database file specified in the `databasePath`. If the file does not exist, it will be created. Multiple DAO objects can be created pointing to the same `databasePath`.

Parameter description

Parameter	Description
filePath	The same as `daoWithPath:userDependent:` interface.
databasePath	The location of the DAO database file. The path can be an absolute path or a relative path, such as `Documents/XXXX.db` or `Library/Movie/XXX.db`.

Return value

DAO object.

1.4.3.2. KV storage

Introduction

In many scenarios, Key-Value storage can meet the client-side storage requirements quite well. `NSUserDefaults` is often used, but it does not support encryption and is slow in data persistence.

The Key-Value storage of Datacenter provides interfaces for storing: PList objects of such data types as NSInteger, long long (the same as NSInteger in 64-bit environment), BOOL, double and NSString, objects supporting NSCoding, and Objective-C objects that can be converted to JSON objects through reflection, greatly reducing the complexity of persistent objects in the client.

For instructions on most of the interfaces of Key-Value storage, see the method description in the header file `APSharedPreferences.h`.

Store basic types of objects

Datacenter provides the following interfaces for storing the basic types of objects:

```
- (NSInteger)integerForKey:(NSString*)key business:(NSString*)business;
- (NSInteger)integerForKey:(NSString*)key business:(NSString*)business defaultValue:(NS
Integer)defaultValue; // Return default value when the data don't exist
- (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; // Return default value when the data don't exist
- (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; // Return default value when the data don't exist
- (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; // Return default value when the data don't exist
- (void)setDouble:(double)value forKey:(NSString*)key business:(NSString*)business;
```

The parameter `defaultValue` is the default value returned when the data don't exist.

Store Objective-C objects

Interface instruction

Datacenter provides the following interfaces for storing the Objective-C objects:

```
- (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

The `setString` and `stringForKey` interfaces are recommended for storing NSString objects as the names are more interpretative.

If the data are not encrypted, strings stored with those two interfaces can be viewed in Sqlite DB viewer in a more intuitive way. Strings stored with `setObject` method will be first converted to NSData through Property List and then saved to the database.

setObject

The `setObject` method is recommended for storing Property List objects to achieve the highest efficiency.

Property List objects: **Property List objects**: NSNumber, NSString, NSData, NSDate, NSArray, and NSDictionary. The sub objects in NSArray and NSDictionary must also be PList objects.

When the Property List objects are saved by using `setObject`, the objects acquired with `objectForKey` method is mutable. The `savedArray` acquired in the following codes is `NSMutableArray`.

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

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

archiveObject

For the Objective-C objects supporting the NSCoding protocol, Datacenter calls the system's `NSKeyedArchiver` convert the objects into NSData objects and persist them.

Property List objects can use this interface, too, but with a low efficiency and thus not recommended.

saveJsonObject

When an Objective-C object is neither a Property List object nor an NSCoding-supporting one, you can use this method to persist it.

Through runtime dynamic reflection, this method maps Objective-C objects to JSON strings. But not all the Objective-C objects can be saved with this method, such as Objective-C objects that have an property serving as C struct pointer, reference each other, or contain dictionaries or arrays in properties.

objectForKey

When Datacenter saves the data of Objective-C objects, it will record the archiving method as well. The `objectForKey` method is used for acquiring objects.

> **Note**
>
> Note: Strings saved using the setString method should be acquired with the stringForKey method.

Encrypt data

Use default encrypt method

Interfaces with `extension` argument support encryption, and pass in `APDataCrypt` struct.

`APDefaultEncrypt` is the default encryption method using AES symmetric encryption.

`APDefaultDecrypt` is the default decryption method and share the key with `APDefaultEncrypt`.

In usual cases, you need only the default encryption method provided by Datacenter, shown as follows:

```
[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"];
```

As the default encryption is applied, the data-acquiring interfaces can skip the `extension` argument.

Use self-defined encryption method

If you have higher security requirements on encryption, you can implement the `APDataCrypt` struct with specified function pointers for encryption and decryption. Ensure that the encryption and decryption methods are matched to save and recover data correctly.

Encrypt objects of basic types

To encrypt and store BOOL, NSInteger, double, and long long objects, you can convert them into strings or put them into the NSNumber, and then call the `setString` or `setObject` interface.

Specify options

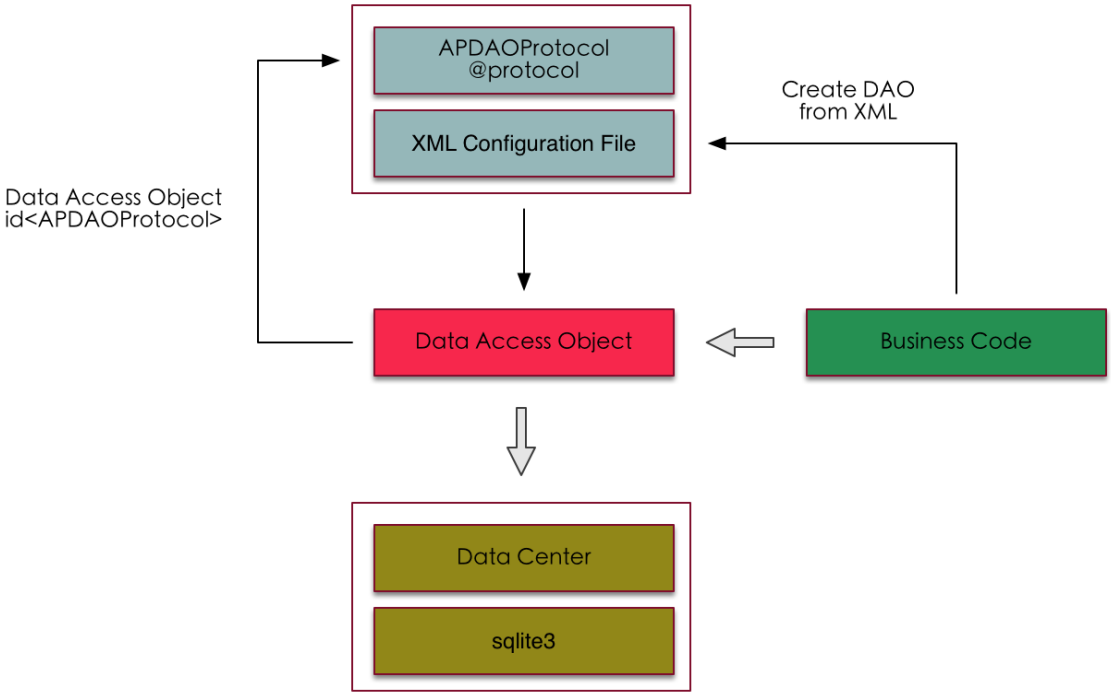
```
typedef NS_OPTIONS (unsigned int, APDataOptions)
{
    //These two options are used to identify data encryption attribute. Do not use them in the interfaces. Please use extension to pass the encryption method.
    APDataOptionDefaultEncrypted    = 1 << 0,       //Do not pass this option. It doesn't work even if it is passed. The Datacenter determines encryption method according to extension in interface rather than options.
    APDataOptionCustomEncrypted     = 1 << 1,       //Do not pass this option. It doesn't work even if it is passed. The Datacenter determines encryption method according to extension in interface rather than options.

    //Indicate that the data can be cleared upon cache cleaning. Here, 1 is casted to unsinged int, because some compilation options may not compute 1 << 31 as unsigned int and cause assignment failure.
    APDataOptionPurgeable           = (unsigned int)1 << 31,
};
```

You can specify options in methods `setObject`, `archiveObject` and `saveJsonObject`. `APDataOptionPurgeable` indicates that the data can be automatically cleared upon data cleaning, see Data cleaning.

1.4.3.3. DAO storage

General KV storage can only store simple data or encapsulated OC objects, and does not support data search. When the business requires the access to SQLite, the DAO function of Datacenter can be utilized for simplification and encapsulation. It works as follows:



1. Define an `xml` configuration file to describe the functions, returned data types and encrypted fields of various SQLite operations.
2. Define an interface for DAO objects, `DAOInterface (@protocol)`. The interface method names and arguments should stay consistent with those described in the configuration file.
3. The business layer passes the `xml` configuration file to the daoWithPath method of APDataCenter, and the DAO access object is generated. The object is directly converted into `id<DAOInterface>`.
4. Next, the business layer will be able to directly call the DAO object methods, and Datacenter will translate the method into the database operations described in the configuration file.

Examples

- The first line of the configuration file defines the default table name and database version, as well as the initialization method.
- `insertItem` and `getItem` are two methods for inserting and reading data. They receive arguments and format the arguments into SQL expressions.
- `createTable` will be called once on the underlying layer by default.

```
<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>
```

- Define the DAO API:

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

- Create a DAO proxy object. Suppose the configuration file is named `demo_config.xml` and is located in Main Bundle.
- Write a piece of data with the `insertItem` method, acquire its index, and then read the written data with the index.

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

- `lastInsertRowId` is a method of `APDAOProtocol`, used for getting the `rowId` of the row last inserted. To enable the DAO object to support this method, you only need to make the `DemoProtocol` inherit from the `APDAOProtocol` in declaration.

Keyword

module

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

- `initializer`: Optional. DAO considers the `update` method specified by `initializer` as a method for building a database table, which will be executed once by default when the first DAO request is initiated.

- `tableName` : Specifies the table name for the default operation in the method below. It can be replaced by `${T} or ${t}` n SQL statements, so that you don't have to input the table name every time. It is recommended that one configuration file be targeted to one table. `tableName` can be empty, so that the same configuration file can operate on multiple tables sharing the same format. For example, to process chat messages in tables, the `setTableName` method of DAO objects can be called for setting the name of the table to be operated on.
- `version` : Represents the version number of the configuration file, in `x.x` format. After the table is created, `tableName` will serve as the key and the table version will be saved to the `TableVersions` table in the database file. `TableVersions` will work in concert with the `upgrade` block for table updates.
- `resetOnUpgrade` : If its value is true or YES, when `version` is updated, the old table will be deleted instead of calling the `upgrade` block. If this argument does not exist, its value is false by default.
- `upgradeMinVersion` : If it is not empty, database files of a earlier earlier than its value will be reset directly. Otherwise the upgrade operation will be performed.

const

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

Define a constant of the string type. `table_columns` is the name of a constant, and the content after the equal sign (=) is the constant value. The constant can be referenced in the configuration file as `${constant name}` .

select

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

`@arguments` :

- List of the argument names, separated by ",". Incoming arguments from the caller are named in turn according to the descriptions in `arguments`. Selectors of DAO objects will not carry the argument name in calling, so arguments must be named in sequence here.
- If an argument has a dollar sign `($)` at its beginning, this argument does not accept the nil value. Calls of DAO interfaces from the business layer allow nil arguments. But if an argument has a dollar sign `($)` at its beginning and the caller accidentally passes a nil value, the DAO call will fail automatically to prevent unexpected issues from happening.
- For more information about how to reference parameters, see Reference methods.

For example, in the code above, the corresponding selector is as follows:

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

If the DAO object calls `[daoProxy find:@1234 time:@2014]` , the ready SQL statement will be:

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

The `NSNumber` value `@1234` will be handed over to SQLite for binding.

`@result` :

- `result` can be the returned value of the DAO method. The use of square brackets `[]` indicates to return the array type of values and iterations occur for the returned values of the SELECT method until no result is returned from SQLite. If the square brackets `[]` are removed, it indicates to return one result only and one iteration is conducted for the SELECT method. It is similar to the `next` method for `FMResultSet` in FMDB database.

Returned types:

- `int` : only returns one result of the `[NSNumber numberWithInt]` type. Note the possibility of overflow.
- `long long` : only returns one result of the `[NSNumber numberWithLongLong]` type.
- `bool` : only returns one result of the `[NSNumber numberWithBool]` type.
- `double` : only returns one result of the `[NSNumber numberWithDouble]` type.
- `string` : only returns one result of the `NSString*` type.
- `binary` : only returns one result of the `NSData*` type.
- `[int]` : an array, with values of the `[NSNumber numberWithInt]` type in the array.
- `[long long]` : an array, with values of the `[NSNumber numberWithLongLong]` type in the array.
- `[bool]` : an array, with values of the `[NSNumber numberWithBool]` type in the array.
- `[double]` : an array, with values of the `[NSNumber numberWithDouble]` type in the array.
- `[string]` : an array, with values of the `NSString*` type in the array.
- `[binary]` : an array, with values of the `NSData*` type in the array.
- `[{}]` : an array, with mapping of column name->column value in the array.
- `[AType]` : an array, with filled custom classes in the array.
- `{}` : only one result, mapping of column name->column value.
- `AType` : only one result, with the filled custom class.
- `[:AMap]` : an array, with XML-defined mapped objects of AMap in the array.
- `:AMap` : only one result. The AMap defined in the configuration file is used to describe the object.

In the example above, the returned type is `:messageModelMap`. The `Objective-C` types returned and columns that require special mapping will be defined in `messageModelMap`. Refer to the keyword map.

`@foreach` :

The SELECT method also supports the `foreach` field and its usage is similar to that of the `insert` , `update` , and `delete` methods to be introduced later. The difference is, if the `select` method specifies the `foreach` argument, the SELECT operation will be executed for N times, and the results will be returned in an array. So if the `select` method in DAO specifies the `foreach` argument, its return value must be defined as `NSArray*` in the protocol.

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>
```

- The INSERT, UPDATE and DELETE methods share the same format. These methods' argument concatenation and reference are the same with that of the SELECT method.
- The INSERT and DELETE keywords serve to differentiate purposes of methods. You can write a "DELETE FROM TABLE" operation in the "UPDATE" function.
- In DAO interfaces, when the INSERT, UPDATE or DELETE methods are executed, the returned value is APDAOResult*, indicating whether the execution succeeded.

`@foreach` :

- When a 'foreach' field follows the INSERT, UPDATE or DELETE methods, the method will query every element in the argument array one by one when the method is called.
- The 'foreach' field must follow the format collectionName.item. The "collectionName" must correspond to an argument in "arguments" and specify the loop container object. It must be of the NSArray or NSSet type when called. The "item" represents the object fetched from the container and will be used as the loop variable. The item name cannot be duplicated with that of any argument in the "arguments". The item can be used as a normal argument in the SQL statement.

For example, the delegate method is:

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

The `messages` is a `MessageModel` array. For every model in the messages, an SQL call will be executed so that elements in the array will be inserted to the database in one shot while the upper layer does not need to care about the loop call. The underlying layer will merge the operation into one transaction to improve efficiency.

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>
```

- In INSERT, UPDATE, DELETE, and UPGRADE methods, you may run into such a circumstance: To execute a DAO method, the SQL update operation is called for multiple times to execute multiple SQL statements. For example, after a table is created, the user wants to create some indexes. The statement packaged in the function will be executed independently as one SQL update operation, and the underlying layer will merge all the operations into one transaction, such as the `createTable` method in the figure above.
- If the step clause exists in a function, no texts are allowed outside the step. The step cannot contain another step.

`@resumable` :

- When the SQL execution generated by this step statement fails, do you continue to execute the following statement. If this argument does not exist, its value is false by default. That is, step is executed in sequence. When a failure occurs, the whole DAO method thinks it fails.

map

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

- It defines a mapping named `messageModelMap` . The actual Objective-C object generated is of the MessageModel class.
- The msgId property of the Objective-C object maps the column value of Column id in the table. Properties not listed are regarded consistent with the column name in the table, thus omitted.

upgrade

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

- With the version updated, the database may have the demand for upgrade. The SQL statements for upgrade are written here. For example, at the very beginning, the version of the configuration file module is 1.0. After upgrade, the configuration file version is changed to 1.1. When the new version configuration file module runs DAO methods for the first time, it will check the current table version with the configuration file version. With an inconsistency found, it will execute the upgrade step by step. This method is called automatically by the underlying layer. The DAO method will be executed after the upgrade is done.
- The upgrade is executed according to the SQL UPDATE statement. If there are multiple statements, they can be enclosed by , which is similar to the implementation of .
- If the operations defined by the upgrade block are required for upgrading the table, the “resetOnUpgrade” argument in the module must be set to “false”.

crypt

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

It describes that the property of the specified class will be encrypted. When data is written to the database, values fetched from this property will be encrypted. When data is read from the database, the generated object will first decrypt the value before assigning values to the property.

For example, in execution of this DAO method, model is of the MessageModel class. As the content property of model is fetched, the value will be encrypted before being written into the database.

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

The execution of this SELECT method will return an object of the MessageModel class. When the underlying layer fetches data from the database and writes the content property to the MessageModel instance, it will first decrypt the data before writting the data, and then return the ready MessageModel object.

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

The methods for setting encryption are defined in APDAOProtocol, as follows:

```
/**
 *  Set an encryptor for encrypting the data in columns marked for encryption in the ta
ble. During data writing to the table, data in this column will be encrypted.
 *
 *  @param crypt The encryption struct which will by copied. If the incoming crypt is c
reated externally, it needs to be freed externally. If it is APDefaultEncrypt(), no fre
e is required.
 */
- (void)setEncryptMethod:(APDataCrypt*)crypt;

/**
 *  Set a decryptor for decrypting the data in columns marked for encryption in the tab
le. During data reading from the table, data in this column will be decrypted.
 *
 *  @param crypt The decryption struct which will be copied. If the incoming crypt is c
reated externally, it needs to be freed externally. If it is APDefaultDecrypt(), no fre
e is required.
 */
- (void)setDecryptMethod:(APDataCrypt*)crypt;
```

If no setting is made, the default encryption of APDataCenter will apply. See KV Store.If a DAO proxy object is `id<DAOProtocol>` And DAOProtocol is `@protocol<APDAOProtocol>`, Then you can call setEncryptMethod and setDecryptMethod directly with the DAO proxy object to set the encryption and decryption methods.

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>
```

- The IF conditional statements can be nested in INSERT, UPFATE, DELETE and SELECT methods. When IF conditions are met, the texts in the IF block will be concatenated into the final SQL statement.
- The IF can be followed by true=”expr” or false=”expr”. The “expr” stands for expression. It can utilize the argument of the method, and “.” to list the argument object properties to call.
- Operators supported by the expression are as follows:

  (): brackets

  +: positive sign

  -: negative sign

  +: plus sign

  -: minus sign

  *: multiplication sign

  /: division sign

  \: exact division sign

  %: modulus

  >: greater than

  <: less than

  >=: greater than or equal to

  <=: less than or equal to

  ==: equal to

  ! =: Not equal to

  and: AND, case-insensitive

  or: OR, case-insensitive

  not: NOT, case-insensitive

  xor: exclusive OR, case-insensitive
- The greater-than sign and less-than sign must be escaped.
- The arguments inside are names of the incoming arguments from external calls, but do not enclose the arguments with #{} or @{} like in the case of the SQL block.
- The meaning of nil here is the same as the nil in Objective-C.
- Strings in the expression should be started or ended with single quotes. Escape characters are not supported, but “\’” is supported to represent a single quote.
- When the arguments are an Objective-C object, ‘.’ is supported for accessing its property, such as the model.read in the example above. If the argument is an array or dictionary, ‘argument name.count’ can be used to get the element count.

Below is a more complex expression:

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

The `title`, `year` and `authors` are all arguments passed from the caller. A nil value is allowed for `title` in the call.

The expression above means “when the book title is not empty, and the year of publication is later than 2010, or there are more than 2 authors for the book”.

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>
```

- It implements similar syntax to SWITCH statements, and its expression requirements are similar to the IF statement. The WHEN keyword can also be followed by `true="expr"` or `false="expr"`.
- Only the first eligible WHEN or OTHERWISE statement will be executed. The OTHERWISE argument can be void.

foreach

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

- The open, separator, close, and reverse arguments can be omitted.
- The `item` represents the loop variable, and `collection` represents the argument name of the loop array.

For example, a method receives string array arguments from the outside. The list content is `@[@"1", @"2", @"3"]`. There is another argument, `prefix=@"abc"`, enclosed by “()”, and separated by “,”. The execution result will be `(abc1,abc2,abc3)`.

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

Foreach statements are usually used for concatenating the “in” blocks in the SELECT statement, for example:

```
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>
```

The WHERE condition will handle superfluous AND and OR (case insensitive) conditions, and it may not return anything when no condition is met, even the WHERE. It is used to concatenate WHERE clauses with a large number of conditions in SQL. As shown in the example above, if only the last judgment is tenable, the statement will correctly return “where author_name like XXX”, instead of “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>
```

The SET keyword will handle the superfluous “,” signs in the end, and return nothing when no condition is met. It is similar to the WHERE statement, but only that it handles the suffixal commas.

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

<trim prefix="SET" suffixOverrides=",">
</trim>
<!--
  is equivalent to<set>
-->
```

- The WHERE and SET statements can be replaced by TRIM statements. TRIM statements define the overall prefix of the statement, and the list of superfluous prefixes and suffixes that each clause will handle (divided by “|”).

- The onVoid argument can appear in WHERE, SET and TRIM statements. It has two values: “ignoreAfter” and “quit”. The values imply the logics used when an empty string is generated because no clause in the TRIM statement is tenable. ignoreAfter means to ignore the following formatting statements and return the current SQL statement for execution, while “quit” means not to execute this SQL statement but to return success.

sql

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

It defines the reusable SQL code segments and uses ${name} to quote the source code segments from other statements.

The `name` in ${name} cannot be ‘T’ or ‘t’, because `${T}` and `${t}` represents the default table name. Inside the sql block other sql blocks can be further quoted.

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>
```

Sometimes the same model may be inserted into the database for multiple times. In this case, the INSERT or REPLACE statements will lead to the loss of old data when conflicts occur in the model primary key (another model of the same primary key already exists in the database), and the data is re-inserted. This will lead to changes in the rowid of the same piece of data. The TRY EXCEPT statement block can solve the problem, among others. TRY EXCEPT statements can only be used in definitions of DAO methods and should have no preceding or following statements. Other statement blocks can be included inside the TRY and EXCEPT statements.

During execution of this DAO method, if execution of the TRY statement fails, the execution will move to the EXCEPT statement automatically. Only when both of them fail will this DAO call return the failure result.

Reference methods

@ reference

`@{something}`: used for method arguments and the argument name is “something”. To format SQL statements, all the object contents will be concatenated into SQL statements. As all the arguments are of the ID type, `[NSString stirngWithFormat:@"%@", id]` is used by default for formatting. The `@{something or ""}` format represents that if the incoming argument is nil, it will be converted to an empty string instead of NULL.

We do not recommend `@{}` for referencing arguments as it is not efficient and subject to the SQL injection risk. If the argument object is of the NSString class, the string will be automatically enclosed in quotes after concatenation to ensure the correctness of the SQL statement format. But if the user has added quotes in the configuration file, the underlying layer will not add the quotes again.

If the `@{something no ""}` format is adopted, you can enforce not to add quotes.

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

In the example above, the incoming id argument is of the NSString class and the codes above are correct. The generated SQL will format and concatenate the id and enclose it with quotes.

reference

`\#{something}` is used for method arguments and the argument name is “something”. To format SQL statements, the argument will be converted to ‘?’ and the object is bound to SQLite. This method is recommended for SQL coding as it is efficient. The `#{something or ""}` format represents that if the incoming argument is nil, it will be converted to an empty string instead of NULL.

$ reference

`${something}` is used for referencing contents in the configuration file, such as the default table name `${T}` or `${t}`, and constants and SQL statement blocks defined in the configuration file.

Chain access

For @ and # references, you can use ‘.’ to list the argument object properties to call. For example, the incoming argument name is “model” of the MessageModel type. It has a property of `NSString* content`. With `@{model.content}`, you will be able to fetch the value of its content property. The internal implementation is `[NSObject valueForKey:]`. So if the argument is a dictionary (The dictionary’s valueForKey is equivalent to `[@""]`), you can also use `#{adict.aaa}` to reference the `adict[@"aaa"]` value.

Delegate method

Each proxy object of generated DAO objects supports APDAOProtocol.

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

The specific method can be found in function comments in the code.

```
#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,    // Will be upgraded very soon.
    APDAOProxyEventUpgradeFailed,        // Table upgrade failed.
    APDAOProxyEventTableCreated,         // Table is created.
    APDAOProxyEventTableDeleted,         // Table is deleted.
};

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

/**
 * The method defined by this protocol is supported by all the DAP proxy objects. In us
age, id<APDAOProtocol> is adopted for converting the DAO object.
 */
@protocol APDAOProtocol <NSObject>

/**
 *  Table names can be set in the configuration file `module`. If you want to use the c
onfiguration file as a template for operations on different tables, you can manually de
sign the table names after the DAO object is generated.
 *  For example, you want to use different tables for conversation messages with differ
ent IDs.
 */
@property (atomic, strong) NSString* tableName;

/**
 *  Return the path of the database file where the operated table by the proxy is locat
ed.
 */
@property (atomic, strong, readonly) NSString* databasePath;

/**
 *  Acquire the handle of the database file operated table by the proxy.
 */
@property (atomic, assign, readonly) sqlite3* sqliteHandle;

/**
 *  Register the global variable arguments. All the methods in configuration files of t
he arguments can be used. #{name} and @{name} can be used for accesses in the configura
tion file.
 */
@property (atomic, strong) NSDictionary* globalArguments;

/**
 *  The event callback of this proxy is set by the business. The callback thread is unc
ertain.
    Pay attention to the circular reference. The business object holds the proxy. Do no
t access the business object of proxy in this handler method and the proxy can be acqui
red in the first argument of the callback.
 */
@property (atomic, copy) APDAOProxyEventHandler proxyEventHandler;

/**
 * Set an encryptor for encrypting the data in columns marked for encryption in the tab
le. During data writing to the table, data in this column will be encrypted.
```

```
 *
 * @param crypt The encryption struct which will be copied. If the incoming crypt is cr
eated externally, it needs to be freed externally. If it is APDefaultEncrypt(), no free
is required.
 */
@property (atomic, assign) APDataCrypt* encryptMethod;

/**
 * Set a decryptor for decrypting the data in columns marked for encryption in the tabl
e. During data reading from the table, data in this column will be decrypted.
 *
 * @param crypt The decryption struct which will be copied. If the incoming crypt is cr
eated externally, it needs to be freed externally. If it is APDefaultDecrypt(), no free
is required.
 */
@property (atomic, assign) APDataCrypt* decryptMethod;

/**
 *  Return the rowId of the last row of SQLite.
 *
 *  @return sqlite3_last_insert_rowid()
 */
- (long long)lastInsertRowId;

/**
 *  Obtain the list of all the methods defined in the configuration file.
 */
- (NSArray*)allMethodsList;

/**
 *  Delete the table defined in the configuration file. It can be used for data recover
y under special circumstances. After the table is deleted, the DAO object can still wor
k normally. When other methods are called, new tables will be created.
 */
- (APDAOResult*)deleteTable;

/**
 *  Delete all the tables conforming to a regular expression rule. Make sure you delete
tables operated by this proxy only, otherwise exceptions may occur.
 *
 *  @param pattern    Regular expression
 *  @param autovacuum After the deletion is complete, whether to call "vacuum" to clear
the database space.
 *  @param progress   Progress callback. Nil value is allowed. The callback is not guar
anteed to be in the main thread. It is a percentage value.
 *
 *  @return Whether the operation is successful.
 */
- (APDAOResult*)deleteTablesWithRegex:(NSString*)pattern autovacuum:(BOOL)autovacuum pr
ogress:(void(^)(float progress))progress;

/**
 *  Call the database link of your own to excecute the  "vacuum" operation.
 */
- (void)vacuumDatabase;
```

```
- (void)vacuumDatabase;

/**
 *  DAO objects can put their operations in transactions to speed up the operation. So
in fact, you call the daoTransaction method of the database file APSharedPreferences th
at this DAO object operates on.
 */
- (APDAOResult*)daoTransaction:(APDAOTransaction)transaction;

/**
 *  Create a parallel connection for the database, for the purpose of speeding up the p
ossible concurrent SELECT operations that may follow. This method can be called for mul
tiple times to create several stand-by connections for the database.
 *  The connections created will be automatically closed and the business layer does no
t to handle them.

 *  @param autoclose Automatically close the connections if they are idle for a certain
number of seconds. The value "0" indicates that the system value will be used.
 */
- (void)prepareParallelConnection:(NSTimeInterval)autoclose;

@end
```

1.4.3.4. LRU storage

Based on LRU elimination rule, LRU storage provides the following two storage methods:

- Memory cache (APLRUMemoryCache): Provide the memory cache based on LRU elimination algorithm. What cached are ID objects. APLRUMemoryCache is thread-safe. In addition, LRU algorithm is implemented based on linked list with high efficiency.
- Disk cache (APLRUDiskCache): Provide the LRU elimination algorithm-based cache that is persisted to databases. What cached are the objects supporting NSCoding. It is easier to maintain databases than files, and using databases makes the disk cleaner.

Memory cache

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

  ```
  Set whether to handle the system memory warning, NO by default. If it is YES, when a
  memory warning appears, the system will clear the cache.
  ```

- **(id)initWithCapacity:(NSInteger)capacity;**

  ```
  Initialize and specify the capacity
  ```

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

  ```
  Store the object into cache. If the object is nil, it will be deleted.
  ```

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

  ```
  Store the object into cache, and specify a expiration timestamp
  ```

- **(id)objectForKey:(NSString*)key;**

  ```
  Get objects
  ```

- **(void)removeObjectForKey:(NSString*)key;**

  ```
  Delete objects
  ```

- **(void)removeAllObjects;**

  ```
  Delete all objects
  ```

- **(void)addObjects:(NSDictionary*)objects;**

  ```
  Add data in batch. It is unable to set the expiration time of every object. By defaul
  t, all objects will never expire.
  ```

- **(void)removeObjectsWithRegex:(NSString*)regex;**

  ```
  Delete data in batch. The key of data matches with the regular expression.
  ```

- **(void)removeObjectsWithPrefix:(NSString*)prefix;**

  ```
  Delete the data that have a specific prefix in batch.
  ```

- **(void)removeObjectsWithSuffix:(NSString*)suffix;**

  ```
  Delete the data that have a specific suffix in batch.
  ```

- **(void)removeObjectsWithKeys:(NSSet*)keys;**

  ```
  Delete all the data that correspond to the keys in batch.
  ```

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

  ```
  Read the cached objects into an array, but not apply the LRU cache strategy on them.
  When fromHead is YES, the system traverses from start to end; otherwise, the end-to-s
  tart traversal is performed.
  ```

- **(BOOL)objectExistsForKey:(NSString*)key;**

  ```
  Fast determine if the object of a key exists or not, without any influence on LRU.
  ```

- **(void)resetCapacity:(NSInteger)capacity;**

  ```
  Reset capacity. If the new capacity is smaller than the original capacity, partial ca
  che will be deleted.
  ```

Disk cache

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

```
Create a persistent LRU cache, and the stored objects must support NsSCoding protocol
.
*  @param name          Cache name, which is used as the table name in the database
*  @param capacity      Capacity (the actual capacity is larger than it), which is us
ed to solve the performance problem on adding data when the cache is full
*  @param userDependent Whether it is related to the user; if yes, when the
APDataCenter.currentUserId is null, the cache doesn't work;
      After user switching, the cache automatically points to the current user’s tabl
e, the business doesn’t necessarily care this event.
*  @param crypted       Whether the data are encrypted
*  @return Cache instance, required for the business
```

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

```
Cache an object, and the expire defaults to 0, namely the object will never expire
```

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

```
Cache an object, and specify a expiration timestamp
*  @param object    Object, if it is nil, the object of the specified key will be del
eted
*  @param key    key
*  @param expire     Expiration timestamp, specify an absolute timestamp relative to
1970 You can use [date timeIntervalSince1970].
```

- **(id)objectForKey:(NSString*)key;**

```
Get objects. If the specified expiration timestamp has been up when the objects are g
ot, the system will return nil and delete the objects. If there are other setObject o
perations that haven’t been completed before objectForKey is called, the system will
keep waiting.
```

- **(void)removeObjectForKey:(NSString*)key;**

```
Delete objects
```

- **(void)removeAllObjects;**

```
Delete all objects
```

- **(void)addObjects:(NSDictionary*)objects;**

```
Add data in batch
```

- **(void)removeObjectsWithSqlLike:(NSString*)like;**

```
Delete data in batch. The key of data uses SQLite LIKE statement to match
```

- **(void)removeObjectsWithKeys:(NSSet*)keys;**

```
Delete all the data that correspond to the keys in batch.
```

1.4.3.5. Custom storage

The default storage space of APDataCenter is the `/Documents/Preferences` directory of the application sandbox. If the business is independent or the data amount is large, you can customize the storage space.

You can create your own storage directory by using APCustomStorage. You can use all services provided by Datacenter in this directory, which is similar to the APDataCenter. For example, you can run the following code to create the `Documents/Contact` directory.

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

The directory contains both `commonPreferences` which stores public data and `userPreferences` which stores user-related data. APCustomStorage is similar to APDataCenter, so you don’t have to pay attention to user switching.

API description

- **(instancetype)storageInDocumentsWithName:(NSString*)name;**

  Create a custom storage with path as /Documents/name.

- **(id)initWithPath:(NSString*)path;**

  This method is usually not used for creating a custom storage at any specified path. The storageInDocumentsWithName can be used instead. The APCustomStorage created with this interface should be held by the yourself. When multiple APCustomStorages share the same path, errors will occur.

- **(APBusinessPreferences*)commonPreferences;**

  For the global storage objects irrelevant to users, access data by using the key-value method. Different from APDataCenter, for APCustomStorage, the business argument is not required for storing key-value data in the custom storage space of the business. Only the key is required.

- **(APBusinessPreferences*)userPreferences;**

  For the global storage objects of login users, access data by using the key-value method. When the user is not logged-in, nil will be returned. Different from APDataCenter, for APCustomStorage, the business argument is not required for storing key-value data in the custom storage space of the business. Only the key is required.

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

  For more information, see description of the APDataCenter class.

1.4.3.6. Data cleanup

Automatically cleaned cache directory

Create an automatically cleaned cache directory. Specify the cleaning logic by using **APPurgeableType** and specify the size of the cache directory by using **size**. Each time the app starts, it checks the directory status in the background process and deletes files as needed. If you set an upper limit for the directory’s capacity, when the directory reaches the upper limit, the app deletes the earliest files to restore the capacity of the cache directory to 1/2 of the upper limit.

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

typedef NS_ENUM(NSUInteger, APPurgeableType)
{
    APPurgeableTypeManual = 0,          // Clear the data when user manually clear the
cache.
    APPurgeableTypeThreeDays = 3,       // Automatically delete the data that was
generated three days ago.
    APPurgeableTypeOneWeek = 7,         // Automatically delete the data that was
generated one week ago.
    APPurgeableTypeTwoWeeks = 14,       // Automatically delete the data that was
generated two weeks ago.
    APPurgeableTypeOneMonth = 30,         // Automatically delete the data that was
generated one month ago.
};

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

    /**
     * Return a cleanable storage path based on the user's input, and automatically det
ermine whether the directory exists. If not, create one.
     *
     * @param userPath: The user-specified path. For example, the path is previously co
ncatenated by using "Documents/SomePath" and is now obtained conveniently by using APPu
rgeableStoragePath(@"Documents/SomePath").
     * @param type: Specify a cleaning type, which can be manual, every week, or every
three days.
     * @param size: Specify a maximum data size in MB. Earlier data is cleared when the
maximum size is reached. 0 indicates that there is no upper limit.
     *
     * @return Target path
     */
    NSString* APPurgeablePath(NSString* path);
    NSString* APPurgeablePathType(NSString* path, APPurgeableType type);
    NSString* APPurgeablePathTypeSize(NSString* path, APPurgeableType type, NSUInteger
size /* MB */);

    /**
     * Clear and reset all registered directories.
     */
    void ResetAllPurgeablePaths();

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

Cache cleaning API

Data Center provides a cache cleaning implementation class, which reads cleaning tasks from **PurgeableCache.plist**. This file must be delivered in the **Main Bundle** of the app. The cleaner is executed asynchronously. The callback function is always called in the main thread and can be used to display and handle UIs.

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

typedef NS_ENUM(NSUInteger, APCacheCleanPhase)
{
    APCacheCleanPhasePreCalculating = 0,    // Scan the sandbox size before cleaning.
    APCacheCleanPhaseCleaning,              // Cleaning
    APCacheCleanPhasePostCalculating,       // Scan the sandbox size after cleaning is
completed.
    APCacheCleanPhaseDone,                  // Cleaned
};

@interface APUserCacheCleaner : NSObject

/**
 * Perform cleaning asynchronously. A callback method must be transferred.
 * progress indicates the real progress when phase returns
APCacheCleanPhasePreCalculating, APCacheCleanPhaseCleaning, or
APCacheCleanPhasePostCalculating. The maximum value is 1.0.
 * progress returns the amount of cleaned data in MB when phase is
APCacheCleanPhaseDone.
 *
 * @param callback: Callback method.
 */
+ (void)execute:(void(^)(APCacheCleanPhase phase, float progress))callback;

@end
```

Two types of cleaning tasks can be defined in **PurgeableCache.plist**.

▼ 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: The file or directory path, which can be the relative path in the sandbox.

Entries: Specify the files or subdirectories to be deleted under **Path** when it is a directory, which supports the wildcard * match.

Class: Specify the definition class of a callback method.

Selectors: Specify the **Class** methods that call the **Class**. Note that the methods must be class methods instead of instance methods.

1.5. FAQ

iOS FAQ

How to set Datacenter to user mode?

Solution: The applications that access mPaaS use their own account system. If you want to use Datacenter to manage the user-mode data, you need to inform Datacenter the first time, so Datacenter can switch the user database, and then inform other business layers.

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

When you log out, you don't have to call `setCurrentUserId` method, and Datacenter continues to open the previous user's database, without any impact.

How to set the default encryption key?

Solution: Datacenter provides default encryption method, and the key is automatically generated by using the `appKey` that is passed in with `mPaasInit` method. It is suggested that the applications accessing mPaaS use their own keys.

Implement the following methods of `mPaasAppInterface` interface, and pass the key to Datacenter in the form of NSData.

```
#pragma mark Datacenter

/**
 * To implement this method, you must return the encryption key (32 bytes) that is used by Datacenter by default. You can let this key be managed by the application with Security Guard, or encrypt and obfuscate the key, and then write it in the client.
 * It is feasible to not to implement the method. Datacenter uses mPaaS and appKey to calculate a result, and uses the result as the encryption key with adequate security.
 *
 * @return 32-byte key, which is placed in NSData
 */
- (NSData*)appDataCenterDefaultCryptKey;
```

It is suggested that each application generates its own 32-byte key, converts the key to Base64 string, and saves it in Security Guard. In this method, the string is extracted through the static interface of Security Guard, and reversely parsed to NSData.

Is Datacenter thread-safe?

Solution: Yes, all the data storage interfaces of Datacenter ar thread-safe. You can call them in any threads.