

# iOS应用架构谈 本地持久化方案及动态部署 (<http://casatwy.com/iosying-yong-jia-gou-tan-ben-di-chi-jiu-hua-fang-an-ji-dong-tai-bu-shu.html>)

**Date** Mon 12 October 2015 **Tags** [iOS](http://casatwy.com/tag/ios.html) (<http://casatwy.com/tag/ios.html>) / [architect](http://casatwy.com/tag/architect.html) (<http://casatwy.com/tag/architect.html>) / [thoughts](http://casatwy.com/tag/thoughts.html) (<http://casatwy.com/tag/thoughts.html>)

[iOS应用架构谈 开篇](#) ([./iosying-yong-jia-gou-tan-kai-pian.html](#))

[iOS应用架构谈 view层的组织和调用方案](#) ([./iosying-yong-jia-gou-tan-viewceng-de-zu-zhi-he-diao-yong-fang-an.html](#))

[iOS应用架构谈 网络层设计方案](#) ([./iosying-yong-jia-gou-tan-wang-luo-ceng-she-ji-fang-an.html](#))

[iOS应用架构谈 本地持久化方案及动态部署](#) ([./iosying-yong-jia-gou-tan-ben-di-chi-jiu-hua-fang-an-ji-dong-tai-bu-shu.html](#))


[iOS应用架构谈 组件化方案](#) ([./iOS-Modulization.html](#))


## 前言

嗯，你们要的大招。跟着这篇文章一起也发布了CTPersistence (<https://github.com/casatwy/CTPersistence>)和CTJSBridge (<https://github.com/casatwy/CTJSBridge>)这两个库，希望大家在实际使用的时候如果遇到问题，就给我提issue或者PR或者评论区。每一个issue和PR以及评论我都会回复的。

持久化方案不管是服务端还是客户端，都是一个非常值得讨论的话题。尤其是在服务端，持久化方案的优劣往往都会在一定程度上影响到产品的性能。然而在客户端，只有为数不多的业务需求会涉及持久化方案，而且在大多数情况下，持久化方案对性能的要求并不是特别苛刻。所以我在移动端这边做持久化方案设计的时候，考虑更多的是方案的可维护和可拓展，然后在此基础上才是性能调优。这篇文章中，性能调优不会单独开一节来讲，而会穿插在各个小节中，大家有心的话可以重点看一下。


### Social

 RSS (<http://casatwy.com/feeds/all.atom.xml>)

 github (<http://github.com/casatwy>)

 facebook (<https://www.facebook.com/taloyum>)

 google+ (<https://plus.google.com/u/0/108264119649922067163>)

 weibo (<http://weibo.com/casatwy>)

### Tags

(<http://casatwy.com/>)

### Links

[casatwy](http://casatwy.com/) (<http://casatwy.com/>)

刘坤的技术博客 (<http://blog.cnbluebox.com>)

齐道长的博客 (<http://qitaos.github.io>)

持久化方案对整个App架构的影响和网络层方案对整个架构的影响类似，一般都是导致整个项目耦合度高的罪魁祸首。而我也是一如既往的 去Model化 的实践者，在持久层去Model化的过程中，我引入了 Virtual Record 的设计，这个在文中也会详细描述。

这篇文章主要讲以下几点：

1. 根据需求决定持久化方案
2. 持久层与业务层之间的隔离
3. 持久层与业务层的交互方式
4. 数据迁移方案
5. 数据同步方案

另外，针对数据库存储这一块，我写了一个CTPersistence (<https://github.com/casatwy/CTPersistence>)，这个库目前能够完成大部分的持久层需求，同时也是我的 Virtual Record 这种设计思路的一个样例。这个库可以直接被cocoapods引入，希望大家使用的时候，能够多给我提issue。这里是CTPersistence Class Reference (<http://persistence.casatwy.com>)。

## 根据需求决定持久化方案

在有需要持久化需求的时候，我们有非常多的方案可供选择：NSUserDefaults、KeyChain、File，以及基于数据库的无数子方案。因此，当有需要持久化的需求的时候，我们首先考虑的是应该采用什么手段去进行持久化。

### NSUserDefaults

一般来说，小规模数据，弱业务相关数据，都可以放到NSUserDefaults里面，内容比较多的数据，强业务相关的数据就不太适合NSUserDefaults了。另外我想吐槽的是，天猫这个App其实是没有一个经过设计的数据持久层的。然后天猫里面的持久化方案就很混乱，我就见到过有些业务线会把大部分业务数据都塞到NSUserDefaults里面去，当时看代码的时候我特么就直接跪了。。。问起来为什么这么做？结果说因为写起来方便～你妹。。。

### keychain

Keychain是苹果提供的带有可逆加密的存储机制，普遍用在各种存密码的需求上。另外，由于App卸载只要系统不重装，Keychain中的数据依旧能够得到保留，以及可被

iCloud同步的特性，大家都会在这里存储用户唯一标识串。所以有需要加密、需要存 iCloud 的敏感小数据，一般都会放在Keychain。

## 文件存储

文件存储包括了Plist、archive、Stream等方式，一般结构化的数据或者需要方便查询的数据，都会以Plist的方式去持久化。Archive方式适合存储平时不太经常使用但很大的数据，或者读取之后希望直接对象化的数据，因为Archive会将对象及其对象关系序列化，以至于读取数据的时候需要Decode很花时间，Decode的过程可以是解压，也可以是对象化，这个可以根据具体 <NSCoding> 中的实现来决定。Stream就是一般的文件存储了，一般用来存存图片啊啥的，适用于比较经常使用，然而数据量又不算非常大的那种。

## 数据库存储

数据库存储的话，花样就比较多了。苹果自带了一个Core Data，当然业界也有无数替代方案可选，不过真正用在iOS领域的除了Core Data外，就是FMDB比较多了。数据库方案主要是为了便于增删改查，当数据有 状态 和 类别 的时候最好还是采用数据库方案比较好，而且尤其是当这些 状态 和 类别 都是强业务相关的时候，就更加要采用数据库方案了。因为你不可能通过文件系统遍历文件去甄别你需要获取的属于某个 状态 或 类别 的数据，这么做成本就太大了。当然，特别大量的数据也不适合直接存储数据库，比如图片或者文章这样的数据，一般来说，都是数据库存一个文件名，然后这个文件名指向的是某个图片或者文章的文件。如果真的要全索引这种需求，建议最好还是挂个API丢到服务端去做。

## 总的说一下

NSUserDefaults、Keychain、File这些持久化方案都非常简单基础，分清楚什么时候用什么就可以了，不要像天猫那样乱写就好。而且在这之上并不会有更复杂的衍生需求，如果真的要针对它们写文章，无非就是写怎么储存怎么读取，这个大家随便Google一下就有了，我就不浪费笔墨了。由于大多数衍生复杂需求都是通过采用基于数据库的持久化方案去满足，所以这篇文章的重点就数据库相关的架构方案设计和实现。如果文章中有哪些问题我没有写到的，大家可以在评论区提问，我会一一解答或者直接把遗漏的内容补充在文章中。

# 持久层实现时要注意的隔离

在设计持久层架构的时候，我们要关注以下几个方面的隔离：

1. 持久层与业务层的隔离
2. 数据库读写隔离
3. 多线程控制导致的隔离
4. 数据表达和数据操作的隔离

## 1. 持久层与业务层的隔离

### 关于Model

在具体讲持久层下数据的处理之前，我觉得需要针对这个问题做一个完整的分析。

在View层设计 (<http://casatwy.com/iosying-yong-jia-gou-tan-viewceng-de-zu-zhi-he-diao-yong-fang-an.html>)中我分别提到了 胖Model 和 瘦Model 的设计思路，而且告诉大家我更加倾向于 胖Model 的设计思路。在网络层设计 (<http://casatwy.com/iosying-yong-jia-gou-tan-wang-luo-ceng-she-ji-fang-an.html>)里面我使用了 去Model化 的思路设计了APIManager与业务层的数据交互。这两个看似矛盾的关于 Model 的设计思路在我接下来要提出的持久层方案中其实是并不矛盾，而且是相互配合的。在网络层设计这篇文章中，我对 去Model化 只给出了思路 and 做法，相关的解释并不多，是因为要解释这个问题涉及面会比较广，写的时候并不认为在那篇文章里做解释是最好的时机。由于持久层在这里 胖Model 和 去Model化 都会涉及，所以我觉得在讲持久层的时候解释这个话题会比较好。

我在跟别的各种领域的架构师交流的时候，发现大家都会或多或少地混用 Model 和 Model Layer 的概念，然后往往导致大家讨论的问题最后都不在一个点上，说 Model 的时候他跟你讲 Model Layer，那好吧，我就跟你讲 Model Layer，结果他又在说 Model，于是问题就讨论不下去了。我觉得作为架构师，如果不分清楚这两个概念，肯定是对你设计的架构的质量有很大影响的。

如果把 Model 说成 Data Model，然后跟 Model Layer 放在一起，这样就能够很容易区分概念了。

### Data Model

Data Model 这个术语针对的问题领域是业务数据的建模，以及代码中这一数据模型的代表方式。两者相辅相成：因为业务数据的建模方案以及业务本身特点，而最终决定了数据的代表方式。同样操作一批数据，你的数据建模方案基本都是细化业务问题之后，抽象得出一个逻辑上的实体。在实现这个业务时，你可以选择不同的代表方式来表征这个逻辑上的实体，比如 字节流 (TCP包等)，字符串流 (JSON、XML等)，对象流。对象流又分 通用数据对象 (NSDictionary等)，业务数据对象 (HomeCellModel等)。

前面已经遍历了所有的 Data Model 的形式。在习惯上，当我们讨论 Model化 时，都

是单指 对象流 中的 业务数据对象 这一种。然而 去Model化 就是指：**更多地使用 通用数据对象 去表征数据**，**业务数据对象 不会在设计时被优先考虑**的一种设计倾向。这里的通用数据对象可以在某种程度上理解为范型。

## Model Layer

Model Layer 描述的问题领域是如何对数据进行增删改查(CURD, C reate U pdate R ead D elete)，和相关业务处理。一般来说如果在 Model Layer 中采用 瘦Model 的设计思路的话，就差不多到CURD为止了。胖Model 还会关心如何为需要数据的上层提供除了增删改查以外的服务，并为他们提供相应的解决方案。例如缓存、数据同步、弱业务处理等。

## 我的倾向

我更加倾向于 去Model化 的设计，在 网络层 (<http://casatwy.com/iosying-yong-jia-gou-tan-wang-luo-ceng-she-ji-fang-an.html>)我设计了 reformer 来实现去Model化。在持久层，我设计了 Virtual Record 来实现去Model化。

因为具体的Model是一种很容易引入耦合的做法，在尽可能弱化Model概念的同时，就能够为引入业务和对接业务提供充分的空间。同时，也能通过去Model的设计达到区分强弱业务的目的，这在将来的代码迁移和维护中，是至关重要的。很多设计不好的架构，就在于架构师并没有认识到区分强弱业务的重要性，所以就导致架构腐化的速度很快，越来越难维护。

所以说回来，持久层与业务层之间的隔离，是通过强弱业务的隔离达到的。而 Virtual Record 正是因为这种去Model化的设计，从而达到了强弱业务的隔离，进而做到持久层与业务层之间既隔离同时又能交互的平衡。具体 Virtual Record 是什么样的设计，我在后面会给大家分析。

# 2. 数据库读写隔离

在网站的架构中，对数据库进行读写分离主要是为了提高响应速度。在iOS应用架构中，对持久层进行读写隔离的设计主要是为了提高代码的可维护性。这也是两个领域要求架构师在设计架构时要求侧重点不同的一个方面。

在这里我们所谓的读写隔离并不是指将数据的读操作和写操作做隔离。而是以某一条界限为准，在这个界限以外的所有数据模型，都是不可写不可修改，或者修改属性的行为不影响数据库中的数据。在这个界限以内的数据是可写可修改的。一般来说我们在设计时划分的这个界限会和持久层与业务层之间的界限保持一致，也就是业务层从持久层拿到数据之后，都不可写不可修改，或业务层针对这一数据模型的写操作、修改操作都对数据库文件中的内容不产生作用。只有持久层中的操作才能够对数据库文件中的内容产生作用。

在苹果官方提供的持久层方案Core Data的架构设计中，并没有针对读写作出隔离，数据的结果都是以 NSManagedObject 抛出。所以只要业务工程师稍微一不小心动一下某

个属性，`NSManagedObjectContext` 在save的时候就会把这个修改给存进去了。另外，当我们需要对所有的增删改查操作做AOP的切片时，`Core Data`技术栈的实现就会非常复杂。

整体上看，我觉得`Core Data`相对大部分需求而言是过度设计了。我当时设计安居客聊天模块的持久层时就采用了`Core Data`，然后为了读写隔离，将所有扔出来的`NSManagedObject`都转为了普通的对象。另外，由于聊天记录的业务相当复杂，使用`Core Data`之后为了完成需求不得不引入很多Hack的手段，这种做法在一定程度上降低了这个持久层的可维护性和提高了接手模块的工程师的学习曲线，这是不太好的。在天猫客户端，我去的时候天猫这个App就已经属于基本毫无持久层可言了，比较混乱。只能依靠各个业务线各显神通去解决数据持久化的需求，难以推动统一的持久层方案，这对于项目维护尤其是跨业务项目合作来说，基本就和车祸现场没啥区别。我现在已经从天猫离职，读者中若是有阿里人想升职想刷存在感拿3.75的，可以考虑给天猫搞个统一的持久层方案。

读写隔离还能够便于加入AOP切点，因为针对数据库的写操作被隔离到一个固定的地方，加AOP时就很容易在正确的地方放入切片。这个会在讲到数据同步方案时看到应用。

### 3. 多线程导致的隔离

#### Core Data

`Core Data`要求在多线程场景下，为异步操作再生成一个`NSManagedObjectContext`，然后设置它的`ConcurrencyType`为`NSPrivateQueueConcurrencyType`，最后把这个Context的parentContext设为Main线程下的Context。这相比于使用原始的`SQLite`去做多线程要轻松许多。只不过要注意的是，如果要传递`NSManagedObject`的时候，不能直接传这个对象的指针，要传`NSManagedObjectID`。这属于多线程环境下对象传递的隔离，在进行架构设计的时候需要注意。

#### SQLite

纯`SQLite`其实对于多线程倒是直接支持，`SQLite`库提供了三种方式：`Single Thread`，`Multi Thread`，`Serialized`。

`Single Thread`模式不是线程安全的，不提供任何同步机制。`Multi Thread`模式要求database connection不能在多线程中共享，其他的在使用上就没什么特殊限制了。`Serialized`模式顾名思义就是由一个串行队列来执行所有的操作，对于使用者来说除了响应速度会慢一些，基本上就没什么限制了。大多数情况下`SQLite`的默认模式是`Serialized`。

根据`Core Data`在多线程场景下的表现，我觉得`Core Data`在使用`SQLite`作为数据载体时，使用的应该就是`Multi Thread`模式。`SQLite`在`Multi Thread`模式下使用的是

读写锁，而且是针对整个数据库加锁，不是表锁也不是行锁，这一点需要提醒各位架构师注意。如果对响应速度要求很高的话，建议开一个辅助数据库，把一个大的写入任务先写入辅助数据库，然后拆成几个小的写入任务见缝插针地隔一段时间往主数据库中写入一次，写完之后再把辅助数据库删掉。

不过从实际经验上看，本地App的持久化需求的读写操作一般都不会大，只要注意好几个点之后一般都不会影响用户体验。因此相比于 Multi Thread 模式， Serialized 模式我认为是性价比比较高的一种选择，代码容易写容易维护，性能损失不大。为了提高几十毫秒的性能而牺牲代码的维护性，我是觉得划不来的。

## Realm

关于Realm我还没来得及仔细研究，所以说不出什么来。

## 4. 数据表达和数据操作的隔离

这是最容易被忽视的一点，数据表达和数据操作的隔离是否能够做好，直接影响的是整个程序的可拓展性。

长久以来，我们都很习惯 Active Record 类型的数据操作和表达方式，例如这样：

```
Record *record = [[Record alloc] init];
record.data = @"data";
[record save];
```

或者这种：

```
Record *record = [[Record alloc] init];
NSArray *result = [record fetchList];
```

简单说就是，让一个对象映射了一个数据库里的表，然后针对这个对象做操作就等同于针对这个表以及这个对象所表达的数据做操作。这里有一个不好的地方就在于，这个 Record 既是数据库中数据表的映射，又是这个表中某一条数据的映射。我见过很多框架(不限于iOS，包括Python, PHP等)都把这两者混在一起去处理。如果按照这种不恰当的方式来组织数据操作和数据表达，在胖Model的实践下会导致强弱业务难以区分从而造成非常大的困难。使用瘦Model这种实践本身就是我认为有缺点的，具体的我在开篇(<http://casatwy.com/iosying-yong-jia-gou-tan-kai-pian.html>)中已经讲过，这里就不细说了。

强弱业务不能区分带来的最大困难在于代码复用和迁移，因为持久层中的强业务对View层业务的高耦合是无法避免的，然而弱业务相对而言只对下层有耦合关系对上层并不存在耦合关系，当我们做代码迁移或者复用时，往往希望复用的是弱业务而不是强业务，若此时强弱业务分不开，代码复用就无从谈起，迁移时就倍加困难。

另外，数据操作和数据表达混在一起会导致的问题在于：客观情况下，数据在view层业务上的表达方式多种多样，有可能是个View，也有可能是个别的什么对象。如果采用映射数据库表的数据对象去映射数据，那么这种多样性就会被限制，实际编码时每到使用数据的地方，就不得不多一层转换。

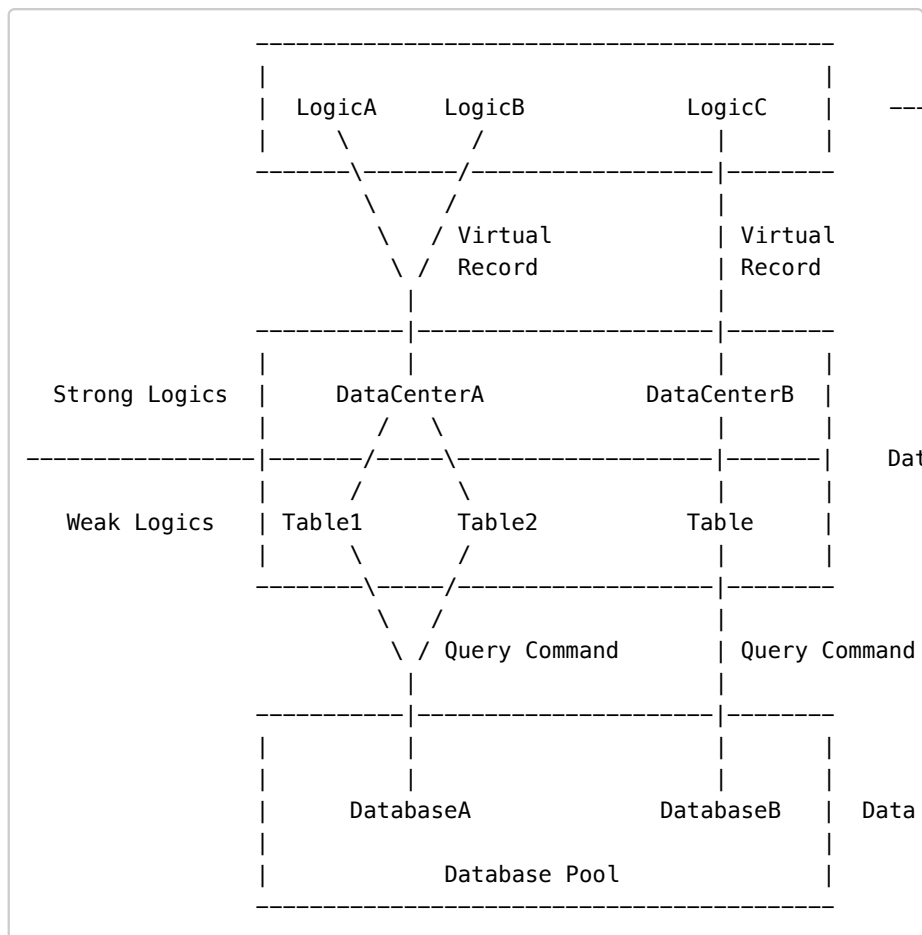
我认为之所以会产生这样不好的做法原因在于，对象对数据表的映射和对象对数据表达的映射结果非常相似，尤其是在表达Column时，他们几乎就是一模一样。在这里要做好针对数据表或是针对数据的映射要做的区分的关键要点是：这个映射对象的操作着手点相对数据表而言，是对内还是对外操作。如果是对内操作，那么这个操作范围就仅限于当前数据表，这些操作映射给数据表模型就比较合适。如果是对外操作，执行这些操作时有可能涉及其他的数据表，那么这些操作就不应该映射到数据表对象中。

因此实际操作中，我是以数据表为单位去针对操作进行对象封装，然后再针对数据记录进行对象封装。数据表中的操作都是针对记录的普通增删改查操作，都是弱业务逻辑。数据记录仅仅是数据的表达方式，这些操作最好交付给数据层分管强业务的对象去执行。具体内容我在下文还会继续说。

## 持久层与业务层的交互方式

说到这里，就不得不说CTPersistence (<https://github.com/casatwy/CTPersistence>)和Virtual Record 了。我会通过它来讲解持久层与业务层之间的交互方式。





我先解释一下这个图：持久层有专门负责对接View层模块或业务的数据中心，它们之间通过Record来进行交互。DataCenter向上层提供业务友好的接口，这一般都是强业务：比如根据用户筛选条件返回符合要求的数据等。

然后DataCenter在这个接口里面调度各个Table，做一系列的业务逻辑，最终生成record对象，交付给View层业务。

DataCenter为了要完成View层交付的任务，会涉及数据组装和跨表的数据操作。数据组装因为View层要求的不同而不同，因此是强业务。跨表数据操作本质上就是各单表数据操作的组合，DataCenter负责调度这些单表数据操作从而获得想要的基础数据用于组装。那么，这时候单表的数据操作就属于弱业务，这些弱业务就由Table映射对象来完成。

Table对象通过QueryCommand来生成相应的SQL语句，并交付给数据库引擎去查询获得数据，然后交付给DataCenter。

### DataCenter 和 Virtual Record

提到 Virtual Record 之前必须先说一下DataCenter。

DataCenter其实是一个业务对象，DataCenter是整个App中，持久层与业务层之间的胶水。它向业务层开放业务友好的接口，然后通过调度各个持久层弱业务逻辑和数据记录来完成强业务逻辑，并将生成的结果交付给业务层。由于DataCenter处在业务层和持久层之间，那么它执行业务逻辑所需要的载体，就要既能够被业务层理解，也能够被持久层理解。

CTPERSISTENCETable 就封装了弱业务逻辑，由DataCenter调用，用于操作数据。而 Virtual Record 就是前面提到的一个既能够被业务层理解，也能够被持久层理解的数据载体。

Virtual Record 事实上并不是一个对象，它只是一个protocol，这就是它 Virtual 的原因。一个对象只要实现了 Virtual Record，它就可以直接被持久层当作Record进行操作，所以它也是一个 Record。连起来就是 Virtual Record 了。所以，Virtual Record 的实现者可以是任何对象，这个对象一般都是业务层对象。在业务层内，常见的数据表达方式一般都是View，所以一般来说 Virtual Record 的实现者也都是一个View对象。

我们回顾一下传统的数据操作过程：一般都是先从数据库中取出数据，然后Model化成一个对象，然后再把这个模型丢到外面，让Controller转化成View，然后再执行后面的操作。

Virtual Record 也是一样遵循类似的步骤。唯一不同的是，整个过程中，它并不需要一个中间对象去做数据表达，对于数据的不同表达方式，由各自 Virtual Record 的实现者自己完成，而不需要把这些代码放到Controller，所以这也就是一个去Model化的设计。如果未来针对这个数据转化逻辑有复用的需求，直接复用 Virtual Record 就可以了，十分方便。

用好 Virtual Record 的关键在于DataCenter提供的接口对业务足够友好，有充足的业务上下文环境。

所以DataCenter一般都是被Controller所持有，所以如果整个App就只有一个DataCenter，这其实并不是一个好事。我见过有很多App的持久层就是一个全局单例，所有持久化业务都走这个单例，这是一种很蛋疼的做法。DataCenter也是需要针对业务做高度分化的，每个大业务都要提供一个DataCenter，然后挂在相关Controller下交给Controller去调度。比如分化成

SettingsDataCenter，ChatRoomDataCenter，ProfileDataCenter 等，另外要注意的是，几个DataCenter之间最好不要有业务重叠。如果一个DataCenter的业务实在是大，那就再拆分成几个小业务。如果单个小业务都很大了，那就拆成各个Category，具体的做法可以参考我的框架中 CTPERSISTENCETable 和 CTPERSISTENCEQueryCommand 的实践。

这么一来，如果要迁移涉及持久层的强业务，那就只需要迁移DataCenter即可。如果要迁移弱业务，就只需要迁移 CTPERSISTENCETable。

## 实际场景

假设业务层此时收集到了用户的筛选条件:

```
NSDictionary *filter = @{
    @"key1":@{
        @"minValue1":@(1),
        @"maxValue1":@(9),
    },
    @"key2":@{
        @"minValue2":@(1),
        @"maxValue2":@(9),
    },
    @"key3":@{
        @"minValue3":@(1),
        @"maxValue3":@(9),
    },
};
```

然后ViewController调用DataCenter向业务层提供的接口, 获得数据直接展示:

```
/* in view controller */

NSArray *fetchedRecordList = [self.dataCenter fetchItemListWithF
[self.dataList appendWithArray:fetchedRecordList];
[self.tableView reloadData];
```

在View层要做的事情其实到这里就已经结束了, 此时我们回过头再来看DataCenter如何实现这个业务:

```

/* in DataCenter */

- (NSArray *)fetchItemListWithFilter:(NSDictionary *)filter
{
    ...
    ...
    ...

    /*
    解析filter获得查询所需要的数据
    whereCondition
    whereConditionParams
    假设上面这两个变量就是解析得到的变量
    */

    ...
    ...
    ...

    /* 告知Table对象查询数据后需要转化成的对象(可选, 统一返回对象可以便于归并) */
    self.itemATable.recordClass = [Item class];
    self.itemBTable.recordClass = [Item class];
    self.itemCTable.recordClass = [Item class];

    /* 通过Table对象获取数据, 此时Table对象内执行的就是弱业务了 */
    NSArray *itemAList = [self.itemATable findAllWithWhereCondition];
    NSArray *itemBList = [self.itemBTable findAllWithWhereCondition];
    NSArray *itemCList = [self.itemCTable findAllWithWhereCondition];

    /* 组装数据 */
    NSMutableArray *resultList = [[NSMutableArray alloc] init];
    [resultList addObjectsFromArray:itemAList];
    [resultList addObjectsFromArray:itemBList];
    [resultList addObjectsFromArray:itemCList];

    return resultList;
}

```

基本上差不多就是上面这样的流程。

一般来说, 架构师设计得差的持久层, 都没有通过设计DataCenter和Table, 去将强业务和弱业务分开。通过设计DataCenter和Table对象, 主要是便于代码迁移。如果迁移强业务, 把DataCenter和Table一起拿走就可以, 如果只是迁移弱业务, 拿走Table就可以了。

另外, 通过代码我希望向你强调一下这个概念: 将Table和Record区分开, 这个在我之前画的架构图上已经有所表现, 不过上文并没有着重强调。其实很多别的架构师在设计持久层框架的时候, 也没有将Table和Record区分开, 对的, 这里我说的框架包括Core Data和FMDB, 这个也不仅限于iOS领域, CodeIgniter、ThinkPHP、Yii、Flask这些也都没有对这个做区分。(这里吐槽一下, 话说上文我还提到Core Data被过度设计了, 事实上该设计的地方没设计到, 不该设计的地方各种设计往上堆...)

以上就是对 Virtual Record 这个设计的简单介绍, 接下来我们就开始讨论不同场景下如何进行交互了。

其中我们最为熟悉的一个场景是这样的：经过各种逻辑组装出一个数据对象，然后把这个数据对象交付给持久层去处理。这种场景我称之为一对一的交互场景，这个交互场景的实现非常传统，就跟大家想得那样，而且CTPersistence (<https://github.com/casatwy/CTPersistence>)的test case里面都是这样的，所以这里我就不多说了。所以，既然你已经知道有了一对一，那么顺理成章地就也会有多对一，以及一对多的交互场景。

下面我会一一描述 Virtual Record 是如何发挥 虚拟 的优势去针对不同场景进行交互的。

## 多对一场景下，业务层如何与持久层交互？

多对一场景其实有两种理解，一种是一个记录的数据由多个View的数据组成。例如一张用户表包含用户的所有资料。然后有的View只包含用户昵称用户头像，有的对象只包含用户ID用户Token。然而这些数据都只存在一张用户表中，所以这是一种 多个对象的数据组成一个完整Record数据 的场景，这是 多对一场景 的理解之一。

第二种理解是这样的，例如一个ViewA对象包含了一个Record的所有信息，然后另一个ViewB对象其实也包含了一个Record的所有信息，这就是一种 多个不同对象表达了一个Record数据 的场景，这也是一种多对一场景的理解。

同时，这里所谓的交互还分两个方向：存和取。

其实这两种理解的解决方案都是一样的，Virtual Record 的实现者通过实现 Merge 操作来完成record数据的汇总，从而实现存操作。任意 Virtual Record 的实现者通过 Merge 操作，就可以将自己的数据交付给其它不同的对象进行表达，从而实现取操作。具体的实现在下面有具体阐释。

### 多对一场景下，如何进行存操作？

```
<CTPersistenceProtocol> 提供了 - (NSObject  
<CTPersistenceRecordProtocol> *)mergeRecord:(NSObject  
<CTPersistenceRecordProtocol> *)record shouldOverride:  
(BOOL)shouldOverride; 这个方法。望文生义一下，就是一个record可以与另外一个  
record进行merge。在 shouldOverride 为NO的情况下，任何一边的 nil 都会被另外  
一边不是 nil 的记录覆盖，如果merge过程中两个对象都不含有这些空数据，则根据  
shouldOverride 来决定是否要让参数中record的数据覆盖自己本身的数据，若  
shouldOverride 为YES，则即便是nil，也会把已有的值覆盖掉。这个方法会返回被  
Merge的这个对象，便于链式调用。
```

举一个代码样例：

```

/*
这里的RecordViewA, RecordViewB, RecordViewC都是符合<CTPersistenceReco
*/

RecordViewA *a;
RecordViewB *b;
RecordViewC *c;

...
收集a, b, c的值的逻辑, 我就不写了~
...

[[a mergeRecord:b shouldOverride:YES] mergeRecord:c shouldOverride:
[self.dataCenter saveRecord:a];

```

基本思路就是通过merge不同的record对象来达到获取完整数据的目的, 由于是 Virtual Record, 具体的实现都是由各自的View去决定。View是最了解自己属性的对象了, 因此它是有充要条件来把自己与持久层相关的数据取出并Merge的, 那么这段凑数据的代码, 就相应分散到了各个View对象中, Controller里面就能够做到非常干净, 整体可维护性也就提高了。

如果采用传统方式, ViewController或者DataCenter中就会散落很多用于凑数据的代码, 写的时候就会出现一大段用于合并的代码, 非常难看, 还不容易维护。

多对一场景下, 如何进行取操作?

其实这样的表述并不恰当, 因为无论 Virtual Record 的实现如何, 对象是谁, 只要从数据库里面取出数据来, 数据就都是能够保证完整的。这里更准确的表述是, 取出数据之后, 如何交付给不同的对象。其实还是用到上面提到的 mergeRecord 方法来处理。

```

/*
这里的RecordViewA, RecordViewB, RecordViewC都是符合<CTPersistenceReco
*/

RecordViewA *a;
RecordViewB *b = [[RecordViewB alloc] init];
RecordViewC *c = [[RecordViewC alloc] init];

a = [self.table findLatestRecordWithError:NULL];
[b mergeRecord:a];
[c mergeRecord:a];

return @[a, b, c]

```

这样就能很容易把a记录的数据交给b和c了, 代码观感同样非常棒, 而且容易写容易维护。

## 一对多场景下，业务层如何与持久层交互？

一对多场景也有两种理解，其一是一个对象包含了多个表的数据，另外一个是一个对象用于展示多种表的数据，这个代码样例其实文章前面已经有过，这一节会着重强调一下。乍看之下两者并没有什么区别，所以我需要指出的是，前者强调的是包含，也就是这个对象是个大熔炉，由多个表的数据组成。

还是举用户列表的例子：

假设数据库中用户相关的表有多张。大多数情况是因为单表Column太多，所以为了提高维护性和查询性能而进行的纵切。

多说一句，纵切在实际操作时，大多都是根据业务场景去切分成多个不同的表，分别来表达用户各业务相关的部分数据，所以纵切的结果就是把Column特别多的一张表拆成Column不那么多的好几个表。虽然数据库经过了纵切，但是有的场景还是要展示完整数据的，比如用户详情页。因此，这个用户详情页的View就有可能包含用户基础信息表(用户名、用户ID、用户Token等)、以及用户详细信息表（用户邮箱地址、用户手机号等）。这就是一对多的一个对象包含了多个表的数据的意思。

后者强调的是展示。举个例子，数据库中有三个表分别是：

二手房、新房、租房，它们三者的数据分别存储在三个表里面，这其实是一种横切。

横切也是一种数据库的优化手段，横切与纵切不同的地方在于，横切是在保留了这套数据的完整性的前提下进行的切分，横切的结果就是把一个原本数据量很大的表，分成了好几个数据量不那么大的表。也就是原来三种房子都能用同一个表来存储，但是这样数据量就太大了，数据库响应速度就会下降。所以根据房子的类型拆成这三张表。横切也有根据ID切的，比如根据ID取余的结果来决定分在哪些表里，这种做法比较广泛，因为拓展起来方便，到时候数据表又大了，大不了除数也跟着再换一个更大的数罢了。其实根据类型去横切也可以，只是拓展的时候就不那么方便。

刚才扯远了现在我再扯回来，这三张表在展示的时候，只是根据类型的不同，界面才有稍许不同而已，所以还是会用同一张View去展示这三种数据，这就是一对多的一个对象用于展示多种表的数据的意思。

一个对象包含了多个表的数据时，如何进行存取操作？

在进行取操作时，其实跟前面多对一的取操作是一样的，用 Merge 操作就可以了。

```
RecordViewA *a;

a = [self.CasaTable findLatestRecordWithError:NULL];
[a mergeRecord:[self.TaloyumTable findLatestRecordWithError:NULL] s
[a mergeRecord:[self.CasatwyTable findLatestRecordWithError:NULL] s

return a;
```

在进行存操作时，Virtual Record 的 <CTPersistenceRecordProtocol> 要求实现者实现 - (NSDictionary \*)dictionaryRepresentationWithColumnInfo: (NSDictionary \*)columnInfo tableName:(NSString \*)tableName; 这个方法，实现者可以根据传入的 columnInfo 和 tableName 返回相应的数据，这样就能够把这一次存数据时关心的内容提供给持久层了。代码样例就是这样的：

```
RecordViewA *a = ..... ;

/*
由于有- (NSDictionary *)dictionaryRepresentationWithColumnInfo:(NSDictionary *)columnInfo tableName:(NSString *)tableName; 这个方法，
所以直接存就好了。
*/

[self.CasaTable insertRecord:a error:NULL];
[self.TaloyumTable insertRecord:a error:NULL];
[self.CasatwyTable insertRecord:a error:NULL];
```

通过上面的存取案例，你会发现使用 Virtual Record 之后，代码量一下子少掉很多，原本那些乱七八糟用于拼凑条件的代码全部被分散进了各个虚拟记录的实现中去了，代码维护因此就变得相当方便。若是采用传统做法，再存取之前少不了要写一大段逻辑，如果涉及代码迁移，这大段逻辑就也得要跟着迁移过去，这就很蛋疼了。

一个对象用于展示多种表的数据，如何进行存取操作？

在这种情况下的存操作其实跟上面一样，直接存。Virtual Record 的实现者自己会根据要存入的表的信息组装好数据提供给持久层。样例代码与上一小节的存操作中给出的一模一样，我就不复制粘贴了。

取操作就不太一样了，不过由于取出时的对象是唯一的(因为一对多嘛)，代码也一样十分简单：



```
ViewRecord *a;
ViewRecord *b;
ViewRecord *c;

self.itemATable.recordClass = [ViewRecord class];
self.itemBTable.recordClass = [ViewRecord class];
self.itemCTable.recordClass = [ViewRecord class];

[a = self.itemATable findLatestRecordWithError:NULL];
[b = self.itemBTable findLatestRecordWithError:NULL];
[c = self.itemCTable findLatestRecordWithError:NULL];
```

这里的 a , b , c 都是同一个View, 然后

itemATable , itemBTable , itemCTable 分别是不同类型的表。这个例子表示了一个对象如何用于展示不同类型的数据。如果使用传统方法, 这里少不了要写很多适配代码, 但是使用 Virtual Record 之后, 这些代码都由各自实现者消化掉了, 在执行数据逻辑时可以无需关心适配逻辑。

## 多对多场景?

其实多对多场景就是上述这些 一对多 和 多对一 场景的排列组合, 实现方式都是一模一样的, 我这里就也不多啰嗦了。

## 交互方案的总结

在交互方案的设计中, 架构师应当区分好强弱业务, 把传统的 Data Model 区分成 Table 和 Record , 并由 DataCenter 去实现强业务, Table 去实现弱业务。在这里由于 DataCenter 是强业务相关, 所以在实际编码中, 业务工程师负责创建 DataCenter, 并向业务层提供业务友好的方法, 然后再在DataCenter中操作Table来完成业务层交付的需求。区分强弱业务, 将 Table 和 Record 拆分开的好处在于:

1. 通过业务细分降低耦合度, 使得代码迁移和维护非常方便
2. 通过拆解数据处理逻辑和数据表达形态, 使得代码具有非常良好的可拓展性
3. 做到读写隔离, 避免业务层的误操作引入Bug
4. 为Virtual Record这一设计思路的实践提供基础, 进而实现更灵活, 对业务更加友好的架构

任何不区分强弱业务的架构都是架构师在耍流氓, 嗯。

在具体与业务层交互时, 采用 Virtual Record 的设计思路来设计 Record , 由具体的业务对象来实现Virtual Record, 并以它作为DataCenter和业务层之间的数据媒介进行

交互。而不是使用传统的数据模型来与业务层做交互。

使用 Virtual Record 的好处在于：

1. 将数据适配和数据转化逻辑封装到具体的Record实现中，可以使得代码更加抽象简洁，代码污染更少
2. 数据迁移时只需要迁移Virtual Record相关方法即可，非常容易拆分
3. 业务工程师实现业务逻辑时，可以在不损失可维护性的前提下，极大提高业务实现的灵活性

这一部分还顺便提了一下 横切 和 纵切 的概念。本来是打算有一小节专门写数据库性能优化的，不过事实上移动App场景下数据库的性能优化手段不像服务端那样丰富多彩，很多牛逼技术和参数调优手段想用也用不了。差不多就只剩下数据切片的手段比较有效了，所以性能优化这块感觉没什么好写的。其实大家了解了切片的方式和场景，就足以根据自己的业务场景去做优化了。再使用一下Instrument的Time Profile再配合SQLite提供的一些函数，就足以找到慢在哪儿，然后去做性能调优了。但如果我把这些也写出来，就变成教你怎么使用工具，感觉这个太low写着也不起劲，大家有兴趣搜使用手册下来看就行。

## 数据库版本迁移方案

一般来说，具有持久层的App同时都会附带着有版本迁移的需求。当一个用户安装了旧版本的App，此时更新App之后，若数据库的表结构需要更新，或者数据本身需要批量地进行更新，此时就需要有版本迁移机制来进行这些操作。然而版本迁移机制又要兼顾跨版本的迁移需求，所以基本上大方案也就只有一种：建立数据库版本节点，迁移的时候一个一个跑过去。

数据迁移事实上实现起来还是比较简单的，做好以下几点问题就不大了：

1. 根据应用的版本记录每一版数据库的改变，并将这些改变封装成对象
2. 记录好当前数据库的版本，便于跟迁移记录做比对
3. 在启动数据库时执行迁移操作，如果迁移失败，提供一些降级方案

CTPersistence (<https://github.com/casatwy/CTPersistence>)在数据迁移方面，凡是对于数据库原本没有的数据表，如果要新增，在使用table的时候就会自动创建。因此对于业务工程师来说，根本不需要额外多做什么事情，直接用就可以了。把这部分工作放到这里，也是为数据库版本迁移节省了一些步骤。

CTPersistence也提供了Migrator。业务工程师可以自己针对某一个数据库编写一个Migrator。这个Migrator务必派生自CTPersistenceMigrator，且符合<CTPersistenceMigratorProtocol>，只要提供一个migrationStep的字典，以及记录版本顺序的数组。然后把你自己派生的Migrator的类名和对应关心的数据库名写在CTPersistenceConfiguration.plist里面就可以。CTPersistence会在初始数据库的时候，根据plist里面的配置对应找到Migrator，并执行数据库版本迁移的逻辑。

在版本迁移时要注意的一点是性能问题。我们一般都不会在主线程做版本迁移的事情，这自然不必说。需要强调的是，SQLite本身是一个容错性非常强的数据库引擎，因此差不多在执行每一个SQL的时候，内部都是走的一个Transaction。当某一版的SQL数量特别多的时候，建议在版本迁移的方法里面自己建立一个Transaction，然后把相关的SQL都包起来，这样SQLite执行这些SQL的时候速度就会快一点。

其他的似乎并没有什么要额外强调的了，如果有没说到的地方，大家可以在评论区提出来。

## 数据同步方案

数据同步方案大致分两种类型，一种类型是 单向数据同步，另一种类型是 双向数据同步。下面我会分别说说这两种类型的数据同步方案的设计。

### 单向数据同步

单向数据同步就是只把本地较新数据的操作同步到服务器，不会从服务器主动拉取同步操作。

比如即时通讯应用，一个设备在发出消息之后，需要等待服务器的返回去知道这个消息是否发送成功，是否取消成功，是否删除成功。然后数据库中记录的数据就会随着这些操作是否成功而改变状态。但是如果换一台设备继续执行操作，在这个新设备上只会拉取旧的数据，比如聊天记录这种。但对于旧的数据并没有删除或修改的需求，因此新设备也不会问服务器索取数据同步的操作，所以称之为单向数据同步。

单向数据同步一般来说也不需要去job去做定时更新的事情。如果一个操作迟迟没有收到服务器的确认，那么在应用这边就可以认为这个操作失败，然后一般都是在界面上把这些失败的操作展示出来，然后让用户去勾选需要重试的操作，然后再重新发起请求。微信在消息发送失败的时候，就是消息前面有个红色的圈圈，里面有个感叹号，只有用户点击这个感叹号的时候才重新发送消息，背后不会有个job一直一直跑。

所以细化需求之后，我们发现单向数据同步只需要做到能够同步数据的状态即可。

### 如何完成单向数据同步的需求

## 添加identifier

添加identifier的目的主要是为了解决客户端数据的主键和服务端数据的主键不一致的问题。由于是单向数据同步，所以数据的生产者只会是当前设备，那么identifier也理所应当由设备生成。当设备发起同步请求的时候，把identifier带上，当服务器完成任务返回数据时，也把这些identifier带上。然后客户端再根据服务端给到的identifier再更新本地数据的状态。identifier一般都会采用UUID字符串。

## 添加isDirty

isDirty主要是针对数据的插入和修改进行标识。当本地新生成数据或者更新数据之后，收到服务器的确认返回之前，isDirty置为YES。当服务器的确认包返回之后，再根据包里提供的identifier找到这条数据，然后置为NO。这样就完成了数据的同步。

然而这只是简单的场景，有一种比较极端的情况在于，当请求发起到收到请求回复的这短短几秒间，用户又修改了数据。如果按照当前的逻辑，在收到请求回复之后，这个又修改了的数据的isDirty会被置为NO，于是这个新的修改就永远无法同步到服务器了。这种极端情况的简单处理方案就是在发起请求到收到回复期间，界面上不允许用户进行修改。

如果希望做得比较细致，在发送同步请求期间依旧允许用户修改的话，就需要在数据库额外增加一张 DirtyList 来记录这些操作，这个表里至少要有两个字

段：identifier，primaryKey。然后每一次操作都分配一次identifier，那么新的修改操作就有了新的identifier。在进行同步时，根据 primaryKey 找到原数据表里的那条记录，然后把数据连同identifier交给服务器。然后在服务器的确认包回来之后，就只要拿出identifier再把这条操作记录删掉即可。这个表也可以直接服务于多个表，只是还需要额外添加一个 tablename 字段，方便发起同步请求的时候能够找得到数据。

## 添加isDeleted

当有数据同步的需求的时候，删除操作就不能是简单的物理删除了，而只是逻辑删除，所谓逻辑删除就是在数据库里把这条记录的isDeleted记为YES，只有当服务器的确认包返回之后，才会真正把这条记录删除。isDeleted和isDirty的区别在于：收到确认包后，返回的identifier指向的数据如果是isDeleted，那么就要删除这条数据，如果指向的数据只是新插入的数据和更新的数据，那么就只要修改状态就行。插入数据和更新数据在收到数据包之后做的操作是相同的，所以就用isDirty来区分就足够了。总之，这是根据收到确认包之后的操作不同而做的区分。两者都要有，缺一不可。

## 在请求的数据包中，添加dependencyIdentifier

在我看到的很多其它数据同步方案中，并没有提供dependencyIdentifier，这会导致一个这样的问题：假设有两次数据同步请求一起发出，A先发，B后发。结果反而是B请求先到，A请求后到。如果A请求的一系列同步操作里面包含了插入某个对象的操作，B请求的一系列同步操作里面正好又删除了这个对象，那么由于到达次序的先后问题错乱，就导致这个数据没办法删除。

这个在移动设备的使用场景下是很容易发生的，移动设备本身网络环境就多变，先发的包反而后到，这种情况出现的几率还是比较大的。所以在请求的数据包中，我们要带上上一次请求时一系列identifier的其中一个，就可以了。一般都是选择上次请求里面最后的那一个操作的identifier，这样就能表征上一次请求的操作了。

服务端这边也要记录最近的100个请求包里面的最后一个identifier。之所以是100条纯属只是拍脑袋定的数字，我觉得100条差不多就够了，客户端发请求的时候dependency应该不会涉及到前面100个包。服务端在收到同步请求包的时候，先看dependencyIdentifier是否已被记录，如果已经被记录了，那么就执行这个包里面的操作。如果没有被记录，那就先放着再等等，等到条件满足了再执行，这样就能解决这样的问题。

之所以不用更新时间而是identifier来做标识，是因为如果要用时间做标识的话，就是只能以客户端发出数据包时候的时间为准。但有时不同设备的时间不一定完全对得上，多少会差个几秒几毫秒，另外如果同时有两个设备发起同步请求，这两个包的时间就都是一样的了。假设A1, B1是1号设备发送的请求，A2, B2，是2号设备发送的请求，如果用时间去区分，A1到了之后，B2说不定就直接能够执行了，而A1还没到服务器呢。

当然，这也是一种极端情况，用时间的话，服务器就只要记录一个时间了，凡是依赖时间大于这个时间的，就都要再等等，实现起来就比较方便。但是为了保证bug尽可能少，我认为依赖还是以identifier为准，这要比以时间为准更好，而且实现起来其实也并没有增加太多复杂度。

## 单向数据同步方案总结

1. 改造的时候添加identifier，isDirty，isDeleted字段。如果在请求期间依旧允许对数据做操作，那么就要把identifier和primaryKey再放到一个新的表中
2. 每次生成数据之后对应生成一个identifier，然后只要是针对数据的操作，就修改一次isDirty或isDeleted，然后发起请求带上identifier和操作指令去告知服务器执行相关的操作。如果是复杂的同步方式，那么每一次修改数据时就新生成一次identifier，然后再发起请求带上相关数据告知服务器。
3. 服务器根据请求包的identifier等数据执行操作，操作完毕回复给客户端确认
4. 收到服务器的确认包之后，根据服务器给到的identifier（有的时候也会有tablename，取决于你的具体实现）找到对应的记录，如果是删除操作，直接把数据删除就好。如果是插入和更新操作，就把isDirty置为NO。如果有额外的表记录了更新操作，直接把identifier对应的这个操作记录删掉就行。

## 要注意的点

在使用表去记录更新操作的时候，短时间之内很有可能针对同一条数据进行多次更新操作。因此在同步之前，最好能够合并这些相同数据的更新操作，可以节约服务器的计算资源。当然如果你服务器强大到不行，那就无所谓了。

## 双向数据同步

双向数据同步多见于笔记类、日程类应用。对于一台设备来说，不光自己会往上推数据同步的信息，自己也会问服务器主动索取数据同步的信息，所以称之为双向数据同步。

举个例子：当一台设备生成了某时间段的数据之后，到了另外一台设备上，又修改了这些旧的历史数据。此时再回到原来的设备上，这台设备就需要主动问服务器索取是否旧的数据有修改，如果有，就要把这些操作下载下来同步到本地。

双向数据同步实现上会比单向数据同步要复杂一些，而且有的时候还会存在实时同步的需求，比如协同编辑。由于本身方案就比较复杂，另外一定要兼顾业务工程师的上手难度（这主要看你这个架构师的良心），所以要实现双向数据同步方案的话，还是很有意思比较有挑战的。

## 如何完成双向数据同步的需求

### 封装操作对象

这个其实在单向数据同步时多少也涉及了一点，但是由于单向数据同步的要求并不复杂，只要告诉服务器是什么数据然后要做什么事情就可以了，倒是没必要将这种操作封装。在双向数据同步时，你也得解析数据操作，所以互相之间要约定一个协议，通过封装这个协议，就做到了针对操作对象的封装。

这个协议应当包括：

1. 操作的唯一标识
2. 数据的唯一标识
3. 操作的类型
4. 具体的数据，主要是在Insert和Update的时候会用到
5. 操作的依赖标识
6. 用户执行这项操作时的时间戳

分别解释一下这6项的意义：

### 1. 操作的唯一标识

这个跟单向同步方案时的作用一样，也是在收到服务器的确认包之后，能够使得本地应用找到对应的操作并执行确认处理。

### 1. 数据的唯一标识

在找到具体操作的时候执行确认逻辑的处理时，都会涉及到对象本身的处理，更新也好删除也好，都要在本地数据库有所体现。所以这个标识就是用于找到对应数据的。

### 1. 操作的类型

操作的类型就是 Delete，Update，Insert，对应不同的操作类型，对本地数据库执行的操作也会不一样，所以用它来进行标识。

### 1. 具体的数据

当更新的时候有 Update 或者 Insert 操作的时候，就需要有具体的数据参与了。这里的数据有的时候不见得是单条的数据内容，有的时候也会是批量的数据。比如把所有10月1日之前的任务都标记为已完成状态。因此这里具体的数据如何表达，也需要定一个协议，什么时候作为单条数据的内容去执行插入或更新操作，什么时候作为批量的更新去操作，这个自己根据实际业务需求去定义就行。

### 1. 操作的依赖标识

跟前面提到的依赖标识一样，是为了防止先发的包后到后发的包先到这种极端情况。

## 1. 用户执行这项操作的时间戳

由于跨设备，又因为旧数据也会被更新，因此在一定程度上就会出现冲突的可能。操作数据在从服务器同步下来之后，会存放在一个新的表中，这个表就是 待操作 数据表，在具体执行这些操作的同时会跟 待同步 的数据表中的操作数据做比对。如果是针对同一条数据的操作，且这两个操作存在冲突，那么就以时间戳来决定如何执行。还有一种做法就是直接提交到界面告知用户，让用户做决定。

## 新增待操作数据表和待同步数据表

前面已经部分提到这一点了。从服务器拉下来的同步操作列表，我们存在 待执行 数据表中，操作完毕之后如果有告知服务器的需求，那就等于是走单向同步方案告知服务器。在执行过程中，这些操作也要跟 待同步 数据表进行匹配，看有没有冲突，没有冲突就继续执行，有冲突的话要么按照时间戳执行，要么就告知用户让用户做决定。在拉取 待执行 操作列表的时候，也要把最后一次操作的identifier丢给服务器，这样服务器才能返回相应数据。

待同步 数据表的作用其实也跟单向同步方案时候的作用类似，就是防止在发送请求的时候用户有操作，同时也是为解决冲突提供方便。在发起同步请求之前，我们都应该先去查询有没有 待执行 的列表，当 待执行 的操作列表同步完成之后，就可以删除里面的记录了，然后再把本地 待同步 的数据交给服务器。同步完成之后就可以把这些数据删掉了。因此在正常情况下，只有在 待操作 和 待执行 的操作间会存在冲突。有些从道理上讲也算是冲突的事情，比如获取 待执行 的数据比较晚，但其中又和 待同步 中的操作有冲突，像这种极端情况我们其实也无解，只能由他去，不过这种情况也是属于比较极端的情况，发生几率不大。

## 何时从服务器拉取待执行列表

1. 每次要把本地数据丢到服务器去同步之前，都要拉取一次待执行列表，执行完毕之后再上传本地同步数据
2. 每次进入相关页面的时候都更新一次，看有没有新的操作
3. 对实时性要求比较高的，要么客户端本地起一个线程做轮询，要么服务器通过长链接将 待执行 操作推送过来
4. 其它我暂时也想不到了，具体还是看需求吧

## 双向数据同步方案总结



1. 设计好同步协议，用于和服务端进行交互，以及指导本地去执行同步下来的操作
2. 添加 待执行，待同步 数据表记录要执行的操作和要同步的操作

## 要注意的点

我也见过有的方案是直接把SQL丢出去进行同步的，我不建议这么做。最好还是将操作和数据分开，然后细化，否则检测冲突的时候你就得去分析SQL了。要是这种实现中有什么bug，解这种bug的时候就要考虑前后兼容问题，机制重建成本等，因为贪图一时偷懒，到最后其实得不偿失。

# 总结

这篇文章主要是基于CTPersistence (<https://github.com/casatwy/CTPersistence>)讲了一下如何设计持久层的设计方案，以及数据迁移方案和数据同步方案。

着重强调了一下各种持久层方案在设计时要考虑的隔离，以及提出了 Virtual Record 这个设计思路，并对它做了一些解释。然后在数据迁移方案设计时要考虑的一些点。在数据同步方案这一节，分开讲了单向的数据同步方案和双向的数据同步方案的设计，然而具体实现还是要依照具体的业务需求来权衡。

希望大家觉得这些内容对各自工作中遇到的问题能够有所价值，如果有问题，欢迎在评论区讨论。

另外，关于动态部署方案，其实直到今天在iOS领域也并没有特别好的动态部署方案可以拿出来，我觉得最靠谱的其实还是H5和Native的Hybrid方案。React Native在我看来相比于Hybrid还是有比较多的限制。关于Hybrid方案，我也提供了CTJSBridge (<https://github.com/casatwy/CTJSBridge>)这个库去实现这方面的需求。在动态部署方案这边其实成文已经很久，迟迟不发的原因还是因为觉得当时并没有什么银弹可以解决iOS App的动态部署，另外也有一些问题没有考虑清楚。当初想到的那些问题现在我已经确认无解。当初写的动态部署方案我一直认为它无法作为一个单独的文章发布出来，所以我就把这篇文章也放在这里，权当给各位参考。

# iOS动态部署方案

# 前言

这里讨论的动态部署方案，就是指通过不发版的方式，将新的内容、新的业务流程部署进已发布的App。因为苹果的审核周期比较长，而且苹果的限制比较多，业界在这里也没有特别多的手段来达到动态部署方案的目的。这篇文章主要的目的就是给大家列举一下目前业界做动态部署的手段，以及其对应的优缺点。然后给出一套我比较倾向于使用的方案。

其实单纯就动态部署方案来讲，没什么太多花头可以说的，就是H5、Lua、JS、OC/Swift这几门基本技术的各种组合排列。写到后面觉得，动态部署方案其实是非常好的用于讲解某些架构模式的背景。一般我们经验总结下来的架构模式包括但不限于：

1. Layered Architecture
2. Event-Driven Architecture
3. Microkernel Architecture
4. Microservices Architecture
5. Space-Based Architecture

我在开篇 (<http://casatwy.com/iosying-yong-jia-gou-tan-kai-pian.html>)里面提到的MVC等方案跟这篇文章中要提到的架构模式并不是属于同一个维度的。比较容易混淆的就是容易把MVC这些方案跟 Layered Architecture 混淆，这个我在开篇 (<http://casatwy.com/iosying-yong-jia-gou-tan-kai-pian.html>)这篇文章里面也做过了区分：MVC等方案比较侧重于数据流动方向的控制和数据流的管理。Layered Architecture 更加侧重于各分层之间的功能划分和模块协作。

另外，上述五种架构模式在Software Architecture Patterns (<http://www.oreilly.com/programming/free/files/software-architecture-patterns.pdf>)这本书里有非常详细的介绍，整本书才45页，个把小时就看完了，非常值得看和思考。本文后半篇涉及的架构模式是以上架构模式的其中两种：Microkernel Architecture 和 Microservices Architecture。

最后，文末还给出了其他一些关于架构模式的我觉得还不错的PPT和论文，里面对架构模式的分类和总结也比较多样，跟 Software Architecture Patterns 的总结也有些不一样的地方，可以博采众长。

## Web App

### 实现方案

其实所谓的web app，就是通过手机上的浏览器进行访问的H5页面。这个H5页面是针对移动场景特别优化的，比如UI交互等。

### 优点

1. 无需走苹果流程，所有苹果流程带来的成本都能避免，包括审核周期、证书成本等。
2. 版本更新跟网页一样，随时生效。
3. 不需要Native App工程师的参与，而且市面上已经有很多针对这种场景的框架。

## 缺点

1. 由于每一页都需要从服务器下载，因此web app重度依赖网络环境。
2. 同样的UI效果使用web app来实现的话，流畅度不如Native，比较影响用户体验。
3. 本地持久化的部分很难做好，绕过本地持久化的部分的办法就是提供账户体系，对应账户的持久化数据全部存在服务端。
4. 即时响应方案、远程通知实现方案、移动端传感器的使用方案复杂，维护难度大。
5. 安全问题，H5页面等于是所有东西都暴露给了用户，如果对安全要求比较高的，很多额外的安全机制都需要在服务端实现。

## 总结

web app一般是创业初期会重点考虑的方案，因为迭代非常快，而且创业初期的主要目标是验证模式的正确性，并不在于提供非常好的用户体验，只需要完成闭环即可。早年facebook曾经尝试过这种方案，最后因为用户体验的问题而宣布放弃。所以这个方案只能作为过渡方案，或者当App不可用时，作为降级方案使用。

# Hybrid App

通过市面上各种Hybrid框架，来做H5和Native的混合应用，或者通过JS Bridge来做到H5和Native之间的数据互通。

## 优点

1. 除了要承担苹果流程导致的成本以外，具备所有web app的优势
2. 能够访问本地数据、设备传感器等

## 缺点

1. 跟web app一样存在过度依赖网络环境的问题

2. 用户体验也很难做到很好
3. 安全性问题依旧存在
4. 大规模的数据交互很难实现，例如图片在本地处理后，将图片传递给H5

## 总结

Hybrid方案更加适合跟本地资源交互不是很多，然后主要以内容展示为主的App。在天猫App中，大量地采用了JS Bridge的方式来让H5跟Native做交互，因为天猫App是一个以内容展示为主的App，且营销活动多，周期短，比较适合Hybrid。

# React-Native

严格来说，React-Native应当放到Hybrid那一节去讲，单独拎出来的原因是Facebook自从放出React-Native之后，业界讨论得非常激烈。天猫的鬼道也做了非常多的关于React-Native的分享。

React-Native这个框架比较特殊，它展示View的方式依然是Native的View，然后也是可以通过URL的方式来动态生成View。而且，React-Native也提供了一个Bridge通道来做Javascript和Objective-C之间的交流，还是很贴心的。

然而研究了一下发现有一个比较坑的地方在于，解析JS要生成View时所需要的View，是要本地能够提供的。举个例子，比如你要有一个特定的MapView，并且要响应对应的delegate方法，在React-Native的环境下，你需要先在Native提供这个MapView，并且自己实现这些delegate方法，在实现完方法之后通过Bridge把数据回传给JS端，然后重新渲染。

在这种情况下我们就能发现，其实React-Native在使用View的时候，这些View是要经过本地定制的，并且将相关方法通过RCT\_EXPORT\_METHOD暴露给js，js端才能正常使用。在我看来，这里在一定程度上限制了动态部署时的灵活性，比如我们需要在某个点击事件中展示一个动画或者一个全新的view，由于本地没有实现这个事件或没有这个view，React-Native就显得捉襟见肘。

## 优点

1. 响应速度很快，只比Native慢一点，比webview快很多。
2. 能够做到一定程度上的动态部署

## 缺点

1. 组装页面的元素需要Native提供支持，一定程度上限制了动态部署的灵活性。

## 总结

由于React-Native框架中，因为View的展示和View的事件响应分属于不同的端，展示部分的描述在JS端，响应事件的监听和描述都在Native端，通过Native转发给JS端。所以，从做动态部署的角度上讲，React-Native只能动态部署新View，不能动态部署新View对应的事件。当然，React-Native本身提供了很多基础组件，然而这个问题仍然还是会限制动态部署的灵活性。因为我们在动态部署的时候，大部分情况下是希望View和事件响应一起改变的。

另外一个问题就在于，View的原型需要从Native中取，这个问题相较于上面一个问题倒是显得不那么严重，只是以后某个页面需要添加某个复杂的view的时候，需要从现有的组件中拼装罢了。

所以，React-Native事实上解决的是 如何不使用Objc/Swift来写iOS App的View 的问题，对于 如何通过不发版来给已发版的App更新功能 这样的问题，帮助有限。

# Lua Patch

大众点评的屠毅敏同学在基于wax (<https://github.com/probablycorey/wax>)的基础上写了waxPatch (<https://github.com/mmin18/WaxPatch>)，这个工具的主要原理是通过lua来针对objc的方法进行替换，由于lua本身是解释型语言，可以通过动态下载得到，因此具备了一定的动态部署能力。然而iOS系统原生并不提供lua的解释库，所以需要在打包时把lua的解释库编译进app。

## 优点

1. 能够通过下载脚本替换方法的方式，修改本地App的行为。
2. 执行效率较高

## 缺点

1. 对于替换功能来说，lua是很不错的选择。但如果要添加新内容，实际操作会很复杂
2. 很容易改错，小问题变成大问题

## 总结

lua的解决方案在一定程度上解决了动态部署的问题。实际操作时，一般不使用它来做新功能的动态部署，主要还是用于修复bug时代码的动态部署。实际操作时需要注意的另外一点是，真的很容易改错，尤其是你那个方法特别长的时候，所以改了之后要彻底回归测试一次。

# Javascript Patch

这个工作原理其实跟上面说的lua那套方案的工作原理一样，只不过是用javascript实现。而且最近新出了一个JSPatch (<https://github.com/bang590/JSPatch>)这个库，相当好用。

## 优点

1. 同Lua方案的优点
2. 打包时不用将解释器也编译进去，iOS自带JavaScript的解释器，只不过要从iOS7.0以后才支持。

## 缺点

1. 同Lua方案的缺点

## 总结

在对app打补丁的方案中，目前我更倾向于使用JSPatch的方案，在能够完成Lua做到的所有事情的同时，还不用编一个JS解释器进去，而且会javascript的人比会lua的人多，技术储备比较好做。

# JSON Descripted View

其实这个方案的原理是这样的：使用JSON来描述一个View应该有哪些元素，以及元素的位置，以及相关的属性，比如背景色，圆角等等。然后本地有一个解释器来把JSON描述的View生成出来。

这跟React-Native有点儿像，一个是JS转Native，一个是JSON转Native。但是同样有的问题就是事件处理的问题，在事件处理上，React-Native做得相对更好。因为JSON不能够描述事件逻辑，所以JSON生成的View所需要的事件处理都必须本地事先挂好。

## 优点

1. 能够自由生成View并动态部署

## 缺点

1. 天猫实际使用下来，发现还是存在一定的性能问题，不够快
2. 事件需要本地事先写好，无法动态部署事件

## 总结

其实JSON描述的View比React-Native的View有个好处就在于对于这个View而言，不需要本地也有一套对应的View，它可以依据JSON的描述来自己生成。然而对于事件的处理是它的硬伤，所以JSON描述View的方案，一般比较适用于换肤，或者固定事件不同样式的View，比如贴纸。

# 架构模式

其实我们要做到动态部署，至少要满足以下需求：

1. View和事件都要能够动态部署
2. 功能完整
3. 便于维护

我更加倾向于H5和Native以JSBridge的方式连接的方案进行动态部署，在cocoapods里面也有蛮多的JSBridge了。看了一圈之后，我还是选择写了一个CTJSBridge (<https://github.com/casatwy/CTJSBridge>)，来满足动态部署和后续维护的需求。关于这个JSBridge的使用中的任何问题和需求，都可以在评论区向我提出来。接下来的内容，会主要讨论以下这些问题：

1. 为什么不是React-Native或其它方案？
2. 采用什么样的架构模式才是使用JSBridge的最佳实践？

为什么不是React-Native或其他方案？

首先针对React-Native来做解释，前面已经分析到，React-Native有一个比较大的局限在于View需要本地提供。假设有一个页面的组件是跑马灯，如果本地没有对应的

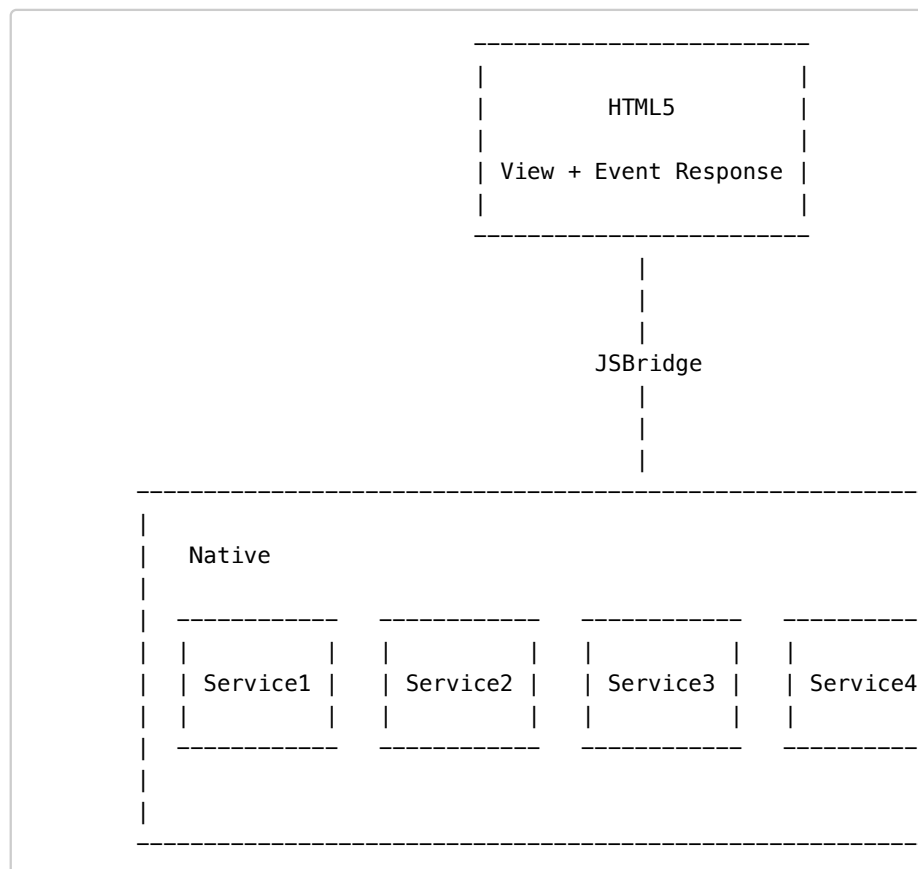
View，使用React-Native就显得很麻烦。然而同样的情况下，HTML5能够很好地实现这样的需求。这里存在一个这样的取舍 在性能和动态部署View及事件之间，选择哪一个？

我更加倾向于 能够动态部署View和事件，至少后者是能够完成需求的，性能再好，难以完成需求其实没什么意义。然而对于HTML5的Hybrid和纯HTML5的web app之间，也存在一个相同的取舍，但是还要额外考虑一个新的问题，纯HTML5能够使用到的设备提供的功能相对有限，JSBridge能够将部分设备的功能以Native API的方式交付给页面，因此在考虑这个问题之后，选择HTML5的Hybrid方案就显得理所应当了。

在诸多Hybrid方案中，除了JSBridge之外，其它的方案都显得相对过于沉重，对于动态部署来说，其实需要补充的软肋就是提供本地设备的功能，其它的反而显得较为累赘。

### 基于JSBridge的微服务架构模式

我开发了一个 `...`，基于JSBridge的微服务架构差不多是这样的：



解释一下这种架构背后的思想：

因为H5和Native之间能够通过JSBridge进行交互，然而JSBridge的一个特征是，只能H5主动发起调用。所以理所应当，被调用者为调用者提供服务。

另外一个想要处理的问题是，希望能够通过微服务架构，来把H5和Native各自的问题





主要原因是因为签名无法通过。因为Distribution的App只能加载相同证书打包的framework。在in house和develop模式下，可以使用相同证书既打包App又打包framework，所以测试的时候没有问题。但是在正式的distribution下，这种做法是行不通的。

所以就目前看来，基于动态库的动态部署方案是没办法做到的。

## 总结

我在文中针对业界常见的动态部署方案做了一些总结，并且提供了我自己认为的最佳解决方案以及对应的JSBridge实现。文中提到的方案我已经尽可能地做到了全面，如果还有什么我遗漏没写的，大家可以在评论区指出，我把它补上去。

---

有任何问题建议直接在评论区提问，这样后来的人如果有相同的问题，就能直接找到答案了。提问之前也可以先看看评论区有没有人问过类似问题了。

所有评论和问题我都会在第一时间回复，QQ上我是不回答问题的哈。

评论系统我用的是Disqus，不定期被墙。所以如果你看到文章下面没有加载出评论列表，翻个墙就有了。

本文遵守CC-BY。请保持转载后文章内容的完整，以及文章出处。本人保留所有版权相关权利。

我的博客拒绝挂任何广告，如果您觉得文章有价值，可以通过支付宝扫描下面的二维码捐助我。



---

## Comments

356 条评论

Casa Taloyum

 登录 Recommend 10 分享

按从新到旧排序



Join the discussion...



eric · 11天前

过了下所有的回复，咨询几个问题

1、博主在回复中提到：“CTPersistenceQueryCommand在这个框架里面是一个对象，这个对象hold住的sql string不是线程安全的，所以这个sql string会被别的线程修改，如果两个线程采用了同一个table对象的话CTPersistenceQueryCommand在多线程解析criteria的时候，尤其容易出现crash。”

我的疑惑：

(a) 是不是改成@property (atomic, strong, readonly)


NSMutableString \*sqlString;就没有这个问题了呢

(b) 不大明白为什么“多线程解析criteria的时候，尤其容易出现crash”

我将demo中的insert操作都改成findAllWithSQL操作（这样就涉及到criteria），但是还是没有crash。从直观上的理解真的就是在一个线程解析criteria的时候被其他线程影响了，顶多不就是操作失败吗，不会crash吧（还是说博主说的crash是业务处理时，数据操作失败，就主动抛出异常这种crash）

2、回复中有人提到

“我之前的代码也是用了一个对象，不过我改成每次操作就会新建一个对象，就不会出现crash了”

[查看更多](#)  · 回复 · 分享CasaTaloyum 管理员  eric · 11天前

1/a. 依旧会出现这个问题

1/b. 这个问题会导致crash，但是几率很小。当时我开发的时候本来也是认为不会crash的，但后来测试过程中我捕捉到了几次crash，研究了一下发现是多个操作请求改动到了同一个sqlString导致的，这时就会扔一个sqlString写入错误，因为写入的String的偏移发生了改变。由于criteria是分段写入sqlString，于是criteria场景下出现crash的几率比较大。

2/a. 业务工程师完全可以基于CTPersistence建立一套自己的持久层体系，此时他甚至也可以实现自己的DatabasePool，所以在dealloc里写好打扫的代码是个好习惯。

2/b. 同上，未来拓展的时候可能会用，是预留的口子。

3. 就是把上一个reformer的数据交给下一个reformer去使用，具体的实现取决于你自己reformer的实现。因为reformer只不过就是个protocol而已

4. api数据返回格式变动的频繁性取决于API工程师的职业道德和人品，本地数据格式变动的频繁性取决于自己的职业道德和人品，一般人在自己写的时候，都会对自己更加负责一点。另外，我遇到过太多傻逼了。所以我在设计方案时，第一个假设的前提

