iOS Socket 重构设计

xcvxvxc 2016-07-20 1924 阅读6分钟

原文链接：zeeyang.com

之前基于GCDAsyncSocket封装了一个Socket Manager类，但是由于业务复杂度的上升，之前设计的业务接口的数量逐渐增加，代理回调也随之增加，代理的使用也越来越麻烦，所以我们针对socket通信这块，进行了一次重构

这里有我们的新童鞋西兰花很大的功劳哈~

代码地址：GCDAsyncSocketManager

之前的设计方案可以看这里：socket重构前方案

针对老的设计，我们做出了以下几点修改方向：

0x00 拆分SocketManager

首先我们对SocketManager进行开刀，我们将socket相关的操作和业务相关的操作进行拆分，将业务相关的单独放到一个类里面完成，我们命名它为CommunicationManager

现在在SocketManager里面，我们只保留了 `服务器读写数据` 、 `断开连接` 、 `心跳` 、 `重连` 、 `GCDAsyncSocket回调设置`

在CommunicationManager里面，我们做所有业务的操作

0x01 业务接口改为通用接口

由于业务请求类型的不断增加，业务接口的数量也在不断增加，这样使得头文件一眼望不到底...自己看起来都很头疼，更别说是使用方了...

首先我们将不同的业务请求以枚举的方式列出来，方便外部调用的时候查看，并且最好在枚举后面加上注释，例如：

复制代码

```
/**
 *  业务类型
 */
typedef NS_ENUM(NSInteger, FIMRequestType) {
    FIMRequestType_Beat = 1,                        //心跳
    FIMRequestType_ConnectionAuthAppraisal,         //连接鉴权
   FIMRequestType_GetConversationsList,            //获取会话列表
    ...
};
```

这样我们就可以将业务接口用下面这一个通用的接口替换掉，只需要传 `type` 业务请求类型， `body` 请求体和 `callback` 回调

复制代码

```
/**
 *  向服务器发送数据
 *
 *  @param type    请求类型
 *  @param body    请求体
 */
- (void)socketWriteDataWithRequestType:(FIMRequestType)type
                           requestBody:(nonnull NSDictionary *)body
                            completion:(nullable SocketDidReadBlock)callback;
```

比如业务方可以如下使用：

复制代码

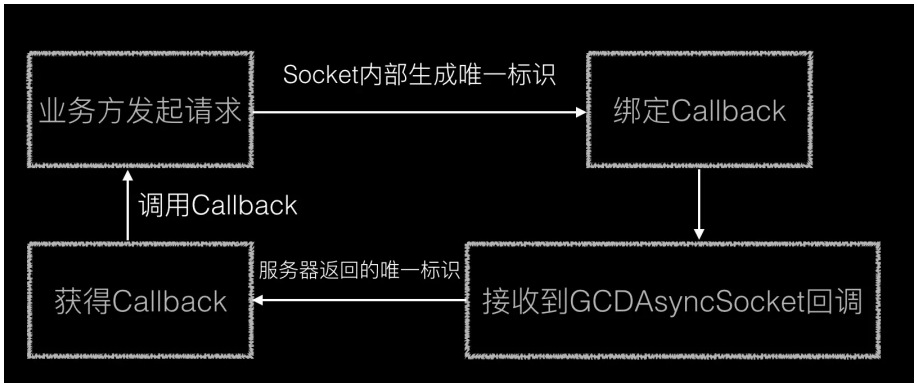
```
NSDictionary *requestBody = @{ @"limit": @(10), @"offset": @(0) };
[[FIMCommunicationManager sharedInstance]
socketWriteDataWithRequestType:FIMRequestType_GetConversationsList
                   requestBody:requestBody
                    completion:^(NSError *error, id data) {
                        // do something
```

```
7                   }];
```

0x02 告别Delegate，使用Blcok

前面也提到，之前会对不同的业务请求，设定相应的delegate回调，但是数量一多，使用起来那真的是槽糕，所以我们参考 `AFNetworking` 的做法，发起请求时将block与一个唯一标识进行绑定，同时将这个唯一标识放到请求里面发给服务器(服务器对该标识不做任何处理)，在等到GCDAsyncSocket回调回来的时候，我们通过服务器返回的这个标识，找到对应的block回调出去，这样对业务方来说，这个socket接口用起来其实和HTTP请求接口是一模一样的，将请求的上下文也关联起来了

如图：



具体实现：

发起请求时

复制代码

```
- (void)socketWriteDataWithRequestType:(FIMRequestType)type
                           requestBody:(nonnull NSDictionary *)body
                            completion:(nullable SocketDidReadBlock)callback {
  // ...

  // 生成唯一标识
  NSString *blockRequestID = [self createRequestID];
  if (callback) {
      // 将block和标识进行绑定，存到一个全局变量里面
      [self.requestsMap setObject:callback forKey:blockRequestID];
  }

  // ...
}
```

接收到GCDAsyncSocket回调时

复制代码

```
- (void)socket:(GCDAsyncSocket *)sock didReadData:(NSData *)data withTag:(long)tag {
  // ...

  // 根据服务器返回的标识得到相应的block
  SocketDidReadBlock didReadBlock = self.requestsMap[requestID];

  switch (requestType) {
        case FIMRequestType_ConnectionAuthAppraisal: {
            if (didReadBlock) {
                didReadBlock(nil, nil);
            }
        } break;
     // ...
     default: {
            // do something
        } break;
  }

  // ...
}
```

0x03 使用模拟服务器时间，来解决缓存消息保序问题

在socket模块里面，我们基于FMDB实现了一套缓存机制，但是聊天页面对数据库读写操作的场景非常复杂，而且我们对发送失败的消息也进行了缓存，如果使用msgID对消息进行保序，你要考虑发送成功和失败消息的

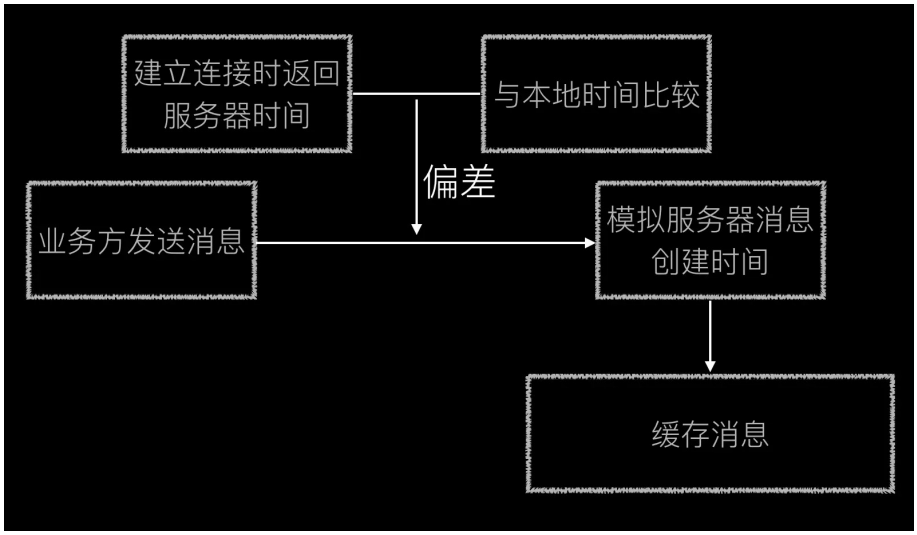
排序，以及重发消息之后的排序，等等场景，这样实现起来也会很让人头大

所以我们采用 `消息的创建时间` 来进行保序，这样不管消息是怎么操作的，从数据库里面读出来的数据，我们只需要根据创建时间来排下序返回给业务层，如果业务层对数据进行修改的时候，我们更新消息的创建时间，这样下次取出来的顺序和UI展示的顺序也还是一样的

那这个创建时间是由服务器生成的，而且消息发送成功之后，服务器也不会返回给我们这条消息的创建时间，而且失败的消息服务器那边是不会存的，所以就需要我们本地模拟服务器来生成这个时间

因为考虑到本地时间和服务器时间存在偏差，所以我们在socket建立连接成功之后，返回给我们服务器时间，我们拿到服务器时间之后和手机的本地时间做个比较，记录下这个偏差值，然后业务层在调用发送消息的接口时，socket内部模拟出服务器创建时间赋值给该消息，然后存到数据库里面，这样就可以基本保证数据库存储消息的顺序和服务器的顺序是一致的

如图：



0x04 监听网络状态来改变socket连接状态

我们对socket连接状态也做了微调，我们通过测试微信的连接，发现以下两点：

1、网络断开后，socket直接断开，显示"未连接"

2、有网但是socket连接不上时，socket会一直重连，重连n次后，休眠几秒后，再重连，如此循环

所以我们也对socket连接做了调整，用 `AFNetWorking` 库里面监测网络状态类 `AFNetworkReachabilityManager` （AFNetworkReachabilityManager原理），在无网时，判断如果socket正在连接或者已连接时，我们主动调用 `disconnect` 断开连接，如果有网，判断如果socket未连接，我们主动建立连接，建立连接不成功的情况时，我们走重连的流程，只是我们依旧保持了重连n次后，n次失败后不再重连了，这个是与微信不同的地方

0x05 使用FIMSocketModel

因为请求的数据结构基本一样，所以我们定义了FIMSocketModel类来方便对数据的转化，我们定义了几个必传的字段，以及可能请求不同所需的一些非必传字段，由于之前我们body体里面的内容是做了2次JSON转化处理的，所以业务层传入body内容时叫苦连天，FIMSocketModel也增加了 `- socketModelToJSONString` 方法，方便Socket内部转化成JSON处理，这样业务层只需要传一个字典进来，Socket内部就会处理好一切，使用起来一下就方便了~