**RetryNetWork网络框架**

**iOS网络模块优化（失败重发、缓存请求有网发送）**

我们有这样两个类, ZYRequest, ZYRequestManager

ZYRequest类用来处理公共的逻辑，Manager负责管理Request。在iOS开发中，很多时候会遇到多个Request集中发送的情况，比如说第一次进入App首页，需要请求骨架文件、首页Banner图片、展示Cell数据等等，如果这时候并发数太少，那些需要优先展示的数据请求可能会被次要的数据请求“阻塞”住。如果并发数太大，带宽有限的场景下，会增加请求的整体延迟。一般而言，在实际开发中，让请求的并发数限制在3~~5即可（也可以给每个请求设计优先级，然后在调度队列里面让优先级高的请求先出队列）。

　　请求的可靠性保障是个很容易被忽视的问题，见过的很多App的网络请求都是只进行一次请求，失败后直接给用户提示网络错误。比较好的做法，是将Request按业务分类：

　　　　第一类，关键核心业务，期望在任何条件下能百分百送达服务器。

　　　　第二类，重要的内容请求、数据展示，需要较高的成功率。

　　　　第三类，一般性内容请求，对成功率无要求。

理论上来说，需要我们应该尽量让每个请求的成功率都达到最高，但是客户端流量、带宽、电量、服务器压力等都是有限的资源，所以只能采取将关键性请求做高强度的可靠性保证

ZYRequest文件里面是所有发送一次请求所需要的数据，例如url\params\method\type等。

ZYRequestManager文件里面是进行request调度的主题逻辑，也没有进行复杂的算法，不按照优先级别，只是一个先入先出队列来进行调用的。里面有两个dispatch\_queue:

//这个串行队列用来控制任务有序的执行

@property (nonatomic, strong) dispatch\_queue\_t taskQueue;

//添加、删除队列，维护添加与删除request在同一个线程

@property (nonatomic, strong) dispatch\_queue\_t addDelQueue;

taskQueue主要是用来处理调度队列的，也就是requestQueue，让它在子线程进行循环查询、处理request，然后再并发进行网络请求，这样可以防止请求很多的情况下，卡住主线程。

addDelQeueu主要是用来处理requestQueue里面的requset增加与删除的。在添加和删除的时候，采用的方案都是串行+同步，主要是避免数据竞争。（因为在AFN发送request要删除requestQueue里面的request的时候，是并发状态）

在处理最大并发数的时候，使用的是dispatch\_semaphore\_t（信号量），设置最大并发数是4。

　需要注意的是，如何避免数据竞争，如何尽可能的不消耗主线程资源。

**针对百分百送达服务器的请求**

根据业务来说，这类请求应用的地方很多。类似于我们发微信发消息时，消息数据一旦从数据框中发出，从用户的角度感知这条消息是一定会到达对方的;在小说阅读App的书架收藏功能，理论上来说用户收藏一本书时，在用户感知角度，这本书一定会被收藏进入书架的等业务。如果网络环境差，网络模块会在后台悄悄重试，一段时间仍然无法成功的话，就直接通知用户发送失败了，但是即使失败，请求数据也会保存在本地，以便用户重新触发此条请求数据的发送。

对于这类请求的处理，第一步并不是直接发送，而是存入本地数据库中，一旦存入了数据库，即使是杀掉进程、断电、重启等极端操作，请求数据也依旧存在，我们只需要在App重启或者进入该业务界面时，还原请求数据到内存中，重新进行发送即可。代码阐释

typedef NS\_ENUM(NSInteger, ZYRequestReliability){

//如果没有发送成功，就放入调度队列再次发送

ZYRequestReliabilityRetry,

//必须要成功的请求，如果不成功就存入DB，然后在网络好的情况下继续发送，类似微信发消息

//需要注意的是，这类请求不需要回调的

//类似于发微信成功与否

//就是必定成功的请求，只需要在有网的状态下，必定成功

ZYRequestReliabilityStoreToDB,

//普通请求，成不成功不影响业务，不需要重新发送

//类似统计、后台拉取本地已有的配置之类的请求

ZYRequestReliabilityNormal

};

//存入数据库的唯一标示

@property (nonatomic, assign) int requestId;

/\*\*请求参数对\*/

@property (nonatomic, strong) NSDictionary \*params;

//如果是ZYRequestReliabilityStoreToDB类型

//第一时间先存储到数据库，然后再发送该请求，如果成功再从数据库中移除

//不成功再出发某机制从数据库中取出重新发送

if (request.reliability == ZYRequestReliabilityStoreToDB)

{

[[ZYRequestCache sharedInstance] saveRequestToRealm:request];

}

//在成功的时候移除realm数据库中的缓存

if (request.reliability == ZYRequestReliabilityStoreToDB)

{

[[ZYRequestCache sharedInstance] deleteRequestFromRealmWithRequestId:request.requestId];

}

//请求失败之后，根据约定的错误码判断是否需要再次请求

//这里，-1001是AFN的超时error

if (error.code == -1001 &&request.retryCount > 0)

{

[request reduceRetryCount];

[self queueAddRequest:request successBlock:successBlock failureBlock:failedBlock];

[self dealRequestQueue];

}

else //处理错误信息

{

failedBlock(error);

}

**如果是ZYRequestReliabilityStoreToDB请求，第一步是存入数据库。**

　第二步，将请求添加到调度队列里面，让调度队列调用AFN去处理该请求。在AFN的成功block里面，判断状态码，如果是真的成功状态，那么将数据库里面的请求移除掉，如果是失败状态，将重新请求的次数递减，再添加到调度队列末尾重新排队请求。

　我设计的是，最多重发三次请求。另外还有一个定时器，这个定时器会每隔60s，从数据库查询需要所有存储的请求，然后将它们尝试加入调度队列再次发送。这样的设计，即使App被kill，再次重启60s之后，也会把数据库中的请求拿出来进行发送。（只是一种思路，实际开发中会进入到具体业务才讲请求拿出来发送）

　通过上面的几个步骤，基本上可以极大的提高请求的可靠性，但是真的100%是无法实现的，如果用户卸载App，再下载，相关数据就无法恢复了。

**三、失败重发**

第二类请求的可靠性为ZYRequestReliabilityRetry，这类请求的例子可以是我们App启动时用户看到的首页，首页的内容从服务器获取，如果第一次请求就失败体验较差，这种场景下我们应该允许请求有机会多试几次，增加一个retryCount即可。

/\*\*

请求没发送成功，重新发送的次数

\*/

@property (nonatomic, assign, readonly) int retryCount;

　　　　在Manager里面，有一个调度队列：

1

@property (nonatomic, strong) NSMutableArray \*requestQueue;

每次将请求加入这个队列，然后在AFN发送完成回调之后，如果失败就进行重发，实际开发时，需要自行处理失败重发的错误码判断：

//请求失败之后，根据约定的错误码判断是否需要再次请求

//这里，-1001是AFN的超时error

if (error.code == -1001 &&request.retryCount > 0)

{

[request reduceRetryCount];

[self queueAddRequest:request successBlock:successBlock failureBlock:failedBlock];

[self dealRequestQueue];

}

在这里，是设置如果超时才进行重发请求，也可以将这个判断去掉，只要retryCount大于0即进行重发。一般开发的时候会和后台确定一些错误码，根据错误码的类型判断是否需要重发会更合理些。

一般3次的重试基本可以排除网络抖动的情况。三次失败之后即可认为请求失败，通过产品交互告知用户。

第三类请求的重要性最低，比如进入Controller的UV采集打点、收集数据等。这类请求只需要做一次，即使失败也不会对App体验产生什么负面影响。

为什么需要设计框架？所有请求直接利用AFN并发发送不行么？（不行，因为网络带宽是有限的，这样做会导致数据返回整体慢上很多，而且，一个网络请求的超时时间是一定的，一次性并发很可能造成本来可以发送成功的请求超时）

这一次的设计，并没有依赖、优先级等，只进行了重发、最大并发设计。思路是一样的，无非就是一个调度队列进行request的处理，这个队列的出队规则可以是按优先级高低来进行，当然得自己封装优先队列的算法。这里是最简单的先进先出队列，每次请求失败，要进行重发的话，就把请求丢到队列末尾。额，理论上来说，请求无限多的情况下，调度队列会是个死循环，这样会造成主线程卡顿，所以把它放到子线程来处理。在并发发送请求之下，不做处理的话，会并发的删除调度队列里面的request;