FBPromise

jayhuan

Promise介绍

- Promise 是异步编程的一种解决方案，比传统的解决方案（回调函数和事件）更合理和更强大。
- Promise保存着某个未来才会结束的事件（通常是一个异步操作）的结果。

常见的异步操作场景

1. 单次；
2. 多次，串行，如获取用户信息——获取用户好友——获取好友头像；
3. 多次，并行，所有结果，比如同时下载多张图片；
4. 多次，并行，1个结果；
5. 串并行操作结合。

异步操作设计

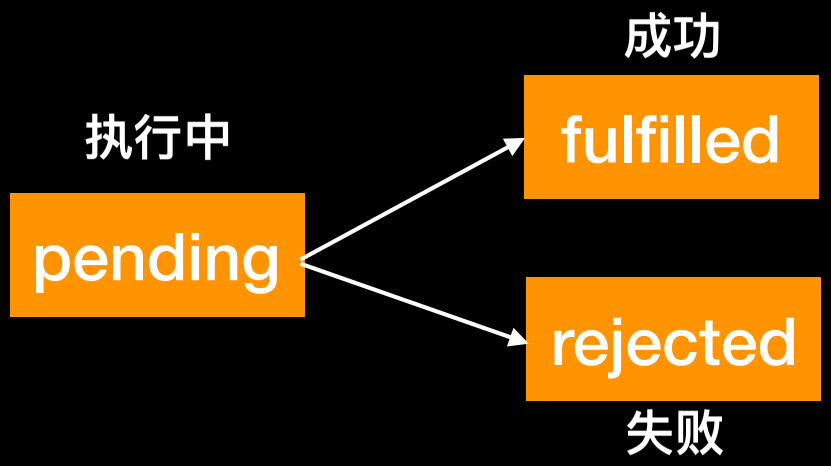
1. 状态变化（执行、暂停、取消、成功、失败）
2. 执行过程
3. 结果（success, failed）
4. 回调

Promise特点

- Promise对象的状态不受外界影响。Promise代表一个异步操作，有3种状态：pending（进行中）、fulfilled（成功）和rejected（失败）。只有异步操作的结果，可以决定当前状态，其它操作无法改变这个状态。
- Promise对象的状态一旦改变，就不会再变，任何时候都可以拿到这个结果。状态改变只有2种可能：从pending变为fulfilled或者从pending变为rejected。

FBPromise实现

```objc
/** All states a promise can be in. */
typedef NS_ENUM(NSInteger, FBLPromiseState) {
  FBLPromiseStatePending = 0,
  FBLPromiseStateFulfilled,
  FBLPromiseStateRejected,
};
```



1、创建立刻执行，不能暂停或者取消；
2、执行完毕，状态不再变化

FBPromise实现

```objc
@implementation FBLPromise {
  /** Current state of the promise. */
  FBLPromiseState _state;  状态
  /**
   Set of arbitrary objects to keep strongly while the promise is pending.
   Becomes nil after the promise has been resolved.
   */
  NSMutableSet *__nullable _pendingObjects;
  /**
   Value to fulfill the promise with.
   Can be nil if the promise is still pending, was resolved with nil or aft
   */
  id __nullable _value;  成功的结果
  /**
   Error to reject the promise with.
   Can be nil if the promise is still pending or after it has been fulfille
   */
  NSError *__nullable _error;  错误
  /** List of observers to notify when the promise gets resolved. */
  NSMutableArray<FBLPromiseObserver> *_observers;  回调
}
```

FBPromise实现

```
const promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

1、Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject，也是两个函数；
2、resolve函数作用：将Promise对象的状态从“执行中”变为“成功”（即从pending变为 resolved），在异步操作成功时调用，并保存异步操作的结果；
3、reject函数作用：将Promise对象的状态从“执行中”变为“失败”（即从pending变为 rejected），在异步操作失败时调用，并保存异步操作的错误。

FBPromise实现

```
typedef void (^FBLPromiseFulfillBlock)(Value __nullable value); resolve函数
typedef void (^FBLPromiseRejectBlock)(NSError *error); reject函数
typedef void (^FBLPromiseAsyncWorkBlock)(FBLPromiseFulfillBlock fulfill,
                                         FBLPromiseRejectBlock reject);
                          构造参数函数
```

```
/**
 Creates a pending promise and executes `work` block asynchronously.

 @param work A block to perform any operations needed to resolve the promise.
 @return A new pending promise.
 */
+ (instancetype)async:(FBLPromiseAsyncWorkBlock)work;
```

```objc
+ (instancetype)async:(FBLPromiseAsyncWorkBlock)work {
  return [self onQueue:self.defaultDispatchQueue async:work];
}

+ (instancetype)onQueue:(dispatch_queue_t)queue async:(FBLPromiseAsyncWorkBlock)work {
  NSParameterAssert(queue);
  NSParameterAssert(work);

  FBLPromise *promise = [[FBLPromise alloc] initPending];
  dispatch_group_async(FBLPromise.dispatchGroup, queue, ^{
    work(
        ^(id __nullable value) {
          if ([value isKindOfClass:[FBLPromise class]]) {
            [(FBLPromise *)value observeOnQueue:queue
                fulfill:^(id __nullable value) {
                  [promise fulfill:value];
                }
                reject:^(NSError *error) {
                  [promise reject:error];
                }];
          } else {
            [promise fulfill:value];
          }
        },
        ^(NSError *error) {
          [promise reject:error];
        });
  });
  return promise;
}
```

可以嵌套执行promise

```objc
- (void)fulfill:(nullable id)value {
  if ([value isKindOfClass:[NSError class]]) {
    [self reject:(NSError *)value];
  } else {
    @synchronized(self) {
      if (_state == FBLPromiseStatePending) {
        _state = FBLPromiseStateFulfilled; 修改状态
        _value = value;   保存结果
        _pendingObjects = nil;
        for (FBLPromiseObserver observer in _observers) {
          observer(_state, _value);  执行回调
        }
        _observers = nil;
        dispatch_group_leave(FBLPromise.dispatchGroup);
      }
    }
  }
}
```

```objc
- (void)reject:(NSError *)error {
  NSAssert([error isKindOfClass:[NSError class]], @"Invalid error type.");

  if (![error isKindOfClass:[NSError class]]) {
    // Give up on invalid error type in Release mode.
    @throw error;  // NOLINT
  }
  @synchronized(self) {
    if (_state == FBLPromiseStatePending) {
      _state = FBLPromiseStateRejected;   修改状态
      _error = error;  保存结果
      _pendingObjects = nil;
      for (FBLPromiseObserver observer in _observers) {
        observer(_state, _error);       执行回调
      }
      _observers = nil;
      dispatch_group_leave(FBLPromise.dispatchGroup);
    }
  }
}
```

then方法

```
promise.then(function(value) {
  // success
}, function(error) {
  // failure
});
```

then方法接受两个函数作为参数，分别在promise对象切换到resolved状态和rejected状态时回调。

then方法

```objc
@interface FBLPromise<Value>(ThenAdditions)

typedef id __nullable (^FBLPromiseThenWorkBlock)(Value __nullable value);

/**
 Creates a pending promise which eventually gets resolved with resolution returned from `work`
 block: either value, error or another promise. The `work` block is executed asynchronously only
 when the receiver is fulfilled. If receiver is rejected, the returned promise is also rejected with
 the same error.

 @param work A block to handle the value that receiver was fulfilled with.
 @return A new pending promise to be resolved with resolution returned from the `work` block.
 */
- (FBLPromise *)then:(FBLPromiseThenWorkBlock)work;
```

then方法

```
@implementation FBLPromise (ThenAdditions)

- (FBLPromise *)then:(FBLPromiseThenWorkBlock)work {
  return [self onQueue:FBLPromise.defaultDispatchQueue then:work];
}

- (FBLPromise *)onQueue:(dispatch_queue_t)queue then:(FBLPromiseThenWorkBlock)work {
  NSParameterAssert(queue);
  NSParameterAssert(work);

  return [self chainOnQueue:queue chainedFulfill:work chainedReject:nil];
}

@end
```

```objc
- (FBLPromise *)chainOnQueue:(dispatch_queue_t)queue
                chainedFulfill:(FBLPromiseChainedFulfillBlock)chainedFulfill
                 chainedReject:(FBLPromiseChainedRejectBlock)chainedReject {
  NSParameterAssert(queue);

  FBLPromise *promise = [[FBLPromise alloc] initPending];
  __auto_type resolver = ^(id __nullable value) {
    if ([value isKindOfClass:[FBLPromise class]]) {
      [(FBLPromise *)value observeOnQueue:queue
          fulfill:^(id __nullable value) {
            [promise fulfill:value];
          }
          reject:^(NSError *error) {
            [promise reject:error];
          }];
    } else {
      [promise fulfill:value];
    }
  };
  [self observeOnQueue:queue
      fulfill:^(id __nullable value) {
        value = chainedFulfill ? chainedFulfill(value) : value;
        resolver(value);
      }
      reject:^(NSError *error) {
        id value = chainedReject ? chainedReject(error) : error;
        resolver(value);
      }];
  return promise;
}
```

创建一个新的promise

给当前的promise添加回调

promise常见方法

1. async，单次异步操作
2. then，可以实现串行操作；
3. all，可以实现并行操作，并且只有所有操作都成功才成功，否则失败；
4. any，可以实现并行操作，保存每个操作的结果；
5. race，可以实现并行操作，只要有一个结束就回调结果；
6. reduce，串行操作，按顺序返回结果；
7. await、delay、recover，wrap等其它方法。

promise优点

1. 使用简洁方便；
2. 支持OC和Swift互相操作；
3. 轻量级，与GCD性能开销基本一致；
4. 灵活，提供多种场景下的使用方法；
5. 安全，回调都是基于GCD，没有循环引用；
6. 都已通过单元测试。

promise缺点

1. 首先，无法取消Promise，一旦创建它就会立即执行，无法取消。

2. 当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）

QNAsyncTask

1. 支持取消操作；
2. 支持的场景少，串行操作书写比较麻烦；
3. 多处使用锁，实现比较复杂
4. 对网络操作支持方便

QNListLoader逻辑

首先加载本地缓存，再加载网络数据，串行操作。

QNAsyncTask实现

```
QNAsyncSerialTaskBuildBlock localBuildBlock = nil;
QNAsyncSerialTaskBuildBlock remoteBuildBlock = nil;
QNAsyncTaskProgressMonitorBlock progressMonitorBlock = nil;
QNListRemoteResultsAggregationBlock resultGenerateBlock = nil;
```

```
localBuildBlock = ^QNAsyncTask<QNListLoaderSuccessResponse *> *(
                        QNAsyncSerialTaskSubState<NSNull *> *previousSubState, BOOL
                            *refOptionalTask) {
    return [[QNAsyncBlockTask<QNListLoaderSuccessResponse *> alloc]
            initWithBlock:^QNListLoaderSuccessResponse *(QNAsyncTask *task, NSError *__strong
                *refError) {
                @strongify(self);
                QNListLoaderLocalcCacheData *result = [self.listDataManager
                    loadDataFromDBWithDirection:direction];
                AS_VAR(listItems, NSArray, result.first);
                AS_VAR(indexData, KBChannelListIndexData, result.second);
                KBListStickedData *stickData = [self.listDataManager getListStickedData];
                NSArray *stickListItems = nil;
```

QNAsyncTask实现

```
progressMonitorBlock = ^(QNAsyncSerialTaskSubState *subTaskState) {
    @strongify(self);
    if ([subTaskState.taskID isEqual:kQNListLoaderTaskIDOfLoadLocalData] && subTaskState.result) {
        QNListLoaderSuccessResponse *result = AS(QNListLoaderSuccessResponse, subTaskState.result);
        if (CHECK_VALID_ARRAY(result.listItems)) {
            QN_I(@"%@ load %@ items from LOCAL", self.channelID, @(result.listItems.count));
            if (cachePolicy != kQNListLoaderPolicyCacheOnly) {
                if (self.finishedBlock) {
                    self.finishedBlock(result);
                }
            }
        }
    }
};

// #2 append a remote data task
remoteBuildBlock = ^QNAsyncTask<NSDictionary *> *(QNAsyncSerialTaskSubState<NSObject *>
    *previousSubState,
                                                  BOOL *refOptionalTask) {
    @strongify(self);
    QNAsyncTask<NSDictionary *> *remoteTask = [self _taskForLoadRemoteServerData:urlString
                                                                       direction:direction
                                                                      postParams:postKeyValues
                                                                     queryParams:queryKeyValues];
    return remoteTask;
};

resultGenerateBlock = ^QNListLoaderSuccessResponse *(QNAsyncSerialTask<QNListLoaderSuccessResponse *>
    *parentTask,
```

promise实现

```
FBLPromise<QNListLoaderSuccessResponse *> *promiseLocal = nil;
FBLPromise<NSString *> *promiseLocalHandle = nil;
FBLPromise<QNListLoaderSuccessResponse *> *promiseRemote = nil;
FBLPromise<NSString *> *promiseRemoteHandle = nil;
```

```
promiseLocal = [FBLPromise async:^(FBLPromiseFulfillBlock  _Nonnull fulfill, FBLPromiseRejectBlock
    _Nonnull reject) {
    QNListLoaderSuccessResponse *response; // 加载本地缓存数据
    fulfill(response);
}];

promiseLocalHandle = [promiseLocal then:^id _Nullable(QNListLoaderSuccessResponse * _Nullable
    response) {
    QN_INVOKE_BLOCK_PARAMS(self.finishedBlock, response);
    return @"promiseLocalHandle";
}];
```

promise实现

```
promiseRemote = [promiseLocalHandle then:^id _Nullable(NSString * _Nullable value) {

    return [FBLPromise async:^(FBLPromiseFulfillBlock  _Nonnull fulfill, FBLPromiseRejectBlock
        _Nonnull reject) {
        QNAsyncTask<NSDictionary *> *remoteTask;// 网络请求task
        [remoteTask startWithSuccess:^(QNAsyncTask *task, NSObject *responseObject) {
            QNListLoaderSuccessResponse *response;
            fulfill(response);
        } fail:^(QNAsyncTask *task, NSError *error) {
            reject(error);
        }];
    }];
}];

promiseRemoteHandle = [promiseRemote then:^id _Nullable(QNListLoaderSuccessResponse * _Nullable
    response) {
    QN_INVOKE_BLOCK_PARAMS(self.finishedBlock, response);
    return @"promiseRemoteHandle";
}];
```

promise实例

获取用户信息——获取用户联系人列表——获取联系人的头像图片

```
- (FBLPromise<NSArray<UIImage *> *> *)getCurrentUserContactsAvatars {
  return [[[MyClient getCurrentUser] then:^id(MyUser *currentUser) {
    return [MyClient getContactsForUser:currentUser];
  }] then:^id(NSArray<MyContact *> *contacts) {
    return [FBLPromise all:[contacts fbl_map:^id(MyContact *contact) {
      return [MyClient getAvatarForContact:contact];
    }]];
  }];
```

串并联结合