## RunLoop终极解析：输入源，定时源，观察者，线程间通信，端口间通信，NSPort，NSMessagePort，NSMachPort，NSPortMessag

OSX / iOS 系统中，提供了两个这样的对象：**NSRunLoop** 和 **CFRunLoopRef**。

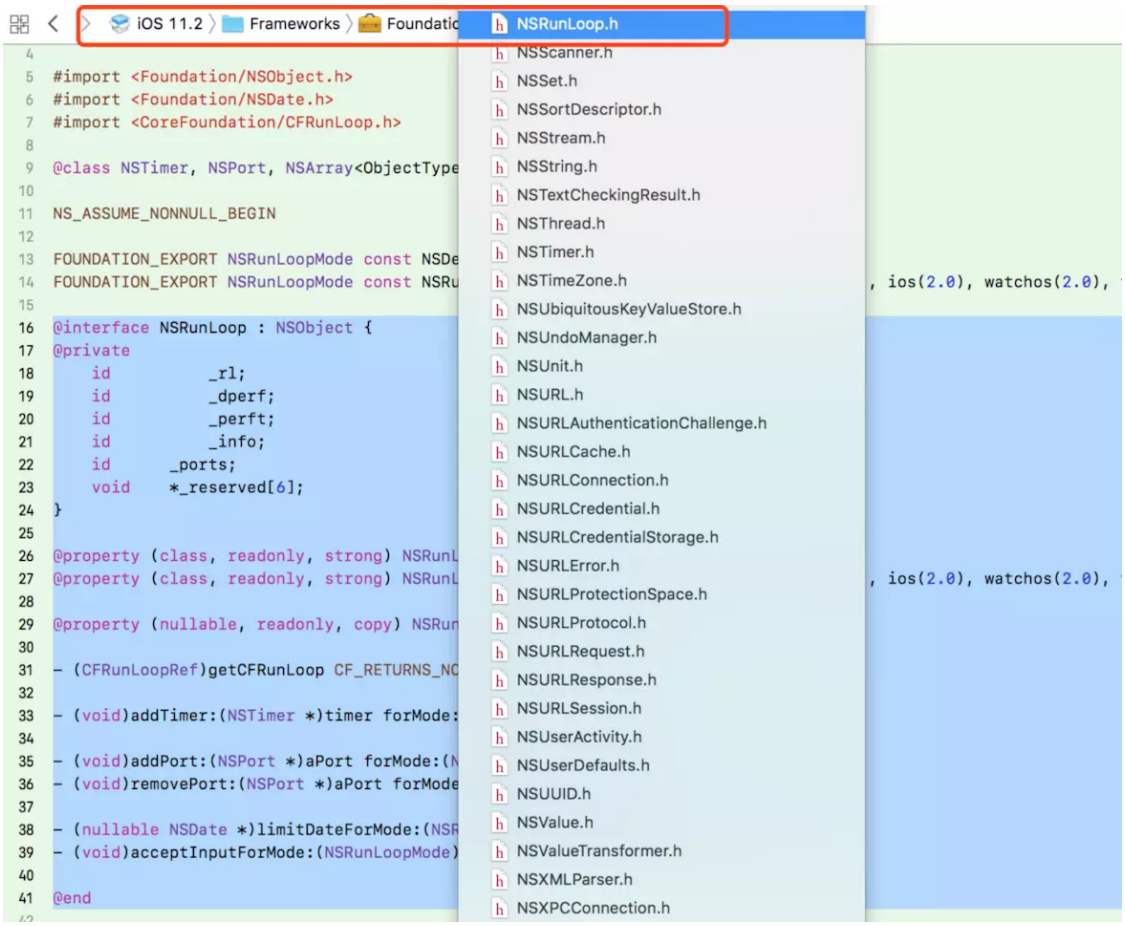
- **CFRunLoopRef** 是在 **CoreFoundation** 框架内的，它提供了纯 C 函数的 API，所有这些 API 都是线程安全的。
- **NSRunLoop** 是基于 **CFRunLoopRef** 的封装，提供了面向对象的 API，但是这些 API 不是线程安全的。

# 1. 如何查看RunLoop源代码

---

### 1.1 NSRunLoop源代码

NSRunLoop是Foundation框架里面的一个类，它的头文件可以在工程里面这样查看：




至于它的实现文件，暂时没有找到公开的资料。

### 1.2 CFRunLoopRef源代码

CFRunLoopRef 的代码是开源的，你可以在这里 opensource.apple.com/tarballs/CF... 下载到整个 CoreFoundation 的源码。为了方便跟踪和查看，你可以新建一个 Xcode 工程，把这堆源码拖进去看。

### 更多苹果源代码下载

苹果公开的源代码在这里可以下载，opensource.apple.com/tarballs/

例如，其中，有两个比较常见需要学习源码的下载地址：

- runtime的源代码在opensource.apple.com/tarballs/ob...
- runloop(其实是整个 CoreFoundation)的源代码在 opensource.apple.com/tarballs/CF...

当然，如果你想在github上在线查看源代码，可以点这里：runtime，runloop

# 2. 简析RunLoop源代码

## 2.1 Foundation相关Runloop的源码

### NSRunLoop

```
@interface NSRunLoop : NSObject {
@private
    id          _rl;
    id          _dperf;
    id          _perft;
    id          _info;
    id      _ports;
    void    *_reserved[6];
}

@property (class, readonly, strong) NSRunLoop *currentRunLoop;
@property (class, readonly, strong) NSRunLoop *mainRunLoop
API_AVAILABLE(macos(10.5), ios(2.0), watchos(2.0), tvos(9.0));

@property (nullable, readonly, copy) NSRunLoopMode currentMode;

- (CFRunLoopRef)getCFRunLoop CF_RETURNS_NOT_RETAINED;

- (void)addTimer:(NSTimer *)timer forMode:(NSRunLoopMode)mode;

- (void)addPort:(NSPort *)aPort forMode:(NSRunLoopMode)mode;
- (void)removePort:(NSPort *)aPort forMode:(NSRunLoopMode)mode;

- (nullable NSDate *)limitDateForMode:(NSRunLoopMode)mode;
- (void)acceptInputForMode:(NSRunLoopMode)mode beforeDate:(NSDate
*)limitDate;

@end
```

## 2.2 Core Foundation相关Runloop的源码

### __CFRunLoop

```
struct __CFRunLoop {
    CFRuntimeBase _base;
    pthread_mutex_t _lock;          /* locked for accessing mode
list */
    __CFPort _wakeUpPort;           // used for CFRunLoopWakeUp
    Boolean _unused;
    volatile _per_run_data *_perRunData;              // reset for
runs of the run loop
    pthread_t _pthread;
    uint32_t _winthread;
    CFMutableSetRef _commonModes;
    CFMutableSetRef _commonModeItems;
    CFRunLoopModeRef _currentMode;
    CFMutableSetRef _modes;
    struct _block_item *_blocks_head;
    struct _block_item *_blocks_tail;
    CFTypeRef _counterpart;
};
```

### __CFRunLoopMode

```
struct __CFRunLoopMode {
    CFRuntimeBase _base;
    pthread_mutex_t _lock;  /* must have the run loop locked before
locking this */
    CFStringRef _name;
    Boolean _stopped;
    char _padding[3];
    CFMutableSetRef _sources0;
    CFMutableSetRef _sources1;
    CFMutableArrayRef _observers;
    CFMutableArrayRef _timers;
    CFMutableDictionaryRef _portToV1SourceMap;
    __CFPortSet _portSet;
    CFIndex _observerMask;
#if USE_DISPATCH_SOURCE_FOR_TIMERS
    dispatch_source_t _timerSource;
    dispatch_queue_t _queue;
    Boolean _timerFired; // set to true by the source when a timer
has fired
    Boolean _dispatchTimerArmed;
#endif
#if USE_MK_TIMER_TOO
    mach_port_t _timerPort;
```

```
    Boolean _mkTimerArmed;
#endif
#if DEPLOYMENT_TARGET_WINDOWS
    DWORD _msgQMask;
    void (*_msgPump)(void);
#endif
    uint64_t _timerSoftDeadline; /* TSR */
    uint64_t _timerHardDeadline; /* TSR */
};
```

### __CFRunLoopSource

```
struct __CFRunLoopSource {
    CFRuntimeBase _base;
    uint32_t _bits;
    pthread_mutex_t _lock;
    CFIndex _order;         /* immutable */
    CFMutableBagRef _runLoops;
    union {
    CFRunLoopSourceContext version0;    /* immutable, except
invalidation */
        CFRunLoopSourceContext1 version1;   /* immutable, except
invalidation */
    } _context;
};
```

### __CFRunLoopObserver

```
struct __CFRunLoopObserver {
    CFRuntimeBase _base;
    pthread_mutex_t _lock;
    CFRunLoopRef _runLoop;
    CFIndex _rlCount;
    CFOptionFlags _activities;      /* immutable */
    CFIndex _order;         /* immutable */
    CFRunLoopObserverCallBack _callout; /* immutable */
    CFRunLoopObserverContext _context;  /* immutable, except
invalidation */
};
```

__CFRunLoopTimer

```
struct __CFRunLoopTimer {
    CFRuntimeBase _base;
    uint16_t _bits;
    pthread_mutex_t _lock;
    CFRunLoopRef _runLoop;
    CFMutableSetRef _rlModes;
    CFAbsoluteTime _nextFireDate;
    CFTimeInterval _interval;       /* immutable */
    CFTimeInterval _tolerance;          /* mutable */
    uint64_t _fireTSR;          /* TSR units */
    CFIndex _order;         /* immutable */
    CFRunLoopTimerCallBack _callout;    /* immutable */
    CFRunLoopTimerContext _context; /* immutable, except
invalidation */
};
```

# 3. Runloop的基本操作

## 3.1 如何创建线程对应的 Runloop?

苹果不允许直接创建 **RunLoop**，它只提供了两个自动获取的函数：**CFRunLoopGetMain()** 和 **CFRunLoopGetCurrent()**。 当然，Foudation 框架也有对应的API。

### Foundation

```
NSRunLoop *mainRunloop = [NSRunLoop mainRunLoop]; // 获得主线程对应的
runloop对象
NSRunLoop *currentRunloop = [NSRunLoop currentRunLoop]; // 获得当前线
程对应的runloop对象
```

### Core Foundation

```
CFRunLoopRef maiRunloop = CFRunLoopGetMain(); // 获得主线程对应的
```

```
runloop对象
CFRunLoopRef maiRunloop = CFRunLoopGetCurrent(); // 获得当前线程对应的
runloop对象
```

## 3.2 底层如何获取RunLoop对象?

获得runloop实现 (创建runloop)

```
CFRunLoopRef CFRunLoopGetMain(void) {
    CHECK_FOR_FORK();
    static CFRunLoopRef __main = NULL; // no retain needed
    if (!__main) __main = _CFRunLoopGet0(pthread_main_thread_np()); //
no CAS needed
    return __main;
}

CFRunLoopRef CFRunLoopGetCurrent(void) {
    CHECK_FOR_FORK();
    CFRunLoopRef rl = (CFRunLoopRef)_CFGetTSD(__CFTSDKeyRunLoop);
    if (rl) return rl;
    return _CFRunLoopGet0(pthread_self());
}

// should only be called by Foundation
// t==0 is a synonym for "main thread" that always works
CF_EXPORT CFRunLoopRef _CFRunLoopGet0(pthread_t t) {
    if (pthread_equal(t, kNilPthreadT)) {
    t = pthread_main_thread_np();
    }
    __CFSpinLock(&loopsLock);
    if (!__CFRunLoops) {
        __CFSpinUnlock(&loopsLock);
    CFMutableDictionaryRef dict =
CFDictionaryCreateMutable(kCFAllocatorSystemDefault, 0, NULL,
&kCFTypeDictionaryValueCallBacks);
    CFRunLoopRef mainLoop =
__CFRunLoopCreate(pthread_main_thread_np());
    CFDictionarySetValue(dict,
pthreadPointer(pthread_main_thread_np()), mainLoop);
    if (!OSAtomicCompareAndSwapPtrBarrier(NULL, dict, (void * volatile
*)&__CFRunLoops)) {
        CFRelease(dict);
    }
    CFRelease(mainLoop);
```

```
        __CFSpinLock(&loopsLock);
    }
    CFRunLoopRef loop =
(CFRunLoopRef)CFDictionaryGetValue(__CFRunLoops, pthreadPointer(t));
    __CFSpinUnlock(&loopsLock);
    if (!loop) {
    CFRunLoopRef newLoop = __CFRunLoopCreate(t);
        __CFSpinLock(&loopsLock);
    loop = (CFRunLoopRef)CFDictionaryGetValue(__CFRunLoops,
pthreadPointer(t));
    if (!loop) {
        CFDictionarySetValue(__CFRunLoops, pthreadPointer(t),
newLoop);
        loop = newLoop;
    }
        // don't release run loops inside the loopsLock, because
CFRunLoopDeallocate may end up taking it
        __CFSpinUnlock(&loopsLock);
    CFRelease(newLoop);
    }
    if (pthread_equal(t, pthread_self())) {
        _CFSetTSD(__CFTSDKeyRunLoop, (void *)loop, NULL);
        if (0 == _CFGetTSD(__CFTSDKeyRunLoopCntr)) {
            _CFSetTSD(__CFTSDKeyRunLoopCntr, (void *)
(PTHREAD_DESTRUCTOR_ITERATIONS-1), (void (*)(void
*))__CFFinalizeRunLoop);
        }
    }
    return loop;
}
```

- 【由上源码可得】：RunLoop 和 线程关系
  - 1.每条线程都有唯一的一个与之对应的RunLoop对象。
  - 2.主线程的RunLoop已经自动创建，子线程的RunLoop需要主动创建。
  - 3.RunLoop在第一次获取时创建，在线程结束时销毁。

Runloop 对象是利用字典来进行存储，而且 Key:线程 -- Value:线程对应的 runloop。

## 3.3 RunLoop对象如何运行?

**① CFRunLoopRun**

RunLoop 其实内部就是do-while循环，在这个循环内部不断地处理各种任务（`比如Source、Timer、Observer`），通过判断result的值实现的。所以 可以看成是一个死循环。如果没有RunLoop，UIApplicationMain 函数执行完毕之后将直接返回，就是说程序一启动然后就结束；

```
void CFRunLoopRun(void) {   /* DOES CALLOUT */
    int32_t result;
    do {
        result = CFRunLoopRunSpecific(CFRunLoopGetCurrent(),
kCFRunLoopDefaultMode, 1.0e10, false);
        CHECK_FOR_FORK();
    } while (kCFRunLoopRunStopped != result &&
kCFRunLoopRunFinished != result);
}
```

源码得知：

1. kCFRunLoopDefaultMode，默认情况下，runLoop是在这个mode下运行的，
2. runLoop的运行主体是一个do..while循环，除非停止或者结束，否则runLoop会一直运行下去

**② CFRunLoopRunInMode**

```
SInt32 CFRunLoopRunInMode(CFStringRef modeName, CFTimeInterval
seconds, Boolean returnAfterSourceHandled) {     /* DOES CALLOUT */
    CHECK_FOR_FORK();
    return CFRunLoopRunSpecific(CFRunLoopGetCurrent(), modeName,
seconds, returnAfterSourceHandled);
}
```

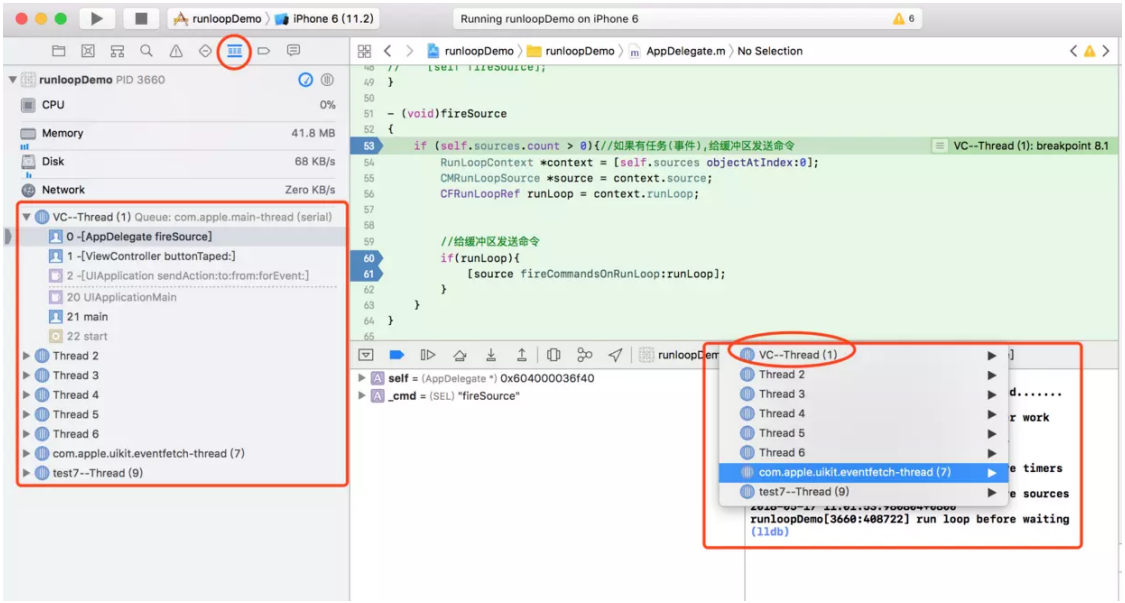
该方法，可以设置runLoop运行在哪个mode下modeName，超时时间seconds，以及是否处理完事件就返回returnAfterSourceHandled。

这两个方法实际调用的是同一个方法CFRunLoopRunSpecific，其返回是一个SInt32类型的值，根据返回值，来决定runLoop的运行状况。

# 4. RunLoop与线程

首先，iOS 开发中能遇到两个线程对象: **pthread_t** 和 **NSThread**。过去苹果有份文档标明了 **NSThread** 只是 **pthread_t** 的封装，但那份文档已经失效了，现在它们也有可能都是直接包装自最底层的 **mach thread**。苹果并没有提供这两个对象相互转换的接口，但不管怎么样，可以肯定的是 **pthread_t** 和 **NSThread** 是一一对应的。比如，你可以通过 `pthread_main_np()` 或 `[NSThread mainThread]` 来获取 **主线程**；也可以通过 `pthread_self()` 或 `[NSThread currentThread]` 来获取 **当前线程**。**CFRunLoop** 是基于 **pthread** 来管理的。

苹果不允许直接创建 **RunLoop**，它只提供了两个自动获取的函数：`CFRunLoopGetMain()` 和 `CFRunLoopGetCurrent()`。从上面的代码（第3.2节）可以看出，线程和 **RunLoop** 之间是一一对应的，其关系是保存在一个全局的 Dictionary 里。线程刚创建时并没有 **RunLoop**，如果你不主动获取，那它一直都不会有。**RunLoop** 的创建是发生在第一次获取时，**RunLoop** 的销毁是发生在线程结束时。你只能在一个线程的内部获取其 **RunLoop**（主线程除外）。



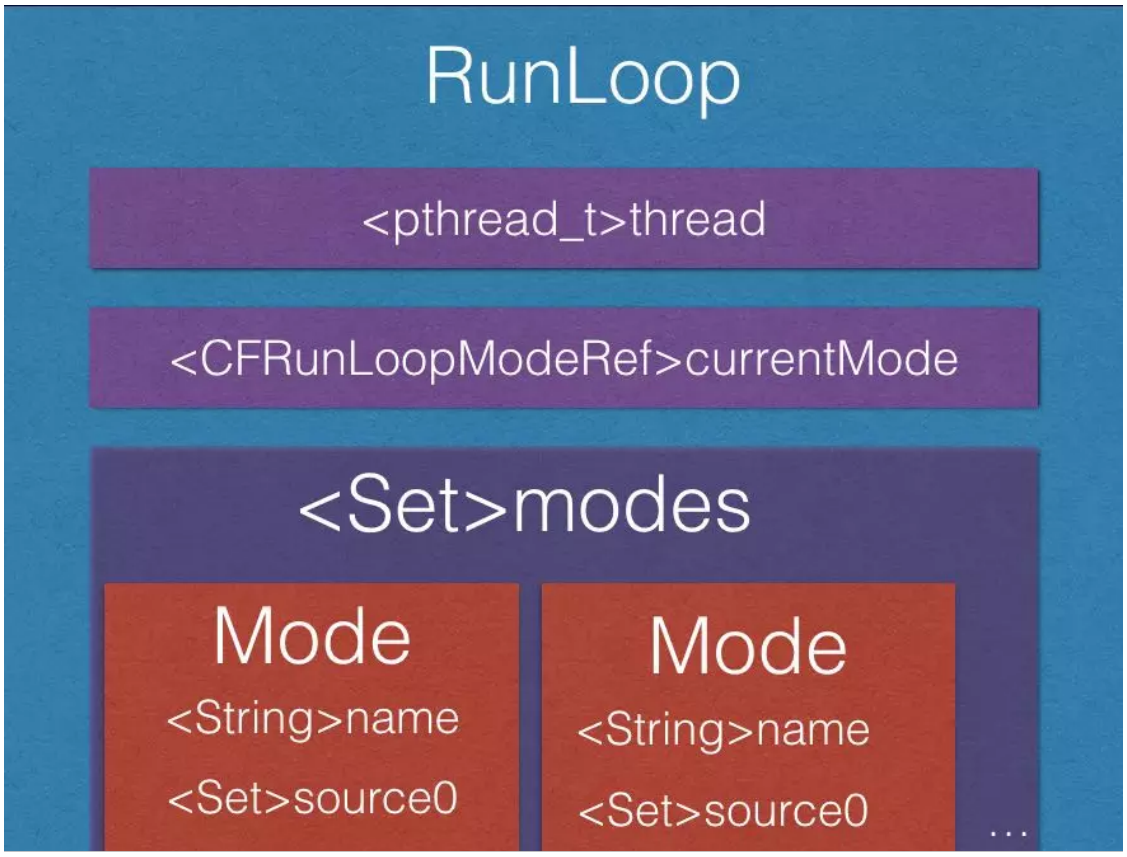

# 5. RunLoop的五个类

在 **Core Foundation** 里面关于 **RunLoop** 有5个类:

| 序号 | 类 | 说明 |
| --- | --- | --- |
| | | |

| | | |
|---|---|---|
| 1 | **CFRunloopRef** | 【RunLoop本身】 |
| 2 | **CFRunloopModeRef** | 【Runloop的运行模式】 |
| 3 | **CFRunloopSourceRef** | 【Runloop要处理的事件源】 |
| 4 | **CFRunloopTimerRef** | 【Timer事件】 |
| 5 | **CFRunloopObserverRef** | 【Runloop的观察者（监听者）】 |

他们的关系如下:




## 5.1 CFRunLoop

① 大致结构

CFRunLoop 的结构大致如下：

```
struct __CFRunLoop {
    CFMutableSetRef _commonModes;     // Set
    CFMutableSetRef _commonModeItems; // Set
    CFRunLoopModeRef _currentMode;    // Current Runloop Mode
    CFMutableSetRef _modes;           // Set
    ...
};
```

### ② CommonModes

如上，有个概念叫 **CommonModes**：一个 Mode 可以将自己标记为"Common"属性：通过将其 ModeName 添加到 RunLoop 的 **commonModes** 中。例如：

```
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent
*)event {
    NSTimer *timer = [NSTimer timerWithTimeInterval:2.0 target:self
selector:@selector(show) userInfo:nil repeats:YES];
    [[NSRunLoop mainRunLoop] addTimer:timer
forMode:NSRunLoopCommonModes];
    NSLog(@"%@",[NSRunLoop mainRunLoop]);
}
```

### ③ CommonModeItems

如上所示，添加 **source** 的时候，如果 modeName 传入**kCFRunLoopCommonModes** 或者 **NSRunLoopCommonModes**，则该 source 会被保存到 RunLoop 的 **_commonModeItems** 中，而且，会被添加到 **commonModes** 中的所有mode中去。

其实，每当 **RunLoop** 的内容发生变化时，**RunLoop** 都会自动将 **_commonModeItems** 里的 Source/Observer/Timer 同步到具有 **Common** 标记的所有Mode里。

### ④ 场景举例

主线程的 **RunLoop** 里有两个预置的 Mode：**kCFRunLoopDefaultMode** 和 **UITrackingRunLoopMode**。这两个 Mode 都已经被标记为 **Common** 属性。

DefaultMode 是 App 平时所处的状态，TrackingRunLoopMode 是追踪 ScrollView 滑动时的状态。当你创建一个 Timer 并加到 DefaultMode 时，Timer 会得到重复回调，但此时滑动一个TableView时，RunLoop 会将 mode 切换为 TrackingRunLoopMode，这时 Timer 就不会被回调，并且也不会影响到滑动操作。

有时你需要一个 Timer，在两个 Mode 中都能得到回调，一种办法就是将这个 Timer 分别加入这两个 Mode。还有一种方式，就是将 Timer 加入到顶层的 RunLoop 的 **commonModeItems** 中。**commonModeItems** 被 RunLoop 自动更新到所有具有 **Common** 属性的 Mode 里去。

**⑤ 特点**

一个 **RunLoop** 包含若干个 Mode，每个 Mode 又包含若干个**Source/Timer/Observer**。但是，运行的时候，一条线程对应一个 **Runloop**，**Runloop** 总是运行在某种特定的**CFRunLoopModeRef**（运行模式）下。

这是因为，在 **Runloop** 中有多个运行模式，每次调用 **RunLoop** 的主函数`__CFRunloopRun()` 时，只能指定其中一个 **Mode**（称 CurrentMode）运行， 如果需要切换 Mode，只能是退出 CurrentMode 切换到指定的 Mode 进入，目的以保证不同 Mode 下的 `Source / Timer / Observer` 互不影响。

每次调用 **RunLoop** 的主函数时，只能指定其中一个 Mode，这个Mode被称作 **CurrentMode**。如果需要切换 Mode，只能退出 Loop，再重新指定一个 Mode 进入。这样做主要是为了分隔开不同组的 Source/Timer/Observer，让其互不影响。

**Runloop** 要有效，mode 里面 至少 要有一个 **timer**（定时器事件）或者是 **source**（源）;

## 5.2 CFRunLoopMode

**① 大致结构**

**CFRunLoopMode** 的结构大致如下：

```
struct __CFRunLoopMode {
    CFStringRef _name;            // Mode Name, 例如
@"kCFRunLoopDefaultMode"
    CFMutableSetRef _sources0;    // Set
    CFMutableSetRef _sources1;    // Set
    CFMutableArrayRef _observers; // Array
    CFMutableArrayRef _timers;    // Array
```

```
    ...
};
```

### ② Mode 及操作接口

**CFRunLoopModeRef** 类并没有对外暴露，只是通过 **CFRunLoopRef** 的接口进行了封装。**CFRunLoopRef** 获取 **Mode** 的接口：

```
CFRunLoopAddCommonMode(CFRunLoopRef runloop, CFStringRef modeName);
CFRunLoopRunInMode(CFStringRef modeName, ...);
```

我们没有办法直接创建一个CFRunLoopMode对象，但是我们可以调用`CFRunLoopAddCommonMode`传入一个字符串向 **RunLoop** 中添加 **Mode**，传入的字符串即为 Mode 的名字，Mode对象应该是此时在RunLoop内部创建的。

这里看一下`CFRunLoopAddCommonMode`源码。

```
void CFRunLoopAddCommonMode(CFRunLoopRef rl, CFStringRef modeName)
{
    CHECK_FOR_FORK();
    if (__CFRunLoopIsDeallocating(rl)) return;
    __CFRunLoopLock(rl);
    //看rl中是否已经有这个mode，如果有就什么都不做
    if (!CFSetContainsValue(rl->_commonModes, modeName)) {
        CFSetRef set = rl->_commonModeItems ?
CFSetCreateCopy(kCFAllocatorSystemDefault, rl->_commonModeItems) :
NULL;
        //把modeName添加到RunLoop的_commonModes中
        CFSetAddValue(rl->_commonModes, modeName);
        if (NULL != set) {
            CFTypeRef context[2] = {rl, modeName};
            /* add all common-modes items to new mode */
            //这里调用
CFRunLoopAddSource/CFRunLoopAddObserver/CFRunLoopAddTimer的时候会调用
            //__CFRunLoopFindMode(rl, modeName, true)，CFRunLoopMode
对象在这个时候被创建
            CFSetApplyFunction(set,
(__CFRunLoopAddItemsToCommonMode), (void *)context);
            CFRelease(set);
        }
```

```
    } else {
    }
    __CFRunLoopUnlock(rl);
}
```

可以看得出：

- modeName不能重复，modeName是mode的唯一标识符
- RunLoop的_commonModes数组存放所有被标记为common的mode的名称
- 添加commonMode会把commonModeItems数组中的所有source同步到新添加的mode中
- CFRunLoopMode对象在CFRunLoopAddItemsToCommonMode函数中调用CFRunLoopFindMode时被创建

**③ mode item 及操作接口**

**Source/Timer/Observer** 被统称为 **mode item**，一个 item 可以被同时加入多个 mode。但一个 item 被重复加入同一个 mode 时是不会有效果的。如果一个 mode 中一个 item 都没有，则 RunLoop 会直接退出，不进入循环。

Mode 暴露的管理 **mode item** 的接口有下面几个，通过他们我们可以为Run Loop 添加 Source（ModeItem）。

```
void CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef source,
CFStringRef mode)
void CFRunLoopRemoveSource(CFRunLoopRef rl, CFRunLoopSourceRef
source, CFStringRef mode)
void CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObserverRef
observer, CFStringRef mode)
void CFRunLoopRemoveObserver(CFRunLoopRef rl, CFRunLoopObserverRef
observer, CFStringRef mode)
void CFRunLoopAddTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer,
CFStringRef mode)
void CFRunLoopRemoveTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer,
CFStringRef mode)
```

你只能通过 **mode name** 来操作内部的 mode，当你传入一个新的 mode name 但 RunLoop 内部没有对应 mode 时，RunLoop会自动帮你创建对应的 **CFRunLoopModeRef**。对于一个 RunLoop 来说，其内部的 mode 只能增加不能删

除。

这里只分析其中 `CFRunLoopAddSource` 的源码

```
//添加source事件
void CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef rls,
CFStringRef modeName) {    /* DOES CALLOUT */
    CHECK_FOR_FORK();
    if (__CFRunLoopIsDeallocating(rl)) return;
    if (!__CFIsValid(rls)) return;
    Boolean doVer0Callout = false;
    __CFRunLoopLock(rl);
    //如果是kCFRunLoopCommonModes
    if (modeName == kCFRunLoopCommonModes) {
        //如果runloop的_commonModes存在，则copy一个新的复制给set
        CFSetRef set = rl->_commonModes ?
CFSetCreateCopy(kCFAllocatorSystemDefault, rl->_commonModes) :
NULL;
       //如果runl _commonModeItems为空
        if (NULL == rl->_commonModeItems) {
            //先初始化
            rl->_commonModeItems =
CFSetCreateMutable(kCFAllocatorSystemDefault, 0,
&kCFTypeSetCallBacks);
        }
        //把传入的CFRunLoopSourceRef加入_commonModeItems
        CFSetAddValue(rl->_commonModeItems, rls);
        //如果刚才set copy到的数组里有数据
        if (NULL != set) {
            CFTypeRef context[2] = {rl, rls};
            /* add new item to all common-modes */
            //则把set里的所有mode都执行一
遍__CFRunLoopAddItemToCommonModes函数
            CFSetApplyFunction(set,
(__CFRunLoopAddItemToCommonModes), (void *)context);
            CFRelease(set);
        }
        //以上分支的逻辑就是，如果你往kCFRunLoopCommonModes里面添加一
个source，那么所有_commonModes里的mode都会添加这个source
    } else {
        //根据modeName查找mode
        CFRunLoopModeRef rlm = __CFRunLoopFindMode(rl, modeName,
true);
        //如果_sources0不存在，则初始化_sources0，_sources0和
_portToV1SourceMap
        if (NULL != rlm && NULL == rlm->_sources0) {
```

```
            rlm->_sources0 =
CFSetCreateMutable(kCFAllocatorSystemDefault, 0,
&kCFTypeSetCallBacks);
            rlm->_sources1 =
CFSetCreateMutable(kCFAllocatorSystemDefault, 0,
&kCFTypeSetCallBacks);
            rlm->_portToV1SourceMap =
CFDictionaryCreateMutable(kCFAllocatorSystemDefault, 0, NULL,
NULL);
        }
        //如果_sources0和_sources1中都不包含传入的source
        if (NULL != rlm && !CFSetContainsValue(rlm->_sources0, rls)
&& !CFSetContainsValue(rlm->_sources1, rls)) {
            //如果version是0，则加到_sources0
            if (0 == rls->_context.version0.version) {
                CFSetAddValue(rlm->_sources0, rls);
                //如果version是1，则加到_sources1
            } else if (1 == rls->_context.version0.version) {
                CFSetAddValue(rlm->_sources1, rls);
                __CFPort src_port = rls-
>_context.version1.getPort(rls->_context.version1.info);
                if (CFPORT_NULL != src_port) {
                    //此处只有在加到source1的时候才会把souce和一个
mach_port_t对应起来
                    //可以理解为，source1可以通过内核向其端口发送消息来主动
唤醒runloop
                    CFDictionarySetValue(rlm->_portToV1SourceMap,
(const void *)(uintptr_t)src_port, rls);
                    __CFPortSetInsert(src_port, rlm->_portSet);
                }
            }
            __CFRunLoopSourceLock(rls);
            //把runloop加入到source的_runLoops中
            if (NULL == rls->_runLoops) {
                rls->_runLoops =
CFBagCreateMutable(kCFAllocatorSystemDefault, 0,
&kCFTypeBagCallBacks); // sources retain run loops!
            }
            CFBagAddValue(rls->_runLoops, rl);
            __CFRunLoopSourceUnlock(rls);
            if (0 == rls->_context.version0.version) {
                if (NULL != rls->_context.version0.schedule) {
                    doVer0Callout = true;
                }
            }
        }
        if (NULL != rlm) {
            __CFRunLoopModeUnlock(rlm);
```

```
            }
        }
        __CFRunLoopUnlock(rl);
        if (doVer0Callout) {
            // although it looses some protection for the source, we
have no choice but
            // to do this after unlocking the run loop and mode locks,
to avoid deadlocks
            // where the source wants to take a lock which is already
held in another
            // thread which is itself waiting for a run loop/mode lock
            rls->_context.version0.schedule(rls-
>_context.version0.info, rl, modeName); /* CALLOUT */
        }
    }
```

通过添加source的这段代码可以得出如下结论：

- 如果modeName传入kCFRunLoopCommonModes，则该source会被保存到RunLoop的_commonModeItems中
- 如果modeName传入kCFRunLoopCommonModes，则该source会被添加到所有commonMode中
- 如果modeName传入的不是kCFRunLoopCommonModes，则会先查找该Mode，如果没有，会创建一个
- 同一个source在一个mode中只能被添加一次

#### ④ mode name

苹果公开提供的 Mode 有两个：**kCFRunLoopDefaultMode** (NSDefaultRunLoopMode) 和 **UITrackingRunLoopMode**，你可以用这两个 Mode Name 来操作其对应的 Mode。

苹果还提供了一个操作 Common 标记的字符串：**kCFRunLoopCommonModes** (NSRunLoopCommonModes)，你可以用这个字符串来操作 **Common Items**，或标记一个 Mode 为 "Common"。使用时注意区分这个字符串和其他 mode name。

更完整的mode name如下表所示：

| mode name | 说明 |
| --- | --- |
| **kCFRunLoop** | |

| | |
|---|---|
| **DefaultMode** | App的默认Mode，通常主线程是在这个Mode下运行 |
| **UITrackingRunLoopMode** | 界面跟踪 Mode，用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他 Mode 影响 |
| UIInitializationRunLoopMode | 在刚启动 App 时第进入的第一个 Mode，启动完成后就不再使用 |
| GSEventReceiveRunLoopMode | 接受系统事件的内部 Mode，通常用不到 |
| **kCFRunLoopCommonModes** | 这是一个占位用的Mode，作为标记kCFRunLoopDefaultMode和UITrackingRunLoopMode用，并不是一种真正的Mode |

## 5.3 CFRunLoopSourceRef (输入源)

CFRunLoopSourceRef 是事件产生的地方。Source有两个版本：Source0 和 Source1。

数据结构（source0/source1）：

```
// source0 (manual): order(优先级), callout(回调函数)
CFRunLoopSource {order =..., {callout =... }}

// source1 (mach port)：order(优先级), port:(端口), callout(回调函数)
CFRunLoopSource {order = ..., {port = ..., callout =...}
```

- **Source0**：只包含了一个回调（函数指针），它并不能主动触发事件。使用时，你需要先调用 CFRunLoopSourceSignal(source)，将这个 Source 标记为待处理，然后手动调用 CFRunLoopWakeUp(runloop) 来唤醒 RunLoop，让其处理这个事件。
- **Source1**：包含了一个 mach_port 和一个回调（函数指针），被用于通过内核和其他线程相互发送消息。这种 Source 能主动唤醒 RunLoop 的线程，其原理在下面会讲到。

### 5.4 CFRunLoopTimerRef (定时源)

CFRunLoopTimerRef 是基于时间的触发器，它和 NSTimer 是 Toll-Free Bridged 的，可以混用。其包含一个时间长度和一个回调（函数指针）。当其加入到 RunLoop 时，RunLoop会注册对应的时间点，当时间点到时，RunLoop会被唤醒以执行那个回调。

### 5.5 CFRunLoopObserverRef (观察者)

CFRunLoopObserverRef 是观察者，每个 Observer 都包含了一个回调（函数指针），当 RunLoop 的状态发生变化时，观察者就能通过回调接受到这个变化。可以观测的时间点有以下几个：

```
typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
    kCFRunLoopEntry         = (1UL << 0), // 即将进入Loop
    kCFRunLoopBeforeTimers  = (1UL << 1), // 即将处理 Timer
    kCFRunLoopBeforeSources = (1UL << 2), // 即将处理 Source
    kCFRunLoopBeforeWaiting = (1UL << 5), // 即将进入休眠
    kCFRunLoopAfterWaiting  = (1UL << 6), // 刚从休眠中唤醒
    kCFRunLoopExit          = (1UL << 7), // 即将退出Loop
};
```

## 6. 实战

### 6.1 设置输入源

#### ① performSelector

performSelector同样是触发Source0事件。selector也是特殊的基于自定义的源.理论上来说,允许在当前线程向任何线程上执行发送消息,和基于端口的源一样,执行selector请求会在目标线程上序列化,减缓许多在线程上允许多个方法容易引起的同步问题.不像基于端口的源,一个selector执行完后会自动从run loop里面移除.

- 主线程执行

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
```

```
    [self performSelectorOnMainThread:@selector(test)
withObject:nil waitUntilDone:YES];
});
```

- 当前线程延时执行

```
// 内部会创建一个Timer到当前线程的runloop中（如果当前线程没runloop则方法无
效；performSelector:onThread: 方法放到指定线程runloop中）
- (void)performSelector:(SEL)aSelector withObject:(id)anArgument
afterDelay:(NSTimeInterval)delay
```

当调用上述API，实际上其内部会创建一个 Timer 并添加到当前线程的 RunLoop 中。所以如果当前线程没有 RunLoop，则这个方法会失效。

- 指定线程执行

```
- (void)performSelector:(SEL)aSelector onThread:(NSThread *)thr
withObject:(id)arg waitUntilDone:(BOOL)wait;
```

当调用 performSelector:onThread: 时，实际上其会创建一个Timer加到对应的线程去，同样的，如果对应线程没有 RunLoop 该方法也会失效.

- 当前线程指定mode name并延时执行

```
// 只在NSDefaultRunLoopMode下执行(刷新图片)
[self.myImageView performSelector:@selector(setImage:) withObject:
[UIImage imageNamed:@""] afterDelay:ti
inModes:@[NSDefaultRunLoopMode]];
```

**② 自定义输入源**

自定义源：使用CFRunLoopSourceRef 类型相关的函数 (线程) 来创建自定义输入源。

- 调用VC

```
-(void)test {
    NSThread* aThread = [[NSThread alloc] initWithTarget:self
selector:@selector(testForCustomedSource) object:nil];
    self.aThread = aThread;
    [aThread start];
}

-(void)testForCustomedSource{
    NSLog(@"starting thread.......");

    NSRunLoop *myRunLoop = [NSRunLoop currentRunLoop];

   // 设置Run Loop observer的运行环境
    CFRunLoopObserverContext context = {0, (__bridge void *)(self),
NULL, NULL, NULL};

    // 创建Run loop observer对象
    CFRunLoopObserverRef observer =
CFRunLoopObserverCreate(kCFAllocatorDefault,kCFRunLoopAllActivities,
YES, 0, &myRunLoopObserver, &context);
    if (observer){
        CFRunLoopRef cfLoop = [myRunLoop getCFRunLoop];
        CFRunLoopAddObserver(cfLoop, observer,
kCFRunLoopDefaultMode);
    }

    _source = [[ZXRunLoopSource alloc] init];
    [_source addToCurrentRunLoop];
    while (!self.aThread.isCancelled)
    {
        NSLog(@"We can do other work");
        [myRunLoop runMode:NSDefaultRunLoopMode beforeDate:[NSDate
dateWithTimeIntervalSinceNow:5.0f]];
    }
    [_source invalidate];
    NSLog(@"finishing thread.........");
}
```

- 自定义输入源

```
- (id)init
{
    CFRunLoopSourceContext  context = {0, (__bridge void *)(self),
NULL, NULL, NULL, NULL, NULL,
```

```
        &RunLoopSourceScheduleRoutine,
        RunLoopSourceCancelRoutine,
        RunLoopSourcePerformRoutine};

    _runLoopSource = CFRunLoopSourceCreate(NULL, 0, &context);
    _commands = [[NSMutableArray alloc] init];

    return self;
}

- (void)addToCurrentRunLoop
{
    //获取当前线程的runLoop(辅助线程)
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFRunLoopAddSource(runLoop, _runLoopSource,
kCFRunLoopDefaultMode);
}

/**
 *  调度例程
 *  当将输入源安装到run loop后，调用这个协调调度例程，将源注册到客户端（可以理
解为其他线程）
 *
 */
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl,
CFStringRef mode)
{
    ZXRunLoopSource *obj = (__bridge ZXRunLoopSource*)info;
//    AppDelegate*   delegate = [[AppDelegate sharedAppDelegate];
    AppDelegate *delegate = [[UIApplication sharedApplication]
delegate];
    RunLoopContext *theContext = [[RunLoopContext alloc]
initWithSource:obj andLoop:rl];

    //发送注册请求
    [delegate
performSelectorOnMainThread:@selector(registerSource:)
withObject:theContext waitUntilDone:YES];
}

/**
 *  处理例程
 *  在输入源被告知（signal source）时，调用这个处理例程，这儿只是简单的调用了
[obj sourceFired]方法
 *
 */
void RunLoopSourcePerformRoutine (void *info)
{
```

```
    ZXRunLoopSource*  obj = (__bridge ZXRunLoopSource*)info;
    [obj sourceFired];
//    [NSTimer scheduledTimerWithTimeInterval:1.0 target:obj
selector:@selector(timerAction:) userInfo:nil repeats:YES];
}

/**
 *  取消例程
 *  如果使用CFRunLoopSourceInvalidate/CFRunLoopRemoveSource函数把输入源
从run loop里面移除的话，系统会调用这个取消例程，并且把输入源从注册的客户端（可以
理解为其他线程）里面移除
 *
 */
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl,
CFStringRef mode)
{
    ZXRunLoopSource* obj = (__bridge ZXRunLoopSource*)info;
    AppDelegate* delegate = [AppDelegate sharedAppDelegate];
    RunLoopContext* theContext = [[RunLoopContext alloc]
initWithSource:obj andLoop:rl];

    [delegate performSelectorOnMainThread:@selector(removeSource:)
withObject:theContext waitUntilDone:NO];
}

- (void)sourceFired
{
    NSLog(@"Source fired: do some work, dude!");
    NSThread *thread = [NSThread currentThread];
    [thread cancel];

    //既然线程没了，就把AppDelegate缓存的runloop也给删了，以免下次调用
CFRunLoopWakeUp(runloop);会崩溃，因为只有runloop没了线程
    [[AppDelegate sharedAppDelegate].sources
removeObjectAtIndex:0];
}
```

③ 端口输入源

配置 NSMachPort 对象

为了和 NSMachPort 对象建立稳定的本地连接，你需要创建端口对象并将之加入相应的线程的 run loop。当运行辅助线程的时候，你传递端口对象到线程的主体入口点。辅助线程可以使用相同的端口对象将消息返回给原线程。

- VC调用

```
- (void)launchThreadForPort
{
    NSPort* myPort = [NSMachPort port];
    if (myPort)
    {
        //这个类持有即将到来的端口消息
        [myPort setDelegate:self];
        //将端口作为输入源安装到当前的 runLoop
        [[NSThread currentThread] setName:@"launchThreadForPort---
Thread"];
        [[NSRunLoop currentRunLoop] addPort:myPort
forMode:NSDefaultRunLoopMode];
        //当前线程去调起工作线程
        MyWorkerClass *work = [[MyWorkerClass alloc] init];
        [NSThread
detachNewThreadSelector:@selector(launchThreadWithPort:)
toTarget:work withObject:myPort];
    }
}
```

为了在线程间建立双向的通信，你需要让工作线程在签到的消息中发送自己的本地端口到主线程。主线程接收到签到消息后就可以知道辅助线程运行正常，并且供了发送消息给辅助线程的方法。

以下代码显示了主线程的 `handlePortMessage:` 方法。当由数据到达线程的本地端口时，该方法被调用。当签到消息到达时，此方法可以直接从辅助线程里面检索端口并保存下来以备后续使用。

- VC实现代理

```
//NSPortDelegate
#define kCheckinMessage 100
//处理从工作线程返回的响应
- (void) handlePortMessage: (id)portMessage {
    //消息的 id
    unsigned int messageId = (int)[[portMessage
valueForKeyPath:@"msgid"] unsignedIntegerValue];

    if (messageId == kCheckinMessage) {
```

```
        //1\. 当前主线程的port
        NSPort *localPort = [portMessage
valueForKeyPath:@"localPort"];
        //2\. 接收到消息的port（来自其他线程）
        NSPort *remotePort = [portMessage
valueForKeyPath:@"remotePort"];
        //3\. 获取工作线程关联的端口，并设置给远程端口，结果同2
        NSPort *distantPort = [portMessage
valueForKeyPath:@"sendPort"];

        NSMutableArray *arr = [[portMessage
valueForKeyPath:@"components"] mutableCopy];
        if ([arr objectAtIndex:0]) {
            NSData *data = [arr objectAtIndex:0];
            NSString * str  =[[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding];
            NSLog(@"");
        }
        NSLog(@"");
        //为了以后的使用保存工作端口
//        [self storeDistantPort: distantPort];
    } else {
        //处理其他的消息
    }
}
```

对于辅助工作线程，你必须配置线程使用特定的端口以发送消息返回给主要线程。

以下显示了如何设置工作线程的代码。创建了线程的自动释放池后，紧接着创建工作对象驱动线程运行。工作对象的 `sendCheckinMessage:` 方法创建了工作线程的本地端口并发送签到消息回主线程。

- MyWorkerClass.m

```
- (void)launchThreadWithPort:(NSPort *)port {
    @autoreleasepool {

        //1\. 保存主线程传入的port
        remotePort = port;

        //2\. 设置子线程名字
        [[NSThread currentThread] setName:@"MyWorkerClassThread"];

        //3\. 开启runloop
```

```
        [[NSRunLoop currentRunLoop] run];

        //4\. 创建自己port
        myPort = [NSPort port];

        //5.
        myPort.delegate = self;

        //6\. 将自己的port添加到runloop
        //作用1、防止runloop执行完毕之后推出
        //作用2、接收主线程发送过来的port消息
        [[NSRunLoop currentRunLoop] addPort:myPort
forMode:NSDefaultRunLoopMode];

        //7\. 完成向主线程port发送消息
        [self sendPortMessage];
    }
}
```

当使用 **NSMachPort** 的时候，本地和远程线程可以使用相同的端口对象在线程间进行单边通信。换句话说，一个线程创建的本地端口对象成为另一个线程的远程端口对象。

以下代码辅助线程的签到例程，该方法为之后的通信设置自己的本地端口，然后发送签到消息给主线程。它使用 `LaunchThreadWithPort:` 方法中收到的端口对象做为目标消息。

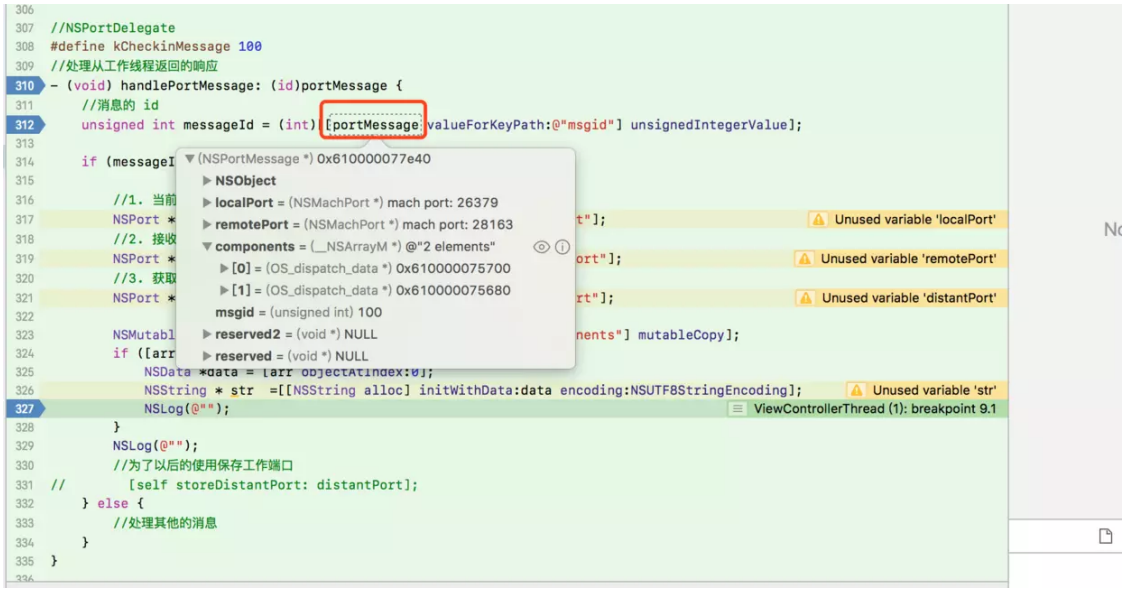
- MyWorkerClass.m

```
- (void)sendPortMessage {

    NSString *str1 = @"aaa111";
    NSString *str2 = @"bbb222";
    arr = [[NSMutableArray alloc] initWithArray:@[[str1
dataUsingEncoding:NSUTF8StringEncoding],[str2
dataUsingEncoding:NSUTF8StringEncoding]]];
    //发送消息到主线程，操作1
    [remotePort sendBeforeDate:[NSDate date]
                         msgid:kMsg1
                    components:arr
                          from:myPort
                      reserved:0];
```

```
    //发送消息到主线程，操作2
    //    [remotePort sendBeforeDate:[NSDate date]
    //                         msgid:kMsg2
    //                    components:nil
    //                          from:myPort
    //                      reserved:0];
}
```

实验验证




> - macOS特殊情况（iOS开发者可忽略）

为了和 NSMeaasgePort 的建立稳定的本地连接，你不能简单的在线程间传递端口对象。远程消息端口必须通过名字来获得。在 Cocoa 中这需要你给本地端口指定一个名字，并将名字传递到远程线程以便远程线程可以获得合适的端口对象用于通信。以下代码显示端口创建，注册到你想要使用消息端口的进程。

```
- (void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runLoop
{
    //当手动调用此方法的时候，将会触发 RunLoopSourceContext的
performCallback
    CFRunLoopSourceSignal(runLoopSource);
    CFRunLoopWakeUp(runLoop);

    NSPort *localPort = [[NSMessagePort alloc] init];

    // configure the port and add it to the current run loop
```

```
    [localPort setDelegate:self];
    [[NSRunLoop currentRunLoop] addPort:localPort
forMode:NSDefaultRunLoopMode];

    // register the port using the specific name, and The name is
unique
    NSString *localPortName = [NSString
stringWithFormat:@"MyPortName"];
    // there is only NSMessagePortNameServer in the mac os x system
    [[NSMessagePortNameServer sharedInstance]
registerPort:localPort name:localPortName];
}
```

需要注意的是，只能在一个设备内程序间通信，不能在不同设备间通信。将端口名称注册到NSMessagePortNameServer里面，其他线程通过这个端口名称从NSMessagePortNameServer来获取这个端口对象。

根据name获取port的API为：

```
- (NSPort *)portForName:(NSString *)name;
```

```
- (NSPort *)portForName:(NSString *)name host:(NSString *)host;
```

> 区分：NSPort，NSMessagePort，NSMachPort，NSPortMessage

### ❶ iOS和macOS都有的类: 在NSPort.h中可找到

- NSPort

```
@interface NSPort : NSObject <NSCopying, NSCoding>

+ (NSPort *)port;

- (void)invalidate;
@property (readonly, getter=isValid) BOOL valid;

- (void)setDelegate:(nullable id <NSPortDelegate>)anObject;
- (nullable id <NSPortDelegate>)delegate;
```

```
- (void)scheduleInRunLoop:(NSRunLoop *)runLoop forMode:
(NSRunLoopMode)mode;
- (void)removeFromRunLoop:(NSRunLoop *)runLoop forMode:
(NSRunLoopMode)mode;

@property (readonly) NSUInteger reservedSpaceLength;
- (BOOL)sendBeforeDate:(NSDate *)limitDate components:(nullable
NSMutableArray *)components from:(nullable NSPort *) receivePort
reserved:(NSUInteger)headerSpaceReserved;
- (BOOL)sendBeforeDate:(NSDate *)limitDate msgid:(NSUInteger)msgID
components:(nullable NSMutableArray *)components from:(nullable
NSPort *)receivePort reserved:(NSUInteger)headerSpaceReserved;

#if (TARGET_OS_MAC && !(TARGET_OS_EMBEDDED || TARGET_OS_IPHONE)) ||
(TARGET_OS_WIN32)
- (void)addConnection:(NSConnection *)conn toRunLoop:(NSRunLoop
*)runLoop forMode:(NSRunLoopMode)mode NS_SWIFT_UNAVAILABLE("Use
NSXPCConnection instead") API_DEPRECATED("Use NSXPCConnection
instead", macosx(10.0, 10.13), ios(2.0,11.0), watchos(2.0,4.0),
tvos(9.0,11.0));
- (void)removeConnection:(NSConnection *)conn fromRunLoop:
(NSRunLoop *)runLoop forMode:(NSRunLoopMode)mode
NS_SWIFT_UNAVAILABLE("Use NSXPCConnection instead")
API_DEPRECATED("Use NSXPCConnection instead", macosx(10.0, 10.13),
ios(2.0,11.0), watchos(2.0,4.0), tvos(9.0,11.0));

#endif

@end
```

- NSMessagePort

```
@interface NSMessagePort : NSPort {
    @private
    void *_port;
    id _delegate;
}
```

- NSMachPort

```
@interface NSMachPort : NSPort {
```

```
    @private
    id _delegate;
    NSUInteger _flags;
    uint32_t _machPort;
    NSUInteger _reserved;
}
```

### ❷ 仅macOS支持的类: 在NSPortMessage.h中可找到

- NSPortMessage

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

@class NSPort, NSDate, NSArray, NSMutableArray;

NS_ASSUME_NONNULL_BEGIN

@interface NSPortMessage : NSObject {
    @private
    NSPort      *localPort;
    NSPort      *remotePort;
    NSMutableArray  *components;
    uint32_t        msgid;
    void        *reserved2;
    void        *reserved;
}

- (instancetype)initWithSendPort:(nullable NSPort *)sendPort
receivePort:(nullable NSPort *)replyPort components:(nullable
NSArray *)components NS_DESIGNATED_INITIALIZER;

@property (nullable, readonly, copy) NSArray *components;
@property (nullable, readonly, retain) NSPort *receivePort;
@property (nullable, readonly, retain) NSPort *sendPort;
- (BOOL)sendBeforeDate:(NSDate *)date;

@property uint32_t msgid;

@end

NS_ASSUME_NONNULL_END
```

## 6.2 设置定时源

### 6.2.1 使用系统Timer

我们的定时器Timer是怎么写的呢？一般的做法是，在主线程（可能是某控制器的viewDidLoad方法）中，创建Timer。

可能会有两种写法，但是都有上面的问题，下面先看下Timer的两种写法：

```
// 第一种写法
NSTimer *timer = [NSTimer timerWithTimeInterval:1.0 target:self
selector:@selector(timerUpdate) userInfo:nil repeats:YES];
[[NSRunLoop currentRunLoop] addTimer:timer
forMode:NSDefaultRunLoopMode];
[timer fire];

// 第二种写法
[NSTimer scheduledTimerWithTimeInterval:1.0 target:self
selector:@selector(timerUpdate) userInfo:nil repeats:YES];
```

上面的两种写法其实是等价的。第二种写法，默认也是将timer添加到**NSDefaultRunLoopMode** 下的，并且会自动fire。

> **可能的问题**： 1.我们经常会在应用中看到tableView 的header 上是一个横向ScrollView，一般我们使用NSTimer，每隔几秒切换一张图片。可是当我们滑动tableView的时候，顶部的scollView并不会切换图片，这可怎么办呢？ 2.界面上除了有tableView，还有显示倒计时的Label，当我们在滑动tableView时，倒计时就停止了，这又该怎么办呢?

要如何解决这一问题呢？ 解决方法很简单，我们只需要在添加timer 时，将mode设置为NSRunLoopCommonModes即可。

```
- (void)timerTest
{
    // 第一种写法
    NSTimer *timer = [NSTimer timerWithTimeInterval:1.0 target:self
selector:@selector(timerUpdate) userInfo:nil repeats:YES];
    [[NSRunLoop currentRunLoop] addTimer:timer
forMode:NSRunLoopCommonModes];
    [timer fire];
    // 第二种写法，因为是固定添加到defaultMode中，就不要用了
```

```
}
```

还有一种方案，在子线程中添加Timer，也可以解决上面的问题，但是需要注意的是把timer加入到当前runloop后，必须让runloop 运行起来，否则timer仅执行一次。

```
//首先是创建一个子线程
- (void)createThread
{
    NSThread *subThread = [[NSThread alloc] initWithTarget:self
selector:@selector(timerTest) object:nil];
    [subThread start];
    self.subThread = subThread;
}

// 创建timer，并添加到runloop的mode中
- (void)timerTest
{
    @autoreleasepool {
        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        NSLog(@"启动RunLoop前--%@",runLoop.currentMode);
        NSLog(@"currentRunLoop:%@",[NSRunLoop currentRunLoop]);
        // 第一种写法,改正前
    //    NSTimer *timer = [NSTimer timerWithTimeInterval:1.0
target:self selector:@selector(timerUpdate) userInfo:nil
repeats:YES];
    //    [[NSRunLoop currentRunLoop] addTimer:timer
forMode:NSDefaultRunLoopMode];
    //    [timer fire];
        // 第二种写法
        [NSTimer scheduledTimerWithTimeInterval:1.0 target:self
selector:@selector(timerUpdate) userInfo:nil repeats:YES];

        [[NSRunLoop currentRunLoop] run];
    }
}

//更新label
- (void)timerUpdate
{
    NSLog(@"当前线程：%@",[NSThread currentThread]);
    NSLog(@"启动RunLoop后--%@",[NSRunLoop
currentRunLoop].currentMode);
    NSLog(@"currentRunLoop:%@",[NSRunLoop currentRunLoop]);
    dispatch_async(dispatch_get_main_queue(), ^{
```

```
        self.count ++;
        NSString *timerText = [NSString stringWithFormat:@"计时
器:%ld",self.count];
        self.timerLabel.text = timerText;
    });
}
```

timer确实被添加到NSDefaultRunLoopMode中了。可是添加到子线程中的NSDefaultRunLoopMode里，无论如何滚动，timer都能够很正常的运转。这又是为啥呢?

这就是多线程与runloop的关系了，每一个线程都有一个与之关联的RunLoop，而每一个RunLoop可能会有多个Mode。CPU会在多个线程间切换来执行任务，呈现出多个线程同时执行的效果。执行的任务其实就是RunLoop去各个Mode里执行各个item。因为RunLoop是独立的两个，相互不会影响，所以在子线程添加timer，滑动视图时，timer能正常运行。

### 6.2.2 使用自定义Timer

使用下面关键两行即可自定义Timer的事件

```
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(kCFAllocatorDefault,
0.1, 0.3, 0, 0,

&myCFTimerCallback, &timerContext);
CFRunLoopAddTimer(runLoop, timer, kCFRunLoopCommonModes);
```

下面是一个例子：

```
-(void)testCustomedTimer{
    // 获得当前thread的Run loop
    NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];
    CFRunLoopObserverContext context = {0, (__bridge void *)(self),
NULL, NULL, NULL};
    // 创建Run loop observer对象
    // 第一个参数用于分配该observer对象的内存
    // 第二个参数用以设置该observer所要关注的的事件，详见回调函
数myRunLoopObserver中注释
    // 第三个参数用于标识该observer是在第一次进入run loop时执行还是每次进入
```

```
run loop处理时均执行
    // 第四个参数用于设置该observer的优先级
    // 第五个参数用于设置该observer的回调函数
    // 第六个参数用于设置该observer的运行环境
    CFRunLoopObserverRef observer =
CFRunLoopObserverCreate(kCFAllocatorDefault,kCFRunLoopAllActivities,
YES, 0, &myRunLoopObserver, &context);
    if (observer){
        CFRunLoopRef cfLoop = [myRunLoop getCFRunLoop];
        CFRunLoopAddObserver(cfLoop, observer,
kCFRunLoopDefaultMode);
    }

    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFRunLoopTimerContext timerContext = {0, NULL, NULL, NULL,
NULL};
    CFRunLoopTimerRef timer =
CFRunLoopTimerCreate(kCFAllocatorDefault, 0.1, 0.3, 0, 0,

&myCFTimerCallback, &timerContext);

    CFRunLoopAddTimer(runLoop, timer, kCFRunLoopCommonModes);
    NSInteger loopCount = 2;
    do{
        [[NSRunLoop currentRunLoop] runUntilDate:[NSDate
dateWithTimeIntervalSinceNow:1]];
        loopCount--;
    }while (loopCount);

}

void myCFTimerCallback(){
    NSLog(@"-----++++-------");
}
```

## 6.3 设置监听

- 添加监听

```
    // 设置Run Loop observer的运行环境
    CFRunLoopObserverContext context = {0, (__bridge void *)(self),
NULL, NULL, NULL};
    // 创建Run loop observer对象
    // 第一个参数用于分配该observer对象的内存
```

```
    // 第二个参数用以设置该observer所要关注的的事件，详见回调函
数myRunLoopObserver中注释
    // 第三个参数用于标识该observer是在第一次进入run loop时执行还是每次进入
run loop处理时均执行
    // 第四个参数用于设置该observer的优先级
    // 第五个参数用于设置该observer的回调函数
    // 第六个参数用于设置该observer的运行环境
    CFRunLoopObserverRef observer =
CFRunLoopObserverCreate(kCFAllocatorDefault,kCFRunLoopAllActivities,
YES, 0, &myRunLoopObserver, &context);
    if (observer){
        CFRunLoopRef cfLoop = [myRunLoop getCFRunLoop];
        CFRunLoopAddObserver(cfLoop, observer,
kCFRunLoopDefaultMode);
    }
```

- 监听回调

```
void myRunLoopObserver(CFRunLoopObserverRef observer,
CFRunLoopActivity activity, void *info)
{
    switch(activity)
    {
            // 即将进入Loop
        case kCFRunLoopEntry:
            NSLog(@"run loop entry");
            break;
        case kCFRunLoopBeforeTimers://即将处理 Timer
            NSLog(@"run loop before timers");
            break;
        case kCFRunLoopBeforeSources://即将处理 Source
            NSLog(@"run loop before sources");
            break;
        case kCFRunLoopBeforeWaiting://即将进入休眠
            NSLog(@"run loop before waiting");
            break;
        case kCFRunLoopAfterWaiting://刚从休眠中唤醒
            NSLog(@"run loop after waiting");
            break;
        case kCFRunLoopExit://即将退出Loop
            NSLog(@"run loop exit");
            break;
        default:
            break;
```

```
    }
}
```