OS X是MacOS与NeXTSTEP的结合。OC是Smalltalk类面向对象编程与C的结合。iCloud则是苹果移动服务与云平台的结合。

上述都是一些亮点，但是不得不说苹果技术中的进程通讯走的是“反人类”的道路。

由于不是根据每个节点上最优原则进行设计，苹果的进程间通信解决方案更显得混乱扎堆。结果是，大量重叠，不兼容的IPC技术在各个抽象层随处可见。（除了GCD还有剪贴板）

* Mach Ports
* Distributed Notifications
* Distributed Objects
* AppleEvents & AppleScript
* Pasteboard
* XPC

从低级内核抽象到高级，面向对象的API，它们都有各自特殊的表现以及安全特性。但是基础层面来看，它们都是从不同上下文段传递或者获取数据的机制。

分述

Mach Ports

所有的进程间通讯最终落实依赖的还是Mach内核API提供的功能。

Mach端口是轻量并且强大的而又缺少相关文档晦涩使用的（天使与恶魔）。

通过一个Mach端口发送一个消息调用一次mach\_msg\_send方法，但是这里需要做一些配置来构建待发送的消息：

natural\_t data;

mach\_port\_t port;

struct {

mach\_msg\_header\_t header;

mach\_msg\_body\_t body;

mach\_msg\_type\_descriptor\_t type;

} message;

message.header = (mach\_msg\_header\_t) {

.msgh\_remote\_port = port,

.msgh\_local\_port = MACH\_PORT\_NULL,

.msgh\_bits = MACH\_MSGH\_BITS(MACH\_MSG\_TYPE\_COPY\_SEND, 0),

.msgh\_size = sizeof(message)

};

message.body = (mach\_msg\_body\_t) {

.msgh\_descriptor\_count = 1

};

message.type = (mach\_msg\_type\_descriptor\_t) {

.pad1 = data,

.pad2 = sizeof(data)

};

mach\_msg\_return\_t error = mach\_msg\_send(&message.header);

**if** (error == MACH\_MSG\_SUCCESS) {

// ...

}

（消息）接收端稍微轻松点，因为消息只需要被声明而不用初始化：

mach\_port\_t port;

struct {

mach\_msg\_header\_t header;

mach\_msg\_body\_t body;

mach\_msg\_type\_descriptor\_t type;

mach\_msg\_trailer\_t trailer;

} message;

mach\_msg\_return\_t error = mach\_msg\_receive(&message.header);

**if** (error == MACH\_MSG\_SUCCESS) {

natural\_t data = message.type.pad1;

// ...

}

还算不错的是，Core Foundation和Foundation为Mach端口提供了高级API。在内核基础上封装的CFMachPort / NSMachPort可以用做runloop源，尽管CFMachPort / NSMachPort有利于的是两个不同端口之间的通讯同步。

CFMessagePort确实非常适合用于简单的一对一通讯。简简单单几行代码，一个本地端口就被附属到runloop源上，只要获取到消息就执行回调。

**static** CFDataRef Callback(CFMessagePortRef port,

SInt32 messageID,

CFDataRef data,

**void** \*info)

{

// ...

}

CFMessagePortRef localPort =

CFMessagePortCreateLocal(nil,

CFSTR("com.example.app.port.server"),

Callback,

nil,

nil);

CFRunLoopSourceRef runLoopSource =

CFMessagePortCreateRunLoopSource(nil, localPort, 0);

CFRunLoopAddSource(CFRunLoopGetCurrent(),

runLoopSource,

kCFRunLoopCommonModes);

若要进行发送数据同样也十分直截了当。只要完成指定远端的端口，装载数据，还有设置发送与接收的超时时间的操作。剩下就由CFMessagePortSendRequest来接管了。

CFDataRef data;

SInt32 messageID = 0x1111; // Arbitrary

CFTimeInterval timeout = 10.0;

CFMessagePortRef remotePort =

CFMessagePortCreateRemote(nil,

CFSTR("com.example.app.port.client"));

SInt32 status =

CFMessagePortSendRequest(remotePort,

messageID,

data,

timeout,

timeout,

NULL,

NULL);

**if** (status == kCFMessagePortSuccess) {

// ...

}

Distributed Notifications

在Cocoa中有很多种两个对象进行通信的途径。

当然也能进行直接消息传递。也有像目标-动作，代理，回调这些解耦，一对一的设计模式。KVO允许让很多对象订阅一个事件，但是它把这些对象都联系起来了。另一方面通知让消息全局广播，并且让有监听该广播的对象接收该消息。【注：想知道发了多少次广播吗？添加 NSNotificationCenter addObserverForName:object:queue:usingBlock，其中name与object置nil，看block被调用了几次。】

每个应用为基础应用消息发布-订阅对自身通知中心实例进行管理。但是鲜有人知的APICFNotificationCenterGetDistributedCenter的通知可以进行系统级别范围的通信。

为了获取通知，添加所要指定监听消息名的观察者到通知发布中心，当消息接收到的时候函数指针指向的函数将被执行一次：

**static** **void** **Callback**(CFNotificationCenterRef center,

**void** \*observer,

CFStringRef name,

**const** **void** \***object**,

CFDictionaryRef userInfo)

{

// ...

}

CFNotificationCenterRef distributedCenter =

CFNotificationCenterGetDistributedCenter();

CFNotificationSuspensionBehavior behavior =

CFNotificationSuspensionBehaviorDeliverImmediately;

CFNotificationCenterAddObserver(distributedCenter,

NULL,

Callback,

CFSTR("notification.identifier"),

NULL,

behavior);

发送端代码更为简单，只要配置好ID,对象还有user info：

**void** \***object**;

CFDictionaryRef userInfo;

CFNotificationCenterRef distributedCenter =

CFNotificationCenterGetDistributedCenter();

CFNotificationCenterPostNotification(distributedCenter,

CFSTR("notification.identifier"),

**object**,

userInfo,

**true**);

链接两个应用通信的方式中，分发式通知是最为简单的。用它来进行大量数据的传输是不明智的，但是对于轻量级信息同步，分发式通知堪称完美。

Distributed Objects

90年代中NeXT全盛时期，分发式对象（DO）是Cocoa框架中一个远程消息发送特性。尽管现在已经不再大范围的使用，在现代奇数层上IPC无障碍通信仍然并未实现。

使用DO分发一个对象仅仅是搭建一个NSConnection并将其注册为特殊（你分的清楚）的名字：

**@protocol** **Protocol**;

**id** <Protocol> vendedObject;

NSConnection \*connection = [[NSConnection alloc] init];

[connection setRootObject:vendedObject];

[connection registerName:@"server"];

另外一个应用将会也建立同样名字的并注册过的链接，然后立即获取一个原子代理当做原始对象。

**id** proxy = [NSConnection rootProxyForConnectionWithRegisteredName:@"server" host:nil];

[proxy setProtocolForProxy:**@protocol**(**Protocol**)];

只要分发对象代理收到消息了，一个通过NSConnection连接远程调用（RPC）将会根据发送对象进行对应的计算并且返回结果给代理。【注：原理是一个OS管理的共享的NSPortNameServer实例对这个带着名字的连接进行管控。】

分发式对象简单，透明，健壮。简直就是Cocoa中的标杆。。。

实际上，分布式对象不能像局部对象那样使用，那就是因为任何发送给代理的消息都可能抛出异常。不想其他语言，OC没有异常处理控制流程。所以对任何东西都进行@try/@catch也算是Cocoa大会很凄凉的补救了。

DO还有一个原因致其使用不便。在试图通过连接“marshal values”时，对象和原语的差距尤为明显。 此外，连接是完全加密的，和下方通信信道扩展性的缺乏致使其在大多数的使用中通信被迫中断。

下方是左列分布式对象用来指定其属性代理行为和方法参数的注解：

* in：输入参数，后续不再引用
* out：参数被引用作为返回值
* inout：输入参数，引用作为返回值
* const：常量参数
* oneway：无障碍结果返回
* bycopy：返回对象的拷贝
* byref：返回对象的代理

AppleEvents & AppleScript

**AppleEvents**是经典**Macintosh**操作系统最持久的遗产。在**System 7**推出的**AppleEvents**允许应用程序在本地使用**AppleScript**或者使用程序链接的功能进行程序控制。现在**AppleScript**使用**Cocoa Scripting Bridge**，仍然是OS X应用进程间最直接的交互方式。【注：Mac系统的苹果时间管理中心为**AppleEvents**提供了原始低级传送机制，但是是在OS X的Mach端口基础之上的重实现】。

也就是说，使用起来这是简单而又古怪的技术之一。

**AppleScript**使用自然语言语法，设计初衷是没有涉及参数而更容易掌握。虽然与人交流更亲和了，但是写起来确实噩梦。

为了更好的了解人类自然性，这里有个栗子教你怎么让Safari在最前的窗口的激活栏打开一个URL。

tell application "Safari"

**set** the URL **of** the front document **to** "http://nshipster.com"

**end** tell

在大部分情况下，**AppleScript**的语法自然语言的特性更多是不便不是优势。（吐槽。。。略略略）

即便是经验老道的OC开发者，不靠文档或者栗子写出**AppleScript**是不可能的任务。

幸运的是，**Scripting Bridge**为Cocoa应用提供了更友善的编程接口。

Cocoa Scripting Bridge

为了使用**Scripting Bridge**与应用进行交互，首先要先添加一个编程接口：

$ sdef /Applications/Safari.app | sdp -fh --basename Safari

sdef为应用生成脚本定义文件。这些文件可以以管道输入道sdp并格式转成（在这里是）C头文件。这样的结果是添加该头文件到应用工程并提供第一类对象接口。

这里举个栗子来解释如何使用**Cocoa Scripting Bridge**：

#import "Safari.h"

SafariApplication \*safari = [SBApplication applicationWithBundleIdentifier:@"com.apple.Safari"];

**for** (SafariWindow \*window **in** safari.windows) {

**if** (window.visible) {

window.currentTab.URL = [NSURL URLWithString:@"http://nshipster.com"];

**break**;

}

}

对比**AppleScript**上面显得冗繁了点，但是却更容易集成到已存在的代码中去。在可读性上更优因为毕竟长得更像OC。

唉，**AppleScript**的星芒也正出现消退，在最近发布的OS X与iWork应用证答复减少它的戏份。从这点说，未必值得在你的应用中去添加这项（脚本）支持。

Pasteboard

剪贴板是OS X与iOS最常见的进程间通信机制。当用户跨应用拷贝了一段文字，图片，文档，这时候通过**mach port**的com.apple.pboard服务媒介进行从一个进程到另一个进程的数据交换。

OS X上是NSPasteboard，iOS上对应的是UIPasteboard。它们几乎是别无二致，但尽管大致一样，对比OS X iOS上提供了更简洁，更现代化却又不影响功效的API。

编写剪贴板代码几乎就跟在GUI应用上使用Edit > Copy操作一样简单：

NSImage \*image;

NSPasteboard \*pasteboard = [NSPasteboard generalPasteboard];

[pasteboard clearContents];

[pasteboard writeObjects:@[image]];

因为剪贴动作太频繁了，所以要确认剪贴内容是否是你（应用）所需要得：

NSPasteboard \*pasteboard = [NSPasteboard generalPasteboard];

**if** ([pasteboard canReadObjectForClasses:@[[NSImage **class**]] **options**:**nil**]) {

NSArray \*contents = [pasteboard readObjectsForClasses:@[[NSImage **class**]] **options**: **nil**];

NSImage \*image = [contents firstObject];

}

XPC

XPC是SDK中最先进的进程间通讯技术。它架构之初的目的在于避免长时间得运行过程，来适应有限的资源，在可能运行的时候才进行初始化。把XPC纳入应用而不做任何事情的想法是不现实的，但这样提供了更好的进程间的特权分离和故障隔离。

XPC作为NSTask替代品甚至更多。

2011推出以来，XPC为OS X上的应用沙盒提供基础设施，iOS上的远程试图控制器，还有两个平台上的应用扩展。它还广范围的用在系统框架和第一方应用：

$ find /Applications -name \\*.xpc

控制台输入上面的命令行你会知道XPC无处不在。在一般应用中同样的情形也在发生，比如图片或者视频转变服务，系统调用，网页服务加载，或是第三方的授权。

XPC负责进程间通讯的同时还负责该服务生命周期的管理。包括注册服务，启动，以及通过launchd解决服务之间的通讯。一个XPC服务可以根据需求地洞，或者在崩溃的时候重启，或者是空闲的时候终止。正因如此，服务可以完全被设计成无状态的，以便于在运行的任何时间点的突然终止都能做到影响不大。

作为被iOS还有OS X中**backported**所采用的安全模块，XPC服务默认运行在最为严格的环境：不能访问文件，不能访问网络，没有根权限升级。任何能做的事情就是对照被赋予的白名单列表。

XPC可以被libxpc C API访问，或者是NSXPCConnection OC API。【注：作者会用低级API去实现（纯C）】

XPC服务要么存在于应用的沙盒中亦或是使用launchd调用跑在后台。

服务调用带事件句柄的xpc\_main来获取新的XPC连接。

**static** **void** **connection\_handler**(xpc\_connection\_t peer) {

xpc\_connection\_set\_event\_handler(peer, ^(xpc\_object\_t **event**) {

peer\_event\_handler(peer, **event**);

});

xpc\_connection\_resume(peer);

}

**int** **main**(**int** argc, **const** **char** \*argv[]) {

xpc\_main(connection\_handler);

exit(EXIT\_FAILURE);

}

每个XPC连接是一对一的，意味着服务在不同的连接进行操作，每次调用xpc\_connection\_create就会创建一个新的链接。【注：类似BSD套接字中的API accept函数，服务在单个文件描述符进行监听来为范围内的链接创建额外描述符】：

xpc\_connection\_t c = xpc\_connection\_create("com.example.service", NULL);

xpc\_connection\_set\_event\_handler(c, ^(xpc\_object\_t **event**) {

// ...

});

xpc\_connection\_resume(c);

当一个消息发送到XPC链接，将自动的派发到一个由runtime管理的消息队列中。当链接的远端一旦开启的时候，消息将出队并被发送。

每个消息就是一个字典，字符串key和强类型值：

xpc\_dictionary\_t message = xpc\_dictionary\_create(NULL, NULL, 0);

xpc\_dictionary\_set\_uint64(message, "foo", 1);

xpc\_connection\_send\_message(c, message);

xpc\_release(message)

XPC对象对下列原始类型进行操作：

* Data
* Boolean
* Double
* String
* Signed Integer
* Unsigned Integer
* Date
* UUID
* Array
* Dictionary
* Null

XPC提供了一个便捷的方法来从dispatch\_data\_t数据类型进行转换，这样从GCD到XPC的工作流程就简化了：

**void** \*buffer;

**size\_t** length;

**dispatch\_data\_t** ddata =

dispatch\_data\_create(buffer,

length,

DISPATCH\_TARGET\_QUEUE\_DEFAULT,

DISPATCH\_DATA\_DESTRUCTOR\_MUNMAP);

**xpc\_object\_t** xdata = xpc\_data\_create\_with\_dispatch\_data(ddata);

服务注册

XPC可以注册成启动项任务，配置成匹配IOKit事件自动启动，BSD通知或者是CFDistributedNotifications。这些标准都指定在服务的launchd.plist文件里：  
.launchd.plist

<key>LaunchEvents</key>

<dict>

<key>com.apple.iokit.matching</key>

<dict>

<key>com.example.device-attach</key>

<dict>

<key>idProduct</key>

<integer>2794</integer>

<key>idVendor</key>

<integer>725</integer>

<key>IOProviderClass</key>

<string>IOUSBDevice</string>

<key>IOMatchLaunchStream</key>

<true/>

<key>ProcessType</key>

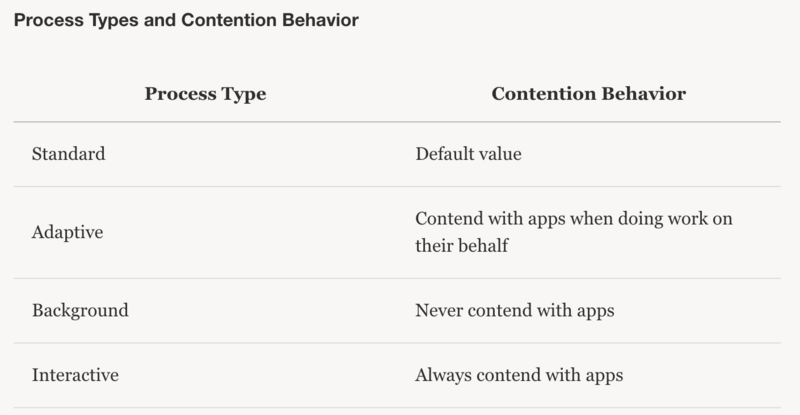
<string>Adaptive</string>

</dict>

</dict>

</dict>

最近一次对于launchd属性列表的修改是增加了ProcessType Key，其用来在高级层面上描述启动机构的预期目的。根据预描述行为期望，操作系统会响应调整CPU和I/O的阈值。



为了注册一个服务运行大概五分钟的时间，一套标准需要传送给xpc\_activity\_register：

xpc\_object\_t criteria = xpc\_dictionary\_create(NULL, NULL, 0);

xpc\_dictionary\_set\_int64(criteria, XPC\_ACTIVITY\_INTERVAL, 5 \* 60);

xpc\_dictionary\_set\_int64(criteria, XPC\_ACTIVITY\_GRACE\_PERIOD, 10 \* 60);

xpc\_activity\_register("com.example.app.activity",

criteria,

^(xpc\_activity\_t activity)

{

// Process Data

xpc\_activity\_set\_state(activity, XPC\_ACTIVITY\_STATE\_CONTINUE);

dispatch\_async(dispatch\_get\_main\_queue(), ^{

// Update UI

xpc\_activity\_set\_state(activity, XPC\_ACTIVITY\_STATE\_DONE);

});

});