一、介绍

注：Objective-C Runtime Programming Guide.

2020-08-31

CJL

一、介绍

重要提示：此文档不再更新。有关Apple SDK的最新信息，请访问 文档网站。

Objective-C语言将尽可能多的决策从编译时和链接时间延迟到运行时。只要有可能，它都是动态的。这意味着该语言不仅需要编译器，还需要运行时系统来执行编译后的代码。运行时系统作为Objective-C语言的一种操作系统；它使语言工作。

这个文档介绍了NSObject类以及Objective-C程序如何与运行时系统交互。特别是，它检查了在运行时动态加载新类和将消息转发到其他对象的范例。它还提供有关如何在程序运行时查找有关对象的信息的信息跑步。

你应该阅读本文档，以了解Objective-C运行时系统的工作原理以及如何利用它。不过，通常情况下，编写Cocoa应用程序时不需要了解和理解这些内容

文档结构

本文档包括以下章节:

- 运行时版本和平台
- 与运行时交互
- 消息传递
- 动态方法解析
- 消息转发
- 类型编码
- 声明属性

另请参阅

Objective-C Runtime Reference 描述Objective-C运行时支持库的数据结构和函数。您的程序可以使用这些接口与Objective-C运行时系统进行交互。例如，您可以添加类或方法，或获取要加载的所有类定义的列表类。.

Programming with Objective-C 描述Objective-C语言.

Objective-C Release Notes 描述了OSX最新版本中Objective-C运行时的一些变化.

二、运行时版本和平台

Objective-C运行时在不同的平台上有不同的版本。

Legacy and Modern 版本

Objective-C运行时有两个版本-“modern”和“legacy”。modern版本是在Objective-c2.0中引入的，它包含了许多新特性。legacy版本的运行时的编程接口在Objective-c1运行时参考中描述；现代版本的运行时的编程接口在*Objective-C 运行时参考*中描述。

最值得注意的新特性是，现代运行时中的实例变量是“非脆弱的”：

- 在legacy runtime中，如果更改类中实例变量的布局，则必须重新编译继承自它。
- 在modern runtime中，如果更改类中实例变量的布局，则不必重新编译继承自它。。

此外，现代运行时支持声明属性的实例变量合成（请参阅 *Objective-C 编程语言*中的 声明属性）

平台

在OS X v10.5及更高版本上的iPhone应用程序和64位程序使用modern 版本的运行时间。

其他程序（OSX桌面上的32位程序）使用 legacy 版本的运行库。

三、与运行时交互

Objective-C程序在三个不同的层次与运行时系统交互：通过Objective-C源码；通过在基础框架的NSObject类中定义的方法；通过直接调用运行时函数。

Objective-C 源码

在大多数情况下，运行时系统在后台自动工作。您只需编写和编译Objective-C源代码就可以使用它。

当您编译包含Objective-C类和方法的代码时，编译器将创建实现语言动态特性的数据结构和函数调用。数据结构捕获类和类别定义以及协议声明中的信息；它们包括在用 *Objective-C 编程语言* 定义类和 协议时讨论的类和协议对象，以及方法选择器、实例变量模板和从源代码中提取的其他信息。主运行时函数是发送消息的函数，如 消息传递中所述。它由源代码消息表达式调用。

NSObject 方法

Cocoa中的大多数对象都是**NSObject**类的子类，因此大多数对象都继承它定义的方法。（一个值得注意的例外是**NSProxy**类；有关更多信息，请参阅 消息转发。）因此，它的方法建立了每个实例和每个类对象固有的行为。然而，在一些情况下，**NSObject**类只定义了一个模板，用于说明应该如何完成某件事；它本身并没有提供所有必需的代码。

例如，**NSObject**类定义了一个**description**实例方法，该方法返回一个描述类内容的字符串。这主要用于调试-GDB **print object**命令打印从该方法返回的字符串。这个方法的**NSObject**实现不知道类包含什么，所以它返回一个带有对象名称和地址的字符串。**NSObject**的子类可以实现此方法以返回更多详细信息。例如，基础类NSArray返回它包含的对象的描述列表。

有些**NSObject**方法只是查询运行时系统的信息。这些方法允许对象执行自省。此类方法的示例有类方法，它要求对象标识其类；**isKindOfClass:**和**isMemberOfClass:**，用于测试对象在继承层次结构中的位置；**respondsToSelector:**，指示对象是否可以接受特定消息；**conformsToProtocol:**，它指示对象是否声明实现特定协议中定义的方法；**methodForSelector:**，它提供方法实现的地址。像这样的方法给对象提供了自我反省的能力。

运行时函数

运行时系统是一个动态共享库，其公共接口由位于目录**/usr/include/objc**中的头文件中的一组函数和数据结构组成。这些函数中的许多都允许您使用纯C来复制编写Objective-C代码时编译器所做的工作。其他的则是通过**NSObject**类的方法导出的功能的基础。这些函数使得

开发运行时系统的其他接口和生成扩展开发环境的工具成为可能；在Objective-C中编程时不需要它们。但是，在编写Objective-C程序时，一些运行时函数可能会有用。所有这些函数都记录在*Objective-C 运行时参考*中。

四、消息传递

本章介绍如何将消息表达式转换为 **objc_msgSend**函数调用，以及如何按名称引用方法。然后解释如何利用**objc_msgSend**，以及如果需要，如何绕过动态绑定。

objc_msgSend 函数

在Objective-C中，消息直到运行时才绑定到方法实现。编译器转换消息表达式，

```
[receiver message]
```

调用消息传递函数 objc_msgSend。此函数将接收方和消息中提到的方法的名称（即方法选择器）作为其两个主要参数：

```
objc_msgSend(receiver, selector)
```

消息中传递的任何参数也将传递给objc_msgSend：

```
objc_msgSend(receiver, selector, arg1, arg2, ...)
```

消息传递函数为动态绑定：

- 先绑定查找选择器引用的过程（方法实现）。由于同一方法可以由不同的类实现，因此它找到的精确过程取决于接受者。
- 它然后调用过程，将接收对象（指向其数据的指针）以及为方法。
- 最后，它传递过程的返回值作为自己的回报值。

注意：编译器生成对消息传递函数的调用。永远不要在编写的代码中直接调用它。

消息传递的关键在于编译器为每个类和对象构建的结构。每个类结构都包含以下两个基本元素：

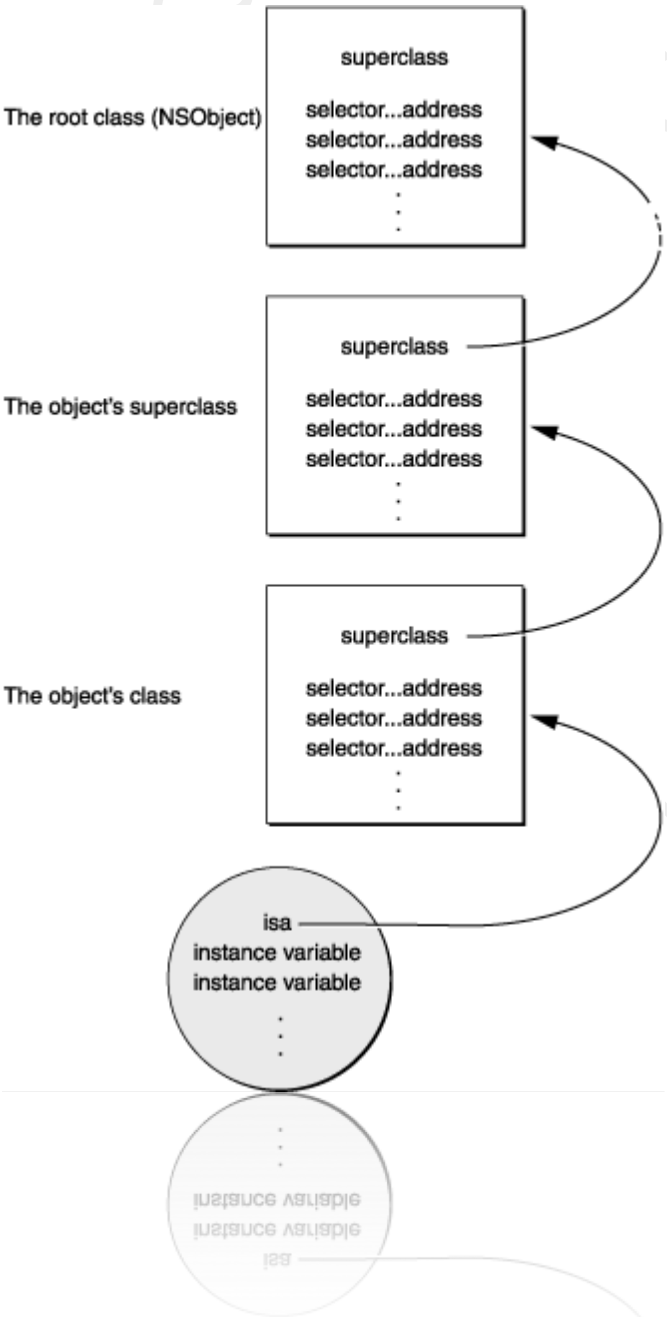
- 指向超类的指针。
- 一个类调度表。此表中的条目将方法选择器与其标识的方法的类特定地址相关联。**setOrigin:**：方法的选择器与**setOrigin:**：（实现的过程）的地址相关联，**display**方法的选择器与display的地址关联，依此类推。

创建新对象时，将为其分配内存，并初始化其实例变量。对象变量中的第一个变量是指向其类结构的指针。这个名为**isa**的指针让对象访问其类，并通过该类访问它继承的所有类。

注意：虽然**isa**指针不是严格意义上的语言的一部分，但它是对象与Objective-C运行时系统一起工作所必需的。在结构定义的任何字段中，对象都需要与**`struct objc_object`**（在**objc/objc.h**中定义）“等效”。但是，您很少需要创建自己的根对象，并且从**NSObject**或**NSProxy**继承的对象自动具有isa变量。

这些类元素和对象结构如图3-1所示。

Figure 3-1 Messaging Framework



当消息发送到对象时，消息传递函数跟随对象的**isa**指针指向类结构，在该类结构中查找调度表中的方法选择器。如果在那里找不到选择器，**objc_msgSend**会跟随指向超类的指针并尝试在其调度表中查找选择器。连续的失败导致**objc_msgSend**爬升类层次结构，直到到达**NSObject**类。一旦找到选择器，函数就会调用表中输入的方法，并将接收对象的数据结构传递给它。

这就是在运行时选择方法实现的方式，或者用面向对象编程的行话来说，方法是动态绑定到消息的。

为了加快消息传递过程，运行时系统在使用方法时缓存选择器和地址。每个类都有一个单独的缓存，它可以包含继承方法的选择器以及类中定义的方法的选择器。在搜索调度表之前，消息传递例程首先检查接收对象类的缓存（理论上，曾经使用过一次的方法可能会再次使用）。如果方法选择器在缓存中，则消息传递只比函数调用稍慢。一旦一个程序运行足够长的时间来“预热”它的缓存，它发送的几乎所有消息都会找到一个缓存方法。缓存动态增长以适应程序运行时的新消息。

使用隐藏参数

当 `objc_msgSend`找到实现方法的过程时，它调用该过程并将消息中的所有参数传递给它。它还向过程传递两个隐藏参数：

- 接收对象
- 方法的选择器

这些参数为每个方法实现提供关于调用它的消息表达式的两部分的显式信息。它们被称为“隐藏”，因为它们没有在定义方法的源代码中声明。它们在代码编译时被插入到实现中。

虽然这些参数没有显式声明，但是源代码仍然可以引用它们（就像它可以引用接收对象的实例变量一样）。方法将接收对象引用为**self**，并将其自己的选择器引用为**_cmd**。在下面的示例中，**_cmd**表示异常方法的选择器，**self**指向接收到异常消息的对象。

```
- strange
{
     //_cmd：表示异常的选择器
     //self：指向接收到异常消息的对象
    id  target = getTheReceiver();
    SEL method = getTheMethod();

    if ( target == self || method == _cmd )
        return nil;
    return [target performSelector:method];
}
```

在这两个论点中，self更有用。实际上，这是接收对象的实例变量可用于方法定义的方式

获取方法地址

规避动态绑定的唯一方法是获取方法的地址，然后像函数一样直接调用它。当一个特定的方法将被连续执行很多次，并且您希望避免每次执行该方法时消息传递的开销，这种情况可能比较合适。

使用**NSObject**类中定义的方法**methodForSelector:**，可以请求指向实现方法的过程的指针，然后使用该指针调用该过程。**methodForSelector:**返回的指针必须谨慎地转换为正确的函数类型。类型转换中应包括返回类型和参数类型。

下面的示例显示如何调用实现**setFilled:**方法的过程：

```
void (*setter)(id, SEL, BOOL);
int i;

setter = (void (*)(id, SEL, BOOL))[target
    methodForSelector:@selector(setFilled:)];
for ( i = 0 ; i < 1000 ; i++ )
    setter(targetList[i], @selector(setFilled:), YES);
```

传递给过程的前两个参数是接收对象（**self**）和方法选择器（**_cmd**）。这些参数隐藏在方法语法中，但必须在方法作为函数调用时显式。

使用**methodForSelector:**绕过动态绑定可以节省消息传递所需的大部分时间。但是，只有在特定的消息被重复多次时，节省的空间才会显著，如上面所示的for循环中。

注意**methodForSelector:**是由Cocoa运行时系统提供的；它不是Objective-C语言本身的特性。

五、动态方法解析

本章描述如何动态地提供方法的实现。

动态方法解析

在某些情况下，您可能希望动态地提供方法的实现。例如，Objective-C声明属性特性（请参阅 *Objective-C* 编程语言中的 声明属性）包括@dynamic指令：

```
@dynamic propertyName;
```

它告诉编译器将动态提供与属性关联的方法。

可以实现 `resolveInstanceMethod:`和 `resolveClassMethod:` 方法，分别为实例和类方法的给定选择器动态提供实现。

Objective-C方法只是一个C函数，它至少有两个参数self和_cmd。可以使用函数class_addMethod将函数作为方法添加到类中。因此，考虑到以下功能：

```
void dynamicMethodIMP(id self, SEL _cmd) {
    // implementation ....
}
```

可以使用**resolveInstanceMethod**将其作为方法（称为**ResolveThisMethodDynamic**）动态添加到类中，如下所示：

```
@implementation MyClass
+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
     if (aSEL == @selector(resolveThisMethodDynamically)) {
          class_addMethod([self class], aSEL, (IMP)
dynamicMethodIMP, "v@:");
          return YES;
      }
      return [super resolveInstanceMethod:aSEL];
}
@end
```

转发方法（如消息转发中所述）和动态方法解析在很大程度上是正交的。在转发机制生效之前，类有机会动态解析方法。如果调用了respondsToSelector:或instanceRespondToSelector:，则动态方法解析器将有机会首先为选择器提供IMP。如果实现resolveInstanceMethod:但希望通过转发机制实际转发特定的选择器，则为这些选择器返回NO。

动态加载

Objective-C程序可以在运行时加载和链接新的类和类别。新代码被合并到程序中，并与开始时加载的类和类别相同。

动态加载可以用来做很多不同的事情。例如，系统首选项应用程序中的各个模块是动态加载的。

在Cocoa环境中，通常使用动态加载来定制应用程序。其他人可以编写程序在运行时加载的模块，就像Interface Builder加载自定义调色板和OSX系统首选项应用程序加载自定义首选项模块一样。可加载模块扩展了应用程序的功能。他们以你所允许的方式为之做出贡献，但却无法预料或定义你自己。您提供框架，但其他人提供代码。

尽管有一个运行时函数可以在**Mach-O**文件（**objc_loadModules**，在**objc/objc load.h**中定义）中执行Objective-C模块的动态加载，但是Cocoa的NSBundle类为动态加载提供了一个非常方便的接口，该接口面向对象并与相关服务集成。有关**NSBundle**类及其用法的信息，请参阅《基础框架参考》中的`NSBundle`类规范。有关Mach-O文件的信息，请参阅OS X ABI Mach-O文件格式参考。

六、消息转发

向不处理该消息的对象发送消息是错误的。但是，在宣布错误之前，运行时系统会给接收对象第二次处理消息的机会。

转发

如果将消息发送到不处理该消息的对象，则在宣布错误之前，运行时会向该对象发送一个**forwardInvocation:message**，其中**NSInvocation**对象作为其唯一参数，**NSInvocation**对象将封装原始消息及其传递的参数。

您可以实现**forwardInvocation:**方法来给消息提供默认响应，或者以其他方式避免错误。顾名思义，**forwardInvocation:**通常用于将消息转发到另一个对象。

要了解转发的范围和意图，请设想以下场景：首先，假设您正在设计一个可以响应名为negotiate的消息的对象，并且希望其响应包含另一种对象的响应。通过将协商消息传递给所实现的协商方法主体中的其他对象，可以很容易地完成此操作。

更进一步，假设您希望对象对**negotiate** 消息的响应与在另一个类中实现的响应完全相同。实现这一点的一种方法是让您的类从另一个类继承该方法。然而，这样安排事情可能是不可能的。您的类和实现**negotiate**的类位于继承层次结构的不同分支中可能有很好的原因。

即使您的类不能继承协商方法，您仍然可以通过实现该方法的一个版本来“借用”该方法，该方法只需将消息传递给另一个类的实例：

```
- (id)negotiate
{
    if ( [someOtherObject respondsTo:@selector(negotiate)] )
        return [someOtherObject negotiate];
    return self;
}
```

这种方式可能会有点麻烦，特别是如果有很多消息需要对象传递给另一个对象。你必须实现一个方法来覆盖你想从另一个类借用的每个方法。而且，你可能不想知道你在哪里写了完整的代码。该集合可能依赖于运行时的事件，并且在将来实现新方法和类时可能会发生变化。

forwardInvocation提供的第二个机会是：message为这个问题提供了一个不那么特别的解决方案，而且是动态的，而不是静态的。它的工作原理是这样的：当一个对象因为没有与消息中的选择器匹配的方法而无法响应消息时，运行时系统通过发送**forwardInvocation:message**通知对象。每个对象都从**NSObject**类继承一个**forwardInvocation:**方法。但是，NSObject的方法版本只是调用**doesNotRecognizeSelector**：。通过重写**NSObject**的版本并实现自己的版本，您可以利用**forwardInvocation:message**提供的机会将消息转发到其他对象。

若要转发消息，**forwardInvocation:**方法只需：

- 确定消息的位置
- 并将位置与原始消息一起发送到那里

可以使用**invokeWithTarget:**方法发送消息：

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    if ([someOtherObject respondsToSelector:
            [anInvocation selector]])
        [anInvocation invokeWithTarget:someOtherObject];
    else
        [super forwardInvocation:anInvocation];
}
```

所转发的消息的返回值返回给原始发送方。所有类型的返回值都可以传递给发送方，包括id、结构和双精度浮点数。

forwardInvocation:方法可以充当未识别消息的分发中心，将它们分发给不同的接收方。或者它可以是一个中转站，将所有消息发送到同一个目的地。它可以将一条消息转换成另一条消息，或者简单地“吞下”一些消息，这样就不会有响应，也不会出错。**forwardInvocation:**方法还可以将多个消息合并到单个响应中。**forwardInvocation:**做什么取决于实现者。然而，它为链接转发链中的对象提供了机会，为程序设计提供了可能性。

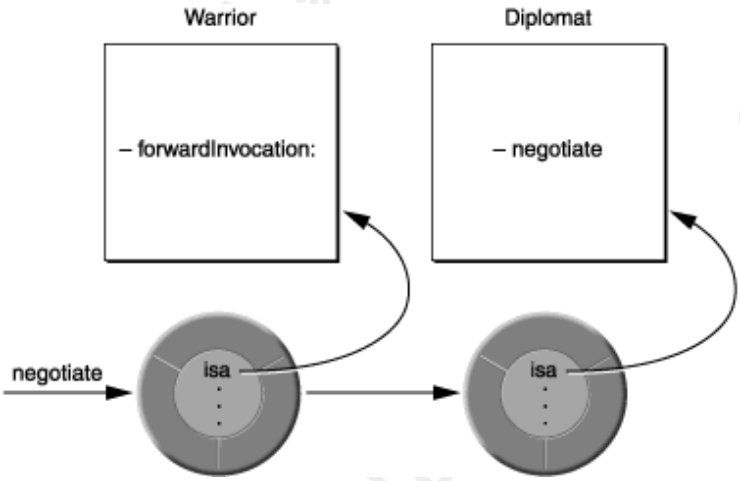
注意：**forwardInvocation:**方法只有在消息没有调用名义接收方中的现有方法时才能处理它们。例如，如果您希望您的对象将**negotiate**消息转发到另一个对象，则它不能有自己的**negotiate**方法。如果是这样，消息就永远不会到达**forwardInvocation:**。

有关转发和调用的更多信息，请参阅《基础框架参考》中的**NSInvocation**类规范。

转发和多重继承

转发模拟继承，可以用于将多重继承的一些效果借给Objective-C程序。如图5-1所示，通过转发消息来响应消息的对象似乎借用或“继承”在另一个类中定义的方法实现。

Figure 5-1 Forwarding



在本例中，Warrior（战士）类的一个实例将**`negotiate`** 消息转发给Diplomat（外交官）类的实例。这位战士看起来像个外交官一样谈判。它似乎对**`negotiate`** 信息做出了回应，而且出于所有实际目的，它确实做出了回应（尽管这项工作实际上是一名外交官在做）。

因此，转发消息的对象从继承层次结构的两个分支“继承”方法它自己的分支和响应消息的对象的分支。在上面的例子中，Warrior类似乎继承了depolator以及它自己的超类。

转发提供了您通常希望从多重继承中获得的大多数功能。然而，两者之间有一个重要的区别：多重继承在一个对象中结合了不同的功能。它倾向于大的，多方面的对象。另一方面，转发将单独的责任分配给不同的对象。它将问题分解为更小的对象，但以对消息发送者透明的方式关联这些对象。

代理对象

转发不仅模拟多重继承，还可以开发表示或“覆盖”更重要对象的轻量级对象。代理代表另一个对象并将消息传递给它。

在 *Objective-C 编程语言*的“远程消息传递”中讨论的代理就是这样一个代理。代理负责将消息转发到远程接收方的管理细节，确保参数值在连接中被复制和检索，等等。但它不尝试做其他事情；它不复制远程对象的功能，而只是给远程对象一个本地地址，一个它可以在另一个应用程序中接收消息的地方。

其他类型的代理对象也是可能的。例如，假设您有一个对象可以处理大量数据—可能它会创建一个复杂的图像或读取磁盘上文件的内容。设置此对象可能很耗时，因此您更喜欢在真正需要它或系统资源暂时空闲时懒洋洋地进行设置。同时，为了使应用程序中的其他对象正常工作，您至少需要此对象的占位符。

在这种情况下，您可以首先创建，而不是完整的对象，而是它的轻量级代理。这个对象可以自己做一些事情，比如回答有关数据的问题，但大多数情况下，它只会为较大的对象保留一个位置，当时间到了，它会将消息转发给它。当代理的**forwardInvocation:**方法第一次接收到发送给另一个对象的消息时，它将确保该对象存在，如果不存在，则会创建该对象。较大对象的所有消息都经过代理项，因此，就程序的其余部分而言，代理项和较大对象将是相同的。

转发和继承

虽然转发模仿继承，但**NSObject**类从不混淆两者。像**respondsToSelector:**和**isKindOfClass:**这样的方法只查看继承层次结构，而不查看转发链。例如，如果一个Warrior对象被询问是否响应协商消息，

```
if ( [aWarrior respondsToSelector:@selector(negotiate)] )
    ...
```

答案是否定的，尽管它可以毫无差错地接收到谈判信息，并在某种意义上通过将其转发给外交官来作出回应。（见图5-1）

在许多情况下，NO是正确的答案。但也可能不是。如果使用转发来设置代理对象或扩展类的功能，则转发机制应该与继承一样透明。如果您希望对象的行为就像它们真正继承了转发消息的对象的行为一样，则需要重新实现**respondsToSelector:**和**isKindOfClass:**方法，以包含转发算法：

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
if ( [super respondsToSelector:aSelector] )
return YES;
else {
/* Here, test whether the aSelector message can     *
* be forwarded to another object and whether that  *
* object can respond to it. Return YES if it can.  */
}
return NO;
}
```

除了**respondsToSelector:**和**isKindOfClass:**，**instanceRespondToSelector:**方法还应镜像转发算法。如果使用协议，**conformsToProtocol:**方法也应该添加到列表中。类似地，如果一个对象转发它接收到的任何远程消息，它应该有一个**methodSignatureForSelector:**的版本，该版本可以返回对转发消息做出最终响应的方法的准确描述；例如，如果一个对象能够将消息转发到其代理项，则可以实现**methodSignatureForSelector:**如下所示：

```
- (NSMethodSignature*)methodSignatureForSelector:(SEL)selector
{
    NSMethodSignature* signature = [super
methodSignatureForSelector:selector];
    if (!signature) {
       signature = [surrogate
methodSignatureForSelector:selector];
    }
    return signature;
}
```

您可以考虑将转发算法放在私有代码中的某个地方，并将所有这些方法**forwardInvocation:included**，调用它。

注意：这是一种先进的技术，只适用于没有其他解决方案的情况。它不是用来替代继承的。如果必须使用此技术，请确保完全了解执行转发的类和要转发到的类的行为。

本节中提到的方法在基础框架参考中的 NSObject类规范中进行了描述。有关**invokeWithTarget:**的信息，请参阅《基础框架参考》中的 NSInvocation类规范。

七、类型编码

为了帮助运行时系统，编译器将每个方法的返回和参数类型编码为字符串，并将字符串与方法选择器相关联。它使用的编码方案在其他上下文中也很有用，因此可以通过**@encode（）**编译器指令公开使用。当给定类型规范时，**@encode（）**返回对该类型进行编码的字符串。类型可以是基本类型，如int、指针、带标记的结构或联合，也可以是任何类型的类名，事实上，任何类型都可以用作C sizeof（）运算符的参数。

```
char *buf1 = @encode(int **);
char *buf2 = @encode(struct key);
char *buf3 = @encode(Rectangle);
```

下表列出了类型代码。请注意，其中许多代码与您在为存档或分发而对对象进行编码时使用的代码重叠。但是，这里列出了一些您在编写编码器时不能使用的代码，还有一些代码可能需要在编写不是由**@encode（）**生成的编码器时使用。（请参阅《基础框架参考》中的`NSCoder`类规范，以获取有关为存档或分发对象编码的更多信息。）

Table 6-1 Objective-C type encodings

Code	Meaning
`c`	A `char`
`i`	An `int`
`s`	A `short`
`l`	A `long` `l` is treated as a 32-bit quantity on 64-bit programs.
`q`	A `long long`
`C`	An `unsigned char`
`I`	An `unsigned int`
`S`	An `unsigned short`
`L`	An `unsigned long`
`Q`	An `unsigned long long`
`f`	A `float`
`d`	A `double`
`B`	A C++ `bool` or a C99 `_Bool`

`v`	A `void`
`*`	A character string (`char *`)
`@`	An object (whether statically typed or typed `id`)
`#`	A class object (`Class`)
`:`	A method selector (`SEL`)
[*array type*]	An array
{*name*=*type*...}	A structure
(*name*=*type*...)	A union
`b`*num*	A bit field of *num* bits
`^`*type*	A pointer to *type*
`?`	An unknown type (among other things, this code is used for function pointers)

重要提示：Objective-C不支持**long double**类型。@encode（long double）返回d，这与double的编码相同。

数组的类型代码用方括号括起来；数组中的元素数在左括号后紧跟数组类型之前指定。例如，一个由12个指向浮点的指针组成的数组将被编码为：

```
[12^f]
```

结构体在大括号内指定，联合体在括号内指定。首先列出结构标记，然后是等号和按顺序列出的结构字段的代码。例如，一个结构

```
typedef struct example {
    id   anObject;
    char *aString;
    int  anInt;
} Example;
```

编码方式如下：

```
{example=@*i}
```

无论定义的类型名（`Example`）还是结构标记（`Example`）传递给**`@encode()`**，都会产生相同的编码结果。结构指针的编码携带与结构字段相同的信息量：

```
^{example=@*i}
```

但是，另一个间接级别会删除内部类型规范：

```
^^{example}
```

对象被视为结构。例如，将**NSObject**类名传递给**@encode**（）将生成以下编码：

```
{NSObject=#}
```

NSObject类只声明一个class类型的实例变量**isa**。

注意，尽管**@encode**（）指令没有返回它们，但是当类型限定符用于声明协议中的方法时，运行时系统使用表6-2中列出的附加编码。

Table 6-2 Objective-C method encodings：Objective-C方法编码

Code	Meaning
r	const
n	in
N	inout
o	out
O	bycopy
R	byref
V	oneway

八、声明属性

当编译器遇到属性声明（请参阅 *Objective-C* 编程语言中 声明属性）时，它会生成与封闭类、类别或协议相关联的描述性元数据。您可以使用支持在类或协议上按名称查找属性、以**@encode**字符串形式获取属性类型以及以C字符串数组形式复制属性属性列表的函数来访问此元数据。声明的属性列表可用于每个类和协议。

属性类型和功能

Property结构定义属性描述符的不透明句柄。

```
typedef struct objc_property *Property;
```

可以使用函数**class_copyPropertyList**和**protocol_copyPropertyList**分别检索与类（包括加载的类别）和协议相关联的属性数组：

```
objc_property_t *class_copyPropertyList(Class cls, unsigned int *outCount)
objc_property_t *protocol_copyPropertyList(Protocol *proto, unsigned int *outCount)
```

例如，给定以下类声明：

```
@interface Lender : NSObject {
    float alone;
}
@property float alone;
@end
```

可以使用以下方法获取属性列表：

```
id LenderClass = objc_getClass("Lender");
unsigned int outCount;
objc_property_t *properties = class_copyPropertyList(LenderClass, &outCount);
```

```
id LenderClass = objc_getClass("Lender");
unsigned int outCount, i;
objc_property_t *properties = class_copyPropertyList(LenderClass, &outCount);
```

```
for (i = 0; i < outCount; i++) {
    objc_property_t property = properties[i];
    fprintf(stdout, "%s %s\n", property_getName(property),
property_getAttributes(property));
}
```

您可以使用**property_getName**函数来发现属性的名称：

```
const char *property_getName(objc_property_t property)
```

可以使用函数**class_getProperty**和**protocol_getProperty**分别获取对类和协议中给定名称的属性的引用：

```
objc_property_t class_getProperty(Class cls, const char *name)
objc_property_t protocol_getProperty(Protocol *proto, const char *name, BOOL
isRequiredProperty, BOOL isInstanceProperty)
```

可以使用**property_getAttributes**函数来发现属性的名称和**@encode** type字符串。有关编码类型字符串的详细信息，请参阅 类型编码；有关此字符串的详细信息，请参阅 Property Type String和 Property Attribute Description Examples。

```
const char *property_getAttributes(objc_property_t property)
```

将这些组合在一起，可以使用以下代码打印与类关联的所有属性的列表：

```
id LenderClass = objc_getClass("Lender");
unsigned int outCount, i;
objc_property_t *properties = class_copyPropertyList(LenderClass,
&outCount);
for (i = 0; i < outCount; i++) {
objc_property_t property = properties[i];
fprintf(stdout, "%s %s\n", property_getName(property),
property_getAttributes(property));
}
```

属性类型字符串

可以使用**property_getAttributes**函数来发现属性的名称、**@encode** type字符串以及属性的其他属性

字符串以T开头，后跟@encode类型和逗号，以V结尾，后跟支持实例变量的名称。其中，属性由以下描述符指定，用逗号分隔：

Table 7-1 Declared property type encodings:声明的属性类型编码

Code	Meaning
`R`	The property is read-only (`readonly`).属性为只读（只读）。
`C`	The property is a copy of the value last assigned (`copy`).
`&`	The property is a reference to the value last assigned (`retain`).
`N`	The property is non-atomic (`nonatomic`).
`G<name>`	The property defines a custom getter selector name. The name follows the `G` (for example, `GcustomGetter,`).
`S<name>`	The property defines a custom setter selector name. The name follows the `S` (for example, `ScustomSetter:,`).
`D`	The property is dynamic (`@dynamic`).
`W`	The property is a weak reference (`__weak`).
`P`	The property is eligible for garbage collection.
`t<encoding>`	Specifies the type using old-style encoding.

示例，请参见 Property Attribute Description Examples.

Property Attribute Description Examples

根据这些定义：

```
enum FooManChu { FOO, MAN, CHU };
struct YorkshireTeaStruct { int pot; char lady; };
typedef struct YorkshireTeaStruct YorkshireTeaStructType;
union MoneyUnion { float alone; double down; };
```

下表显示了示例属性声明和属性**`property_getAttributes`**返回的相应字符串：

Property declaration	Property description
`@property char charDefault;`	`Tc,VcharDefault`

`@property double doubleDefault;`	`Td,VdoubleDefault`
`@property enum FooManChu enumDefault;`	`Ti,VenumDefault`
`@property float floatDefault;`	`Tf,VfloatDefault`
`@property int intDefault;`	`Ti,VintDefault`
`@property long longDefault;`	`Tl,VlongDefault`
`@property short shortDefault;`	`Ts,VshortDefault`
`@property signed signedDefault;`	`Ti,VsignedDefault`
`@property struct YorkshireTeaStruct structDefault;`	`T{YorkshireTeaStruct="pot"i"lady"c},VstructDefault`
`@property YorkshireTeaStructType typedefDefault;`	`T{YorkshireTeaStruct="pot"i"lady"c},VtypedefDefault`
`@property union MoneyUnion unionDefault;`	`T(MoneyUnion="alone"f"down"d),VunionDefault`
`@property unsigned unsignedDefault;`	`TI,VunsignedDefault`
`@property int (*functionPointerDefault)(char *);`	`T^?,VfunctionPointerDefault`
`@property id idDefault;` Note: the compiler warns: "`no 'assign', 'retain', or 'copy' attribute is specified - 'assign' is assumed`"	`T@,VidDefault`
`@property int *intPointer;`	`T^i,VintPointer`
`@property void *voidPointerDefault;`	`T^v,VvoidPointerDefault`

`@property int intSynthEquals;`
 	
In the implementation block:
 	
`@synthesize intSynthEquals=_intSynthEquals;`
	`Ti,V_intSynthEquals`

`@property(getter=intGetFoo, setter=intSetFoo:) int intSetterGetter;`	`Ti,GintGetFoo,SintSetFoo:,VintSetterGetter`
`@property(readonly) int intReadonly;`	`Ti,R,VintReadonly`
`@property(getter=isIntReadOnlyGetter, readonly) int intReadonlyGetter;`	`Ti,R,GisIntReadOnlyGetter`
`@property(readwrite) int intReadwrite;`	`Ti,VintReadwrite`
`@property(assign) int intAssign;`	`Ti,VintAssign`
`@property(retain) id idRetain;`	`T@,&,VidRetain`
`@property(copy) id idCopy;`	`T@,C,VidCopy`
`@property(nonatomic) int intNonatomic;`	`Ti,VintNonatomic`

`@property(nonatomic, readonly, copy) id idReadonlyCopyNonatomic;`	`T@,R,C,VidReadonlyCopyNonatomic`
`@property(nonatomic, readonly, retain) id idReadonlyRetainNonatomic;`	`T@,R,&,VidReadonlyRetainNonatomic`