Objective-C 2.0 运行时系统编程指南

概述

本部分包括如下内容：

本文档的组织结构
参考

Objective-C 语言将决定尽可能的从编译和链接时推迟到运行时。只要有可能，Objective-C 总是使用动态的方式来解决问题。这意味着 Objective-C 语言不仅需要一个编译器，同时也需要一个运行时系统来执行编译好的代码。这儿的运行时系统扮演的角色类似于 Objective-C 语言的操作系统，Objective-C 基于该系统来工作。

本文档将具体介绍 NSObject 类以及 Objective-C 程序是如何和运行时系统交互的。特别地，本文档还给出来怎样在运行时动态地加载新类和将消息转发给其它对象的范例，同时也给出了怎样在程序运行时获取对象信息的方法。

通常，如果是仅仅写一个 Cocoa 程序，您不需要知道和理解 Objective-C 运行时系统的底层细节，但这篇文档仍然值得推荐您阅读一下，以了解 Objective-C 运行时系统的原理，并能更好的利用 Objective-C 的优点。

本文档的组织结构

本文档包括如下章节：

- “运行时系统的版本和平台”
- 和运行时系统的交互”
- “消息”
- “动态方法解析”
- “消息转发”
- “类型编码”
- “属性声明”

参考

*Objective-C 2.0 运行时系统参考库*描述了Objective-C运行库的数据结构和函数接口。您的程序可以通过这些接口来和Objective-C运行时系统交互。例如，您可以增加一个类或者方法，或者获得所有类的定义列表等。

*Objective-C 2.0 程序设计语言*介绍了Objective-C语言本身。

*Objective-C 版本说明*给出了在最近版本的Mac OS X系统中关于Objecitive-C运行时系统的一些改动。

运行时系统的版本和平台

在不同的平台上 Objective-C 运行时系统的版本也不相同。

本部分包含如下内容：

早期版本和现行版本

Objective-C运行时系统有两个已知版本：早期版本和现行版本。现行版本主要是Objective-C 2.0 及与其相关的新特性。早期版本的编程接口见*Objective-C 1 运行时系统参考库*；现行版本的编程接口见*Objective-C 2.0 运行时系统参考库。*

在现行版本中，最显著的新特性就是实例变量是"健壮"（non-fragile）的：

- 在早期版本中，如果您改变类中实例变量的布局，您必须重新编译该类的所有子类。
- 在现行版本中，如果您改变类中实例变量的布局，您无需重新编译该类的任何子类。

此外，现行版本支持声明property的synthesis属性（参考*Objective-C 2.0 程序设计语言*的属性一节）。

平台

iPhone 程序和 Mac OS X v10.5 及以后的系统中的 64 位程序使用的都是 Objective-C 运行时系统的现行版本。

其它情况（Mac OS X 系统中的 32 位程序）使用的是早期版本。

和运行时系统的交互

Objective-C 程序有三种途径和运行时系统交互：通过 Objective-C 源代码；通过 Foundation 框架中类 NSObject 的方法；通过直接调用运行时系统的函数。

本部分包含如下内容：

通过Objective-C源代码

大部分情况下，运行时系统在后台自动运行，您只需编写和编译 Objective-C 源代码。

当您编译Objective-C类和方法时，编译器为实现语言动态特性将自动创建一些数据结构和函数。这些数据结构包含类定义和协议类定义中的信息，如在*Objective-C 2.0 程序设计语言*中定义类和协议类一节所讨论的类的对象和协议类的对象，方法选标，实例变量模板，以及其它来自于源代码的信息。运行时系统的主要功能就是根据源代码中的表达式发送消息，如"消息"一节所述。

通过类NSObject的方法

Cocoa程序中绝大部分类都是NSObject类的子类，所以大部分都继承了NSObject类的方法，因而继承了NSObject的行为。（NSProxy类是个例外；更多细节参考"*消息转发*"一节。）然而，某些情况下，NSObject类仅仅定义了完成某件事情的模板，而没有提供所有需要的代码。

例如，NSObject 类定义了 description 方法，返回该类内容的字符串表示。这主要是用来调试程序——GDB 中的 print-object 方法就是直接打印出该方法返回的字符串。NSObject 类中该方法的实现并不知道子类中的内容，所以它只是返回类的名字和对象的地址。NSObject 的子类可以重新实现该方法以提供更多的信息。例如，NSArray 类改写了该方法来返回 NSArray 类包含的每个对象的内容。

某些 NSObject 的方法只是简单地从运行时系统中获得信息，从而允许对象进行一定程度的自我检查。例如，class 返回对象的类；isKindOfClass:和 isMemberOfClass:则检查对象是否在指定的类继承体系中；respondsToSelector:检查对象能否响应指定的消息；conformsToProtocol:检查对象是否实现了指定协议类的方法；methodForSelector:则返回指定方法实现的地址。

通过运行时系统的函数

运行时系统是一个有公开接口的动态库，由一些数据结构和函数的集合组成，这些数据结构和函数的声明头文件在/usr/include/objc 中。这些函数支持用纯 C 的函数来实现和 Objective-C 同样的功能。还有一些函数构成了 NSObject 类方法的基础。这些函数使得访问运行时系统接口和提供开发工具成为可能。尽管大部分情况下它们在 Objective-C 程序不是必须的，但是有时候对于 Objecitve-C 程序来说某些函数是非常有用的。 这些函数的文档参见 *Objective-C 2.0 运行时系统参考库*。

消息

本章描述了代码的消息表达式如何转换为对 objc_msgSend 函数的调用，如何通过名字来指定一个方法，以及如何使用 objc_msgSend 函数。

本部分包含如下内容：

获得方法地址

避免动态绑定的唯一办法就是取得方法的地址，并且直接象函数调用一样调用它。当一个方法会被连续调用很多次，而且您希望节省每次调用方法都要发送消息的开销时，使用方法地址来调用方法就显得很有效。

利用 NSObject 类中的 methodForSelector:方法，您可以获得一个指向方法实现的指针，并可以使用该指针直接调用方法实现。methodForSelector:返回的指针和赋值的变量类型必须完全一致，包括方法的参数类型和返回值类型都在类型识别的考虑范围中。

下面的例子展示了怎么使用指针来调用 setFilled:的方法实现：

```
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);
```

方法指针的第一个参数是接收消息的对象（self），第二个参数是方法选标（_cmd）。这两个参数在方法中是隐藏参数，但使用函数的形式来调用方法时必须显示的给出。

使用 methodForSelector:来避免动态绑定将减少大部分消息的开销，但是这只有在指定的消息被重复发送很多次时才有意义，例如上面的 for 循环。

注意，methodForSelector:是 Cocoa 运行时系统的提供的功能，而不是 Objective-C 语言本身的功能。

objc_msgSend函数

在 Objective-C 中，消息是直到运行的时候才和方法实现绑定的。编译器会把一个消息表达式，

```
[receiver message]
```

转换成一个对消息函数 objc_msgSend 的调用。该函数有两个主要参数：消息接收者和消息对应的方法名字——也就是方法选标：

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

同时接收消息中的任意数目的参数：

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

该消息函数做了动态绑定所需要的一切：

- 它首先找到选标所对应的方法实现。因为不同的类对同一方法可能会有不同的实现，所以找到的方法实现依赖于消息接收者的类型。
- 然后将消息接收者对象（指向消息接收者对象的指针）以及方法中指定的参数传给找到的方法实现。
- 最后，将方法实现的返回值作为该函数的返回值返回。

注意：编译器将自动插入调用该消息函数的代码。您无须在代码中显示调用该消息函数。

消息机制的关键在于编译器为类和对象生成的结构。每个类的结构中至少包括两个基本元素：

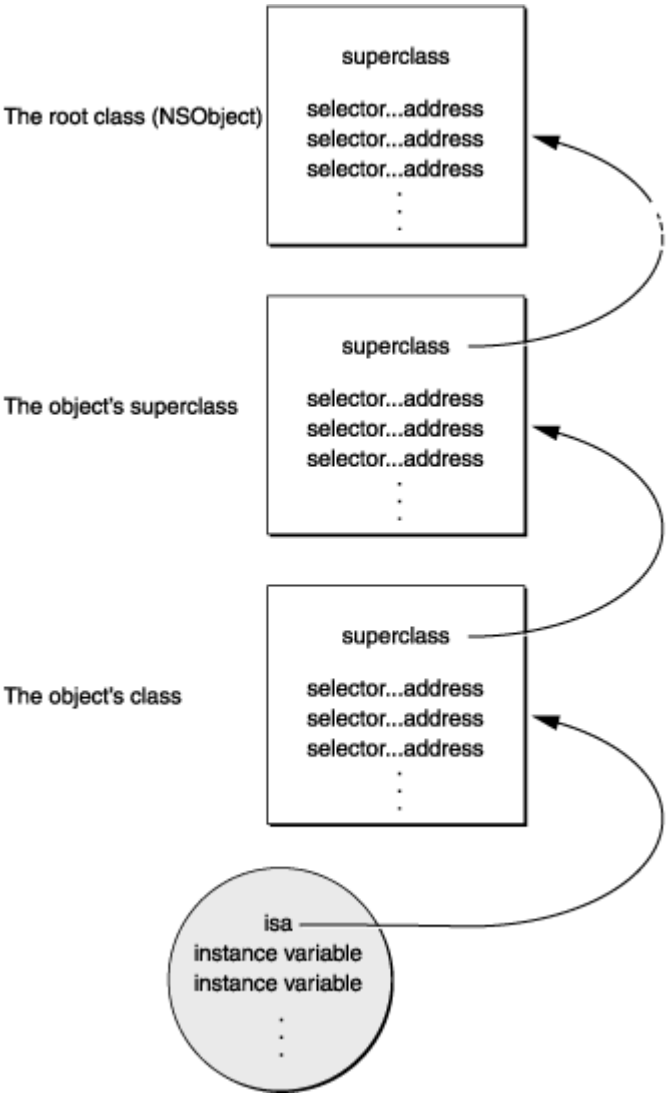
- 指向父类的指针。
- 类的**方法表**。方法表将方法选标和该类的方法实现的地址关联起来。例如，setOrigin::的方法选标和 setOrigin::的方法实现的地址关联，display 的方法选标和 display 的方法实现的地址关联，等等。

当新的对象被创建时，其内存同时被分配，实例变量也同时被初始化。对象的第一个实例变量是一个指向该对象的类结构的指针，叫做 isa。通过该指针，对象可以访问它对应的类以及相应的父类。

注意：尽管严格来说这并不是 Obective-C 语言的一部分，但是在 Objective-C 运行时系统中对象需要有 isa 指针。对象和结构体 struct objc_object（在 objc/objc.h 中定义）必须“一致”。然而，您很少需要创建您自己的根对象，因为从 NSObject 或者 NSProxy 继承的对象都自动包括 isa 变量。

类和对象的结构如图 3-1 所示。

图 3-1 消息框架



当对象收到消息时，消息函数首先根据该对象的 isa 指针找到该对象所对应的类的方法表，并从表中寻找该消息对应的方法选标。如果找不到，objc_msgSend 将继续从父类中寻找，直到 NSObject 类。一旦找到了方法选标， objc_msgSend 则以消息接收者对象为参数调用，调用该选标对应的方法实现。

这就是在运行时系统中选择方法实现的方式。在面向对象编程中，一般称作方法和消息动态绑定的过程。

为了加快消息的处理过程，运行时系统通常会将使用过的方法选标和方法实现的地址放入缓存中。每个类都有一个独立的缓存，同时包括继承的方法和在该类中定义的方法。消息函数会首先检查消息接收者对象对应的类的缓存（理论上，如果一个方法被使用过一次，那么它很可能被再次使用）。如果在缓存中已经有了需要的方法选标，则消息仅仅比函数调用慢一点点。如果程序运行了足够长的时间，几乎每个消息都能在缓存中找到方法实现。程序运行时，缓存也将随着新的消息的增加而增加。

使用隐藏的参数

当 objc_msgSend 找到方法对应的实现时，它将直接调用该方法实现，并将消息中所有的参数都传递给方法实现，同时，它还将传递两个隐藏的参数：

- 接收消息的对象
- 方法选标

这些参数帮助方法实现获得了消息表达式的信息。它们被认为是"隐藏"的是因为它们并没有在定义方法的源代码中声明，而是在代码编译时是插入方法的实现中的。

尽管这些参数没有被显示声明，但在源代码中仍然可以引用它们（就象可以引用消息接收者对象的实例变量一样）。在方法中可以通过 self 来引用消息接收者对象，通过选标_cmd 来引用方法本身。在下面的例子中，_cmd 指的是 strange 方法，self 指的收到 strange 消息的对象。

```
- strange

{

    id  target = getTheReceiver();

    SEL method = getTheMethod();



    if ( target == self || method == _cmd )

        return nil;

    return [target performSelector:method];

}
```

在这两个参数中，self 更有用一些。实际上，它是在方法实现中访问消息接收者对象的实例变量的途径。

动态方法解析

本章将描述怎样动态地提供一个方法的实现。

本部分包含如下内容：

动态方法解析

有时候，您需要动态地提供一个方法的实现。例如，Objective-C中属性（Property）（参考*Objective-C 2.0 程序设计语言*中属性小节）前的修饰符@dynamic

```
@dynamic propertyName;
```

表示编译器须动态地生成该属性对应地方法。

您可以通过实现 resolveInstanceMethod:和 resolveClassMethod:来动态地实现给定选标的对象方法或者类方法。

Objective-C 方法可以认为是至少有两个参数——self 和_cmd—— 的 C 函数。您可以通过 class_addMethod 方法将一个函数加入到类的方法中。例如，有如下的函数：

```
void dynamicMethodIMP(id self, SEL _cmd) {

    // implementation ....

}
```

您可以通过 resolveInstanceMethod:将它作为类方法 resolveThisMethodDynamically 的实现：

```
@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
```

通常消息转发（见 "消息转发"）和动态方法解析是互不相干的。在进入消息转发机制之前，respondsToSelector:和instancesRespondToSelector: 会被首先调用。您可以在这两个方法中为传进来的选标提供一个IMP。如果您实现了resolveInstanceMethod:方法但是仍然希望正常的消息转发机制进行，您只需要返回NO就可以了。

动态加载

Objective-C 程序可以在运行时链接和载入新的类和范畴类。新载入的类和在程序启动时载入的类并没有区别。

动态加载可以用在很多地方。例如，系统配置中的模块就是被动态加载的。

在 Cocoa 环境中，动态加载一般被用来对应用程序进行定制。您的程序可以在运行时加载其他程序员编写的模块——和 Interface Build 载入定制的调色板以及系统配置程序载入定制的模块的类似。 这些模块通过您许可的方式扩展了您的程序，而您无需自己来定义或者实现。您提供了框架，而其它的程序员提供了实现。

尽管已经有一个运行时系统的函数来动态加载Mach-O文件中的Objective-C模块（objc_loadModules，在objc/objc-load.h中定义），Cocoa的NSBundle类为动态加载提供了一个更方便的接口——一个面向对象的，已和相关服务集成的接口。关于NSBundle类的更多相关

信息请参考Foundation框架中关于NSBundle类的文档。关于Mach-O文件的有关信息请参考*Mac OS X ABI Mach-O 文件格式参考库*。

消息转发

通常，给一个对象发送它不能处理的消息会得到出错提示，然而，Objective-C 运行时系统在抛出错误之前，会给消息接收对象发送一条特别的消息来通知该对象。

本部分包含如下内容：：

消息转发

如果一个对象收到一条无法处理的消息，运行时系统会在抛出错误前，给该对象发送一条 forwardInvocation:消息，该消息的唯一参数是个 NSInvocation 类型的对象——该对象封装了原始的消息和消息的参数。

您可以实现 forwardInvocation:方法来对不能处理的消息做一些默认的处理，也可以以其它的某种方式来避免错误被抛出。如 forwardInvocation:的名字所示，它通常用来将消息转发给其它的对象。

关于消息转发的作用，您可以考虑如下情景：假设，您需要设计一个能够响应 negotiate 消息的对象，并且能够包括其它类型的对象对消息的响应。 通过在 negotiate 方法的实现中将 negotiate 消息转发给其它的对象来很容易的达到这一目的。

更进一步，假设您希望您的对象和另外一个类的对象对 negotiate 的消息的响应完全一致。一种可能的方式就是让您的类继承其它类的方法实现。 然后，有时候这种方式不可行，因为您的类和其它类可能需要在不同的继承体系中响应 negotiate 消息。

虽然您的类无法继承其它类的 negotiate 方法，您仍然可以提供一个方法实现，这个方法实现只是简单的将 negotiate 消息转发给其他类的对象，就好像从其它类那儿“借”来的现一样。如下所示：

```
- negotiate

{

    if ( [someOtherObject respondsTo:@selector(negotiate)] )

        return [someOtherObject negotiate];

    return self;

}
```

这种方式显得有欠灵活，特别是有很多消息您都希望传递给其它对象时，您必须为每一种消息提供方法实现。此外，这种方式不能处理未知的消息。当您写下代码时，所有您需要转发的消息的集合也必须确定。然而，实际上，这个集合会随着运行时事件的发生，新方法或者新类的定义而变化。

forwardInvocation:消息给这个问题提供了一个更特别的，动态的解决方案：当一个对象由于没有相应的方法实现而无法响应某消息时，运行时系统将通过 forwardInvocation:消息通知该对象。每个对象都从 NSObject 类中继承了 forwardInvocation:方法。然而，NSObject 中的方法实现只是简单地调用了 doesNotRecognizeSelector:。通过实现您自己的 forwardInvocation:方法，您可以在该方法实现中将消息转发给其它对象。

要转发消息给其它对象，forwardInvocation:方法所必须做的有：

- 决定将消息转发给谁，并且
- 将消息和原来的参数一块转发出去

消息可以通过 invokeWithTarget:方法来转发：

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

转发消息后的返回值将返回给原来的消息发送者。您可以将返回任何类型的返回值，包括: id，结构体，浮点数等。

forwardInvocation:方法就像一个不能识别的消息的分发中心，将这些消息转发给不同接收对象。或者它也可以象一个运输站将所有的消息都发送给同一个接收对象。它可以将一个消息翻译成另外一个消息，或者简单的"吃掉"某些消息，因此没有响应也没有错误。forwardInvocation:方法也可以对不同的消息提供同样的响应，这一切都取决于方法的具体实现。该方法所提供是将不同的对象链接到消息链的能力。

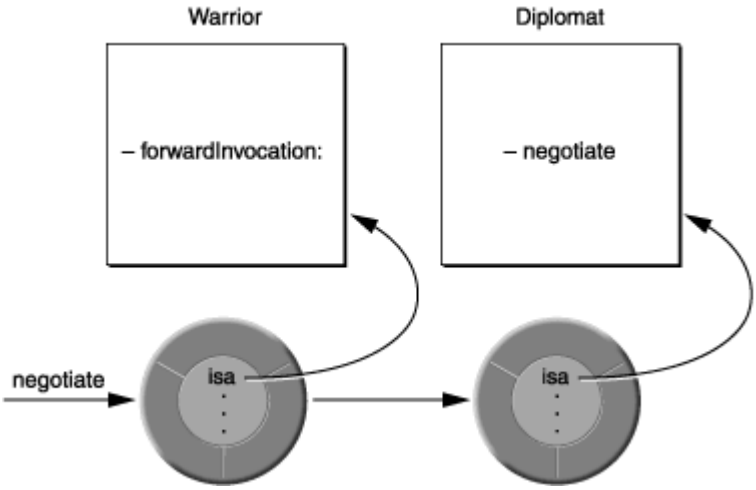
注意： forwardInvocation:方法只有在消息接收对象中无法正常响应消息时才会被调用。 所以，如果您希望您的对象将 negotiate 消息转发给其它对象，您的对象不能有 negotiate 方法。否则，forwardInvocation:将不可能会被调用。

更多消息转发的信息，参考 Foundation 框架参考库中 NSInvocation 类的文档。

消息转发和多重继承

消息转发很象继承，并且可以用来在Objective-C程序中模拟多重继承。如图 5-1所示， 一个对象通过转发来响应消息，看起来就象该对象从别的类那借来了或者"继承"了方法实现一样。

图 5-1 消息转发



在上图中，Warrior 类的一个对象实例将 negotiate 消息转发给 Diplomat 类的一个实例。看起来，Warrior 类似乎和 Diplomat 类一样， 响应 negotiate 消息，并且行为和 Diplomat 一样（尽管实际上是 Diplomat 类响应了该消息）。

转发消息的对象看起来有两个继承体系分支——自己的和响应消息的对象的。在上面的例子中， Warrior 看起来同时继承自 Diplomat 和自己的父类。

消息转发提供了多重继承的很多特性。然而，两者有很大的不同：多重继承是将不同的行为封装到单个的对象中，有可能导致庞大的，复杂的对象。而消息转发是将问题分解到更小的对象中，但是又以一种对消息发送对象来说完全透明的方式将这些对象联系起来。

消息代理对象

消息转发不仅和继承很象，它也使得以一个轻量级的对象（消息代理对象）代表更多的对象进行消息处理成为可能。

*Objective-C 2.0 程序设计语言*中“远程消息”一节中的代理类就是这样一个代理对象。代理类负责将消息转发给远程消息接收对象的管理细节，保证消息参数的传输等等。但是消息类没有进一步的复制远程对象的功能，它只是将远程对象映射到一个本地地址上，从而能够接收其它应用程序的消息。

同时也存在着其它类型的消息代理对象。例如，假设您有个对象需要操作大量的数据——它可能需要创建一个复杂的图片或者需要从磁盘上读一个文件的内容。创建一个这样的对象是很费时的，您可能希望能推迟它的创建时间——直到它真正需要时，或者系统资源空闲时。同时，您又希望至少有一个预留的对象和程序中其它对象交互。

在这种情况下，你可以为该对象创建一个轻量的代理对象。该代理对象可以有一些自己的功能，例如响应数据查询消息，但是它主要的功能是代表某个对象，当时间到来时，将消息转发给被代表的对象。当代理对象的 forwardInvocation: 方法收到需要转发给被代表的对象的消息时，代理对象会保证所代表的对象已经存在，否则就创建它。所有发到被代表的对象的消息都要经过代理对象，对程序来说，代理对象和被代表的对象是一样的。

消息转发和类继承

尽管消息转发很“象”继承，但它不是继承。例如在 NSObject 类中，方法 respondsToSelector: 和 isKindOfClass:只会出现在继承链中，而不是消息转发链中。例如，如果向一个 Warrior 类的对象询问它能否响应 negotiate 消息，

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

    ...
```

返回值是NO，尽管该对象能够接收和响应negotiate。（见图 5-1。）
大部分情况下，NO 是正确的响应。但不是所有时候都是的。例如，如果您使用消息转发来创建一个代理对象以扩展某个类的能力，这儿的消息转发必须和继承一样，尽可能的对用户透明。如果您希望您的代理对象看起来就象是继承自它代表的对象一样，您需要重新实现 respondsToSelector:和 isKindOfClass:方法：

```
- (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:外，instancesRespondToSelector: 方法也必须重新实现。如果您使用的是协议类，需要重新实现的还有 conformsToProtocol:方法。类似地，如果对象需要转发远程消息，则 methodSignatureForSelector:方法必须能够返回实际响应消息的方法的描述。例如，如果对象需要将消息转发给它所代表的对象，您可能需要如下的 methodSignatureForSelector:实现：

```
- (NSMethodSignature*)methodSignatureForSelector:(SEL)selector

{

    NSMethodSignature* signature = [super methodSignatureForSelector:selector];

    if (!signature) {

       signature = [surrogate methodSignatureForSelector:selector];
```

```
    }

    return signature;

}
```

您也可以将消息转发的部分放在一段私有的代码里，然后从 forwardInvocation: 调用它。

注意： 消息转发是一个比较高级的技术，仅适用于没有其它更好的解决办法的情况。它并不是用来代替继承的。如果您必须使用该技术，请确定您已经完全理解了转发消息的类和接收转发消息的类的行为。

本节中涉及的方法在 Foundation 框架参考库中的 NSObject 类的文档中都有描述。关于 invokeWithTarget: 的具体信息，请参考 Foundation 框架参考库中 NSInvocation 类的文档。

类型编码

为了和运行时系统协作，编译器将方法的返回类型和参数类型都编码成一个字符串，并且和方法选标关联在一起。这些编码在别的上下文环境中同样有用，所以您可以直接使用@encode() 编译指令来得到具体的编码。给定一个类型， @encode() 将返回该类型的编码字符串。类型可以是基本类型例如整形，指针，结构体或者联合体，也可以是一个类，就和 C 语言中的 sizeof() 操作符的参数一样，可以是任何类型。

```
char *buf1 = @encode(int **);

char *buf2 = @encode(struct key);

char *buf3 = @encode(Rectangle);
```

下表列出了这些类型编码。注意，它们可能很多和您使用的对象编码有一些重合。然而，这儿列出来的有些编码是您写编码器时候不会使用的，也有一些不是@encode() 产生的，但是在您写编码器的时候是会使用的。（关于对象编码的更多信息，请参考Foundation框架参考库中的NSCoder类文档。）

表 6-1 Objective-C 类型编码

编码	含义
c	char
i	int
s	short
l	long 在 64 位程序中，l 为 32 位。
q	long long
C	unsigned char
I	unsigned int
S	unsigned short

L	unsigned long
Q	unsigned long long
f	float
d	double
B	C++标准的 bool 或者 C99 标准的_Bool
v	void
*	字符串（char *）
@	对象（无论是静态指定的还是通过 id 引用的）
#	类（Class）
:	方法选标（SEL）
[*array type*]	数组
{*name=type...*}	结构体
(*name=type...*)	联合体
bnum	*num* 个 bit 的位域
^type	*type* 类型的指针
?	未知类型（其它时候，一般用来指函数指针）

重要： Objective-C 不支持 long double 类型。 @encode(long double)和 double 一样，返回的字符串都是 d。

数组的类型编码以方括号来表示，紧接着左方括号的是数组元素的数量，然后是数据元素的类型。例如，一个 12 个浮点数（floats）指针的数组可以表示如下：

```
[12^f]
```

结构体和联合体分别用大括号和小括号表示。括号中首先是结构体标签，然后是一个=符号，接着是结构体中各个成员的编码。例如，结构体

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

```
} Example;
```

的编码如下：

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

定义的类型名（Example）和结构体标签（example）有同样的编码结果。指向结构体类型的指针的编码同样也包含了结构体内部数据成员的编码信息，如下所示：

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

然而，更高层次的间接关联就没有了内部数据成员的编码信息：

```
^^{example}
```

对象的编码类似结构体。 例如， @encode() 对 NSObject 编码如下：

```
{NSObject=#}
```

NSObject 类仅声明了一个 Class 类型的实例变量，isa。

注意，尽管有一些编码无法从 @encode() 的结果中直接得到，但是运行时系统会使用它们来表示协议类中方法的修饰符，这些编码如表 6-2 所示。

表 6-2 Objective-C 方法编码

编码	含义
r	const
n	in
N	inout
o	out
O	bycopy
R	byref
V	oneway

属性声明

当编译器遇到一个属性（Property）声明时（参考*Objective-C 2.0 程序设计语言*中的属性小节），编译器将产生一些描述性的元数据与属性所在的类或者协议类关联。您可以通过函数访问元数据，这些函数支持在类或者协议类中通过名字来查找，通过@encode获得属性的类型编码，将属性的特征（Attribute）作为C字符串的数组返回等。每个类或者协议类都维护了一个声明了的属性列表。

本部分包含如下内容：

属性类型和相关函数

属性（Property）类型定义了对描述属性的结构体 objc_property 的不透明的句柄。

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

您可以使用函数 class_copyPropertyList 和 protocol_copyPropertyList 来获得类（包括范畴类）或者协议类中的属性列表：

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

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

例如，有如下的类声明：

```
@interface Lender : NSObject {

    float alone;

}

@property float alone;

@end
```

您可以象这样获得它的属性：

```
id LenderClass = objc_getClass("Lender");

unsigned int outCount;

objc_property_t *properties = class_copyPropertyList(LenderClass, &outCount);
```

您还可以通过 property_getName 函数获得属性的名字：

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

函数 class_getProperty 和 protocol_getProperty 则在类或者协议类中返回具有给定名字的属性的引用：

```
objc_property_t class_getProperty(Class cls, const char *name)

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

通过property_getAttributes函数可以获得属性的名字和@encode编码。关于类型编码的更多细节，参考“类型编码“一节；关于属性的类型编码，见“属性类型编码”及“属性特征的描述范例”。

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

综合起来，您可以通过下面的代码得到一个类中所有的属性。

```
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));

}
```

属性类型编码

property_getAttributes 函数将返回属性（Property）的名字，@encode 编码，以及其它特征（Attribute）。

- property_getAttributes 返回的字符串以字母 T 开始，接着是@encode 编码和逗号。
- 如果属性有 readonly 修饰，则字符串中含有 R 和逗号。
- 如果属性有 copy 或者 retain 修饰，则字符串分别含有 C 或者&，然后是逗号。
- 如果属性定义有定制的 getter 和 setter 方法，则字符串中有 G 或者 S 跟着相应的方法名以及逗号（例如，GcustomGetter，ScustomSetter:，）。

  如果属性是只读的，且有定制的 get 访问方法，则描述到此为止。

- 字符串以 V 然后是属性的名字结束。

范例请参考 “属性特征的描述范例”一节。

属性特征的描述范例

给定如下定义：

```
enum FooManChu { FOO, MAN, CHU };

struct YorkshireTeaStruct { int pot; char lady; };

typedef struct YorkshireTeaStruct YorkshireTeaStructType;

union MoneyUnion { float alone; double down; };
```

下表给出了属性（Property）声明以及 property_getAttributes 返回的相应的字符串：

属性声明	属性描述
@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