第2章 Cocoa的可选语言

大多数人想起 Cocoa 的时候，他们想到的是 Objective-C。在 NeXT 时代，Objective-C 是在 OpenStep 平台下开发程序的唯一途径。在 OS X 10.0 版本中，系统也支持 Java，近来的版本还加入了对其他语言的支持，但 Objective-C 还是 Cocoa 开发的标准语言。

Objective-C 本身已经过了多年的演化。在 NeXT 下，它相对比较稳定，而在 OS X 下，每个发布版本都会更新 Objective-C。类似 Java 的异常处理和同步关键字已经加入有几年了，10.5 版本在兼容性方面又有大动作，并且引入了该语言的新版本：Objective-C 2.0。只有 10.5 和后续版本才支持 64 位 Cocoa 和 Objective-C 2.0，因而使用它们就意味着在使用全新的 Objective-C 运行时库。其中包含了很多为了让 Objective-C 未来更容易发展而进行的改变。

2.1 面向对象

Alan Kay 创造了面向对象这个术语，把它描述为：

简单的计算机们，通过消息传递来通信。

实现了原始 Smalltalk 系统的 Dan Ingalls，提出了如下的测试：

能否定义一种新的整数，把它放进矩形内（这些矩形已经是窗口系统的一部分），让矩形变黑，能实现这些么？

Smalltalk 是第一个纯粹的面向对象语言，Objective-C 从中获得了很多灵感。Objective-C 的对象模型和语法都直接来自 Smalltalk。有时它被亲切地称为“C 语言和 Smalltalk 的私生子”。

Smalltalk 的模型非常简单。对象是发送和接收信息的计算机的简单模型。如何处理这些信息全取决于它们自己，但是大多数时候用一个类（class）作为模板。对象带有一个指向类的指针，它把消息名到方法的映射委托给这个类来处理。这个类可能再把它委托给另外一个类。类之间的委托叫做继承（inheritance）。

如果类链都不知道如何实现一个指定的方法，会给对象发送一个 doesNotUnderstand: 消息，参数是一个封装了原始消息的对象。在 Cocoa 中等价的消息是 forwardInvocation: 消息：接受一个 NSInvocation 对象作为参数。这被叫做二次转发（second-chance dispatch）机制。类无法找到一个方法时，对象得到了处理方法的第二次机会。在 OS X 下，这比直接发送消息差不多要慢 600 倍，所以不应用于性能攸关的代码，但是在需要的时候确实可以极大地增加灵活性。

在 Smalltalk 中，任何东西都是对象，类也是对象。它们可以接收消息并作出回应。在 Smalltalk 和 Objective-C 中，对类的消息查找都由 metaclass（一个类的类）来处理。Smalltalk 在这种抽象上做得比 Objective-C 更多。Objective-C 中很少直接操作 metaclass。实际上，在 Objective-C 中 MetaClass 就是一个 Class 的 typedef，虽然 metaclass 的类体中会有一个标志标明。

理解在 Objective-C 或 Smalltalk 概念上的消息传递和在 C++ 或 Simula 概念上的方法调用之间的区别很重要。当调用一个方法时，使用对象的隐藏指针可以高效地调用一个函数。如果函数声明为 virtual，对每个类会得到不同的函数，但是这个映射是静态的。

发送消息的抽象程度更高。方法名（selector）到方法的映射是动态的。这不取决于调用者认为这个类是什么，只取决于这个类实际是什么，以及对象是否实现了二次转发机制。

OS X 上有一个 Smalltalk 的方言 F-Script 可以做脚本开发，Étoilé 也包含一个 Smalltalk 编译器，可以用来在 GNUstep 上开发 OpenStep 程序，不过它目前只支持 GNU Objective-C 运行时库，所以在 OS X 上不可用。

2.2 Objective-C

Objective-C 是 C 语言的纯超集，由 Brad Cox 和 Tom Love 于 1986 年设计。1988 年，他们把这个语言授权给了 NeXT 公司，之后很长一段时间，NeXT 公司都是这个语言的工具的唯一主要提供者。

自由软件基金会也常常把 Objective-C 作为 GPL 的成功案例。Steve Naroff 写了 GNU Complier Collection（GCC）下最初版本的 Objective-C 支持，但是 NeXT 公司并不想分发这个代码。他们试图发布一个二进制对象文件，要求用户把这个文件与他们的 GCC 链接，试图绕过 GPL 的要求。然而自由软件基金会最终迫使 NeXT 公司发布了源代码。

遗憾的是，故事并没有就此结束。编译器只是问题的一部分。Objective-C 还需要一个运行时库来处理模块载入、类查找，以及消息发送之类的事情。编译器其实只相当于一个 C 语言预处理器，把 Objective-C 的语法转换为对这个运行时库的调用。第一个 Objective-C 编译器和开源的 Portable Object Compiler（POC）都是这么干的。

NeXT 公司不发布自己的运行时库，所以自由软件基金会只能自己写一个。他们这样做了，而且还包含了一些小的改进，结果使得两者并不兼容。GCC 里的 Objective-C 支持代码，开始充满了协调 GNU 和 NeXT 运行库的 #ifdef 语句。接下来的 10 年，接口的差别越来越大，最终它们已经少有共享的代码了。NeXT 公司（以及后来的 Apple）从不把 GNU 运行时库代码合并到其 GCC 分支内，所以合并由 Apple 带来的改进也逐渐地困难起来。

2005 年，Apple 开始和 Low-Level Virtual Machine（LLVM）项目合作，并且雇佣了其中的一些主要开发者。他们用它来实现 OS X 10.5 中基于 CPU 的 OpenGL shader 实现，并开始使用它和 GCC 一起做代码生成。某种程度上这是因为 GCC 切换到了 GPLv3。2007 年，他们开始开发一个新的前端。像 LLVM 一样，这个叫做 Clang 的新前端也基于 BSD 许可。笔者编写了这个编译器下 Objective-C 代码生成部分的最初实现，它严格地区分了运行时部分和运行时无关部分。GNU 运行时后端只有少于一千行的代码，因而更容易维护、容易保持特性与 Apple 实现之间的同步。也就是说，长远来看，其他平台上的 Objective-C 支持会得到加强。

Objective-C 产生的年代 Smalltalk 虽然强大但太慢。运行 Smalltalk-80 环境所需的硬件非常昂贵。Objective-C 是一个折衷。它去掉了 Smalltalk 的一些特性，主要是闭包和垃圾回收，把剩下的特性加入 C。由于 Objective-C 有来自 C 语言的流程控制，于是闭包就没那么重要了。在 Smalltalk 中，闭包是唯一的流程控制方法；

如果想要个 if 语句，需要给值发一个 ifTrue: 消息，把一个闭包作为参数，这个值是 true 时闭包才会执行。Objective-C 丧失了这一灵活性，但是获得了大幅的速度提升。Apple 在 OS X 10.6 中加入了对 block 的支持，但是其灵活性不如 Smalltalk 的 block，而且不支持反射。

因为 Objective-C 是 C 语言的纯超集，所以每个 C 程序都是合法的 Objective-C 程序。虽然语言中定义了些新的类型，例如 id（任何对象）和 Class，但其都是在头文件中定义的 C 类型，和其他 C 类型有一样的作用域规则。相比而言 C++ 里 class 是一个关键字。一个有名为 class 的变量的 C 程序就不是合法的 C++ 程序。

除这些类型外，Objective-C 还包含少量新关键字。所有这些新关键字都以 @ 符号开头，这样就避免了和现有的 C 标识符产生冲突。

2.2.1 Objective-C 的可选编译器

原始的 Objective-C“编译器”其实就是一个把 Objective-C 代码转换为 C 语言代码的预处理器，然后使用 C 编译器来编译。NeXT 公司把它整合到 GCC 中，创造了一个真正的编译器。另外一个项目 POC，保持了原有的预处理器模型，但是因为进化的方向不同，所以它已经跟 GCC 版本不兼容了。

自从引入 OS X 10.6 以后，GCC 就被认为是一个陈旧的编译器。虽然 Apple 还会继续支持它一段时间，但它已不是新代码的最佳选择。因为授权的限制，GCC 主分支中对代码生成的改进不再合并到 Apple 的版本中。

OS X 10.6 中还包括了另外两个编译器，都基于 LLVM 构建。第一个叫 LLVM GCC。采用 Apple 分支的 GCC 前端代码，编译成 LLVM 的中间表达，然后由 LLVM 优化和转化为本地代码。

LLVM GCC 的主要目标是作为一个过渡步骤，使用 LLVM 进行代码生成、优化等，但是使用 GCC 的语法分析器。这意味着，它可以略有限制地编译任何 GCC 可以编译的代码，并在大多数情况下生成更好的代码。其中 LLVM 引入的一个好处是链接时优化（link-time optimization，LTO）。

一般来说，C 语言（也包括 Objective-C）都是一次编译一个预处理过的文件。每个分别编译的单元都是完全独立的，直到最后才被链接器合并到一起。LLVM 前端编译器输出 LLVM bitcode 文件，然后由一个支持 LLVM 的链接器合并到一起，优化，最后才转换为本地代码。这意味着优化过程可以更充分理解整个程序。一个小例子就是函数的内联。如果有一个短函数在一个文件中，而要在另外一个文件中调用，传统的 C 语言编译器是不会内联它的，而 LTO 则会。

LLVM 整合了大量可以从链接期间获益的优化手段。一个例子是函数特化，生成针对特定输入的特定函数版本。如果用一个常量作为参数调用函数，LLVM 会生成该函数的两个版本：一个版本接受任意参数；另一个版本会去掉那个参数，然后把该常数值放置在所有该参数被使用的位置上。用这个常数作为参数调用这个函数，实际上就会调那个新版本。某些情况下，它会让调用的函数变得很小（例如，可以去掉那些基于已经移除的参数的条件分支），从而允许其被内联。这样就把对一个复杂函数的调用转换成了几条内联后的指令，是一个很大的改进。

第三个编译器，Clang，使用和 LLVM GCC 一样的后端，但是它使用一个完全重写的语法解释器。如果使用命令行编译，会发现 clang 提供了比 GCC 更有帮助的错误信息[1]。它还提供了一个 C 语言、Objective-C 和 C++ 共用的语法解析器。笔者在写本书的时候，C++（同时 Objective-C++）支持还很不成熟，简单的 C++ 程序用 Clang 编译都会失败，预期可以在 2010 年晚些时候支持绝大多数 C++ 程序[2]。因为它使用和 LLVM GCC 一样的后端，所以可以用 Clang 编译 C 语言和 Objective-C，用 LLVM GCC 编译 C++ 和 Objective-C++（直到 Clang 成熟起来），而仍然得到 LLVM 提供的跨模块的优化所带来的好处。

Clang 采用了模块化的设计。Apple 使用 GCC 时遇到的一个问题是——从技术和法律上——很难把解析器和语义分析的部分抽出来，用来在 IDE 里进行语法检查和高亮，或者静态分析。Clang 的组件在 Apple 的工具里面应用广泛，而不仅用于编译。一个例子就是静态分析器，它可以在很多 C 语言和 Objective-C 程序中找到 bug。它在非 Apple 平台上也可用。

一般来说，可以用 Clang 就用 Clang 来编译你的代码，不行就用 LLVM GCC，还不行再用 GCC。Clang 的开发最活跃。GCC 的 Apple 分支改善不大。LLVM GCC 的前端和 Apple 的 GCC 同步，但 LLVM 的开发很活跃，所以代码生成部分会持续获得改进。可以说，GCC 是这三个当中最成熟的，而且仍旧是唯一支持某些晦涩的 GNU 扩展的，如 __builtin_apply() 系列，这使它成为了某些情况下的最佳选择。目前它是 Apple 工具中的默认选项，因为它的兼容性更好，但在未来可能会发生改变。

2.2.2 与 Java 及 C++ 的区别

很多人以为 Java 很接近 C++，那是不对的。Java 语法虽然像 C++，但是语

1 Xcode 3.2之后也支持解析clang的错误输出了，所以不必用命令行编译也能享受到更容易理解的信息。——译者注

2 在翻译本书的时候，Clang的C++/Objective-C++支持已经非常成熟了。——译者注

义上继承了很多 Objective-C 的东西。这倒不稀奇，因为很多研发 Java 的人来自 NeXT 公司，或在 Sun 参与了 OpenStep 项目。

Java 的设计目标是“普通程序员”的语言，包含了类似 Objective-C 的对象模型和类似 C++ 的语法。如果你用过 Java，就会觉得 Objective-C 很熟悉。它们的类型系统很相似，对象类别由接口和一些原始或者内建类型构成。Objective-C 类方法成为了 Java 的静态方法。Objective-C 协议成为了 Java 接口。Objective-C 的 category 在 Java 中没有出现。

Objective-C 和 C++ 的区别更为深切。Objective-C 是给 C 语言加入类似 Smalltalk 的扩展构成的。而 C++ 给 C 语言加入了类似 Simula 的扩展。虽然 Simula 常常被说成是第一个面向对象的语言，但是它没有通过 Ingall 测试。

C++ 类不是真正的对象。不能把它们当做对象，其消息分发机制也很不同。乍看之下，Objective-C 的消息分发机制和 C++ 的调用虚函数很相似。然而它们是完全不同的。C++ 支持多继承，为了支持它，实现了很复杂的类型转换规则。当把 Objective-C 中一个对象的指针转换成另外一个对象的指针时，编译器不生成任何代码。而在 C++ 中，指针转换的结果可以是一个不同的地址。当调用 C++ 的虚函数 doSomething() 时，就是在 vtable 里面作这种指针转换。

vtable 是 C++ 中最接近类对象的东西，虽然它不暴露在语言层面。vtable 其实就是一个函数指针的数组。每个虚函数在 vtable 中都有一个入口。在 C++ 中创建一个子类，就会得到一个新的 vtable，它包含了所有父类的虚函数。把指针转换成其中一个父类类型（隐式或显式地），就得到了一个包含父类 vtable 指针的结构体。调用方法则会调用指定位置下的函数。

这意味着即使两个对象实现了同名的虚函数，但如果其没有共有的父类声明该方法为虚函数，就不能用同样的途径调用它们。

在 Objective-C 中，方法查找机制仅依靠接收者的类和消息的名字。可以交换使用两个实现了相同方法的对象，而不用去管它们是否有公共的父类。这避免了大量多重继承的需求。

尽管语义方面的区别更重要，但语法上的区别更明显。C++ 和 Java 都借用了 C 的结构体语法用于方法调用。相比而言，Objective-C 采用了 Smalltalk 的语法使用方括号。下面的例子是把一个对象插入到 Java 的 Dictionary 对象和 Cocoa 的 NSMutableDictonary 对象：

```
// Java
aDict.put(a, b);
// Objective-C
[aDict setObject: b forKey: a];
```

Java 代码更短，但是可读性要差一些。不查看文档，或者对所涉及的 API 不够熟悉，就不知道 a 是 key，b 是 value。相比较而言，无须熟知这些就可以阅读 Objective-C 代码（只要懂英文）。

这就是 Cocoa 相比许多其他竞争框架更好的原因之一。因为这个语言会强制给每个参数命名，所以使用了大量 API 的代码就更容易阅读，无须对每个类和方法都熟悉。这也让学习 API 变得容易，如果记得方法名，就无须查阅文档去了解参数顺序。Java 当然也可以用 putKeyValue() 或类似的方法名，但它没有。而 Cocoa 总是会这样的。

2.2.3 Objective-C 2.0

在 OS X 10.5 中，Apple 发布了新版本的 Objective-C，命名为 Objective-C 2.0。随着 OS X 早期版本发行的 Objective-C 已经加入了许多新的特性，它们和老版本保持了二进制兼容。可以在任何新版本中使用一个 OS X 10.0 版本中的框架。然而 Objective-C 2.0 提供了一个崭新的开始。

在 OS X 10.5 中，Objective-C 包含了两个运行时库，Apple 把它们分别叫做“现代”和“遗留（legacy）”运行时库。64 位代码和为 iPhone 编译的代码使用现代库，32 位代码使用遗留库。OS X 10.5 之前的所有版本都使用遗留库，而且只支持 32 位 Cocoa 应用程序。

虽然现代运行时库经常被叫做 Objective-C 运行时库，但这不全对。在 Leopard 里，遗留库可以使用几乎所有的 Objective-C 2.0 特性。有些，例如快速遍历（fast enumeration），不需要任何运行时库的支持，但只能用于 OS X 10.5，这是因为它们需要一些仅在新版本 Foundation 框架中提供的对象的支持。

如果写的是 32 位代码，还是可以与早先 OS X 版本编译的框架一起使用 Objective-C 2，尽管这样就无法使用最大的新特性：垃圾收集。Objective-C 2 加入的垃圾收集意味着完全无须再操心内存管理问题。这是一个明显的进步，当然也带来了相应的开销。最明显的问题是可移植性，使用垃圾收集的程序只能运行在 OS X 10.5 和更新版本上，不支持 OS X 的早期版本及其他的平台。特别是，这包括了（当前版本的）iPhone。如果你考虑未来把代码移植到 iPhone 上去，那么就别用垃圾收集。

其他特性也很有用。属性（property）可用以定义一个访问对象中数据的标准接口，快速遍历可用以快速得到一个集合中的所有对象。把 Objective-C 2.0 叫做新版本有点误导，因为属性和快速遍历实际上是语法糖，而垃圾收集是可选的。

10.3 版本附带加入的结构化异常处理，其实才称得上真正重大的改变，但是它没有争得一个新的版本号。

2.3 Ruby与Python

Ruby 在 Cocoa 开发中的应用正在增长。Ruby 有着 Smalltalk 式的对象模型，以及 Perl 式的语法。这个对象模型意味着开发 Cocoa 程序很容易，虽然语法上的感觉并不相同。

OS X 上有两个版本的 Ruby。RubyCocoa 使用了标准的 Ruby 实现，并提供了对 Objective-C 的桥接，允许使用 Objective-C 对象。MacRuby 是一个新实现，它会编译 Rudy 对象，以使用和 Objective-C 一样的底层对象模型。

也可以通过 PyObjC 桥接使用 Python 进行 Cocoa 开发。这跟 RubyCocoa 类似，使用现有的 Python 实现，简单地提供一个访问 Cocoa 对象的桥接。这类 Objective-C 桥接实现最大的问题是，如 Smalltalk 一样，Objective-C 使用命名参数。最常见的做法是把 Objective-C 方法名中的冒号替换为下画线。Python 和 JavaScript 的桥接就是这么做的。

实现一个动态语言的桥接相对容易。可以用 NSProxy 实现一个简单的桥来在 Cocoa 端表示你的语言中的对象，使用你的语言中的 C 外部函数调用来调用 Objective-C 运行时函数，或者发送类似 performSelector: 这样的消息给 NSObject。Damien Pollet 和我只花了一个下午就用这个机制实现了一个简单的与 GNU Smalltalk 的桥接。

2.4 小结

在 OS X 上编程还有很多的语言可以选择，但是 Cocoa 框架是为 Objective-C 设计的。如果想得到 Cocoa 全部的能力和灵活性，还是应该使用 Objective-C。

很多语言支持桥接或其他接口，但是提供的能力不同。如果你熟悉其中的一种，或者必须要整合用它们写的代码，那么你应该用它们。

本书的其他部分都用 Objective-C 举例，但任何使用 Cocoa 桥接的语言，都可以发送一样的消息，可以实现大部分的例子。如果你想用别的语言开发 Cocoa 程序，那么用你喜欢的语言来改写本书中的例子是一个不错的开始。