概述

本部分包含如下内容：

面向对象的开发方法使得程序开发更加直观、快捷，程序更容易被重构、理解。大多数面向对象开发环境至少由以下三个部分组成：

- 对象库
- 开发工具集
- 支持面向对象的编程语言和相应的类库

Objective-C 是一种设计用来支持面向对象开发的简洁的计算机语言，它是标准 C 语言的一个很小但是很强大的超集。除了 C 之外，Objective-C 主要基于 Smalltalk，最早的面向对象的编程语言之一。Objective-C 以一种简单和直接的方式使得 C 语言具有了面向对象的能力。

如果您以前从来没有使用过面向对象的方法开发程序，本文档则能够帮助您熟悉面向对象的开发技术。它阐述了面向对象设计的意义，以及如何开发面向对象的程序.

谁应该阅读本文档

本文档的阅读对象为对如下几个方面感兴趣的读者：

- 面向对象编程
- Cocoa 应用程序框架的基础
- Objective-C 编程

本文档介绍了 Objective-C 所基于的面向对象模型。

本文档不是 C 语言的介绍文档，所以假设您已具备 C 语言的相关知识。然而，您无需为您不是一个熟练的 C 程序员而担心，因为 Objective-C 的面向对象编程和基于过程的标准 C 的编程有很大不同。

重要： 本文档只是描述了对于使用 Objective-C 编程来说很重要的一些基础概念，并没有对 Objective-C 语言本身做过多的叙述，如果您对 Objective-C 语言感兴趣，请参考 *Objective-C 2.0 程序设计语言*。

本文档的结构

本文档分为如下几个章节：

- “为何是Objective-C？”解释了为什么选择Objective-C作为Cocoa框架的开发语言。
- “面向对象编程”讨论了面向对象编程的基本原理，并阐述了面向对象技术背后的思想，介绍了大量的专门术语。即使您已经对面向对象编程非常熟悉，也推荐您阅读一下该章节，从而对面向对象的Objective-C和使用的术语有一个感性认识。
- “对象模型”

- “程序的组织结构”
- “结构化编程”

参考

Objective-C 2.0 程序设计语言 介绍了Objective-C编程语言。

Objective-C 2.0 运行时系统编程指南 描述了怎样和Objective-C运行时系统交互。

Objective-C 2.0 运行时系统参考库 介绍了 Objective-C 运行时系统库的数据结构和函数。您的程序可以使用这些接口来和 Objective-C 运行时系统进行交互。例如，你可以增加类或者方法，或者获取所有已经加载的类的类定义表单。

为何是Objective-C?

Cocoa 框架选择了 Objective-C 作为开发语言有许多方面的原因。首先，也是最主要的原因，它是一个面向对象 的语言。Cocoa 框架中的很多功能只能通过面向对象的技术来呈现，本文档将对 Cocoa 框架的功能进行具体阐述并介绍怎样使用它们。其次，是标准 C 语言的一个超集，现存的 C 程序无需重新开发就能够使用 Cocoa 软件框架，并且您可以在 Objective-C 中使用 C 的所有特性。您可以选择什么时候采用面向对象的编程方式（例如定义一个新的类），什么时候使用传统的面向过程的编程方式（定义数据结构和函数而不是类）。

此外，Objective-C 是一个简洁的语言，它的语法简单，没有歧义，并且易于学习。 因为易于混淆的术语以及抽象设计的重要性，对于初学者来说，面向对象编程的学习曲线比较陡峭。象 Objective-C 这种结构良好的语言使得成为一个熟练的面向对象程序员更为容易。 介绍 Objetive-C 的章节也如同其语言本身一样简洁。

和其他的基于标准 C 语言的面向对象语言相比，Objective-C 对动态机制支持得更为彻底。编译器为运行环境保留了很多对象本身的数据信息，因此某些在编译时需要做出的选择就可以推迟到运行时来决定。这种特性使得基于 Objective-C 的程序非常灵活和强大。例如，Objective-C 的动态机制提供了两个一般面向对象语言很难提供的优点：

- Objective-C 支持开放式的动态绑定，从而有助于交互式用户接口架构的简单化。例如，在 Objective-C 中发送消息既无需考虑消息接收者的类也不用考虑方法的名字，从而可以允许用户在运行时再做出决定，也给了开发者在设计时极大的自由（术语“动态绑定”，“消息“，“消息接收者”， “类”将在随后的章节中进行介绍）。
- Objective-C 的动态机制成就了各种复杂的开发工具。运行环境提供了访问运行中程序数据的接口，所以使得开发工具监控 Objective-C 程序成为可能。

注意：作为编程语言来说，Objective-C 有很悠久的历史。它在二十世纪八十年代早期被发明于 Stepstone 公司，作者是 Brad Cox 和 Tom Love。八十年代后期，NeXT 计算机有限责任公司获得了使用 Objective-C 来开发 NeXTStep 框架的授权，也就是后来的 Cocoa。NeXT 在多方面对 Objective-C 进行了扩展，例如协议部分。

面向对象编程技术

在现实生活中，我们必须弄明白我们所面对的大量的事实以及观念。为此，我们需要从表面细节中抽象出其内在逻辑，发现事物的本质。抽象法可以帮助我们揭示事物的因果，结构和表现形式，区分主要和次要。

面向对象编程提供了一种对您所操作的数据进行抽象的方法—而且，面向对象编程将数据和对数据的操作组合到一起，从而使数据具有了行为。

本部分包含如下内容：

数据和操作

传统的编程语言通常划分成两个部分—数据和对数据的操作。数据是静态的，不变的，除非通过操作来改变它。对数据操作的函数并不保留上一次操作时数据的状态，它们的作用仅体现在操作数据上。

很明显，这种划分是基于计算机的工作方式，所以您很难忽视它。和无处不在的原料和能量以及名词和动词的划分一样，它构成了程序的本质。从某种意义上说，所有的程序员—即使是面向对象的程序员—都是工作在数据结构之上，他们的程序也会使用和定义函数来操作数据。

对于面向过程编程语言例如 C 来说，这几乎就是一切了。语言本身可能为组织数据和函数提供了多种支持，但是本质上仍然是分成数据和操作两个部分。函数和数据结构是设计的基本元素。

面向对象编程当然不会这样来划分，而是在更高的层次重新组织。它把操作和数据组合为一个模块单元叫做**对象** ，并且把对象组成一个结构化的网络来完成一个程序。在面向对象的编程语言中，对象和对象间的交互才是设计的基本元素。

每个对象都具有状态（数据）和行为（对数据的操作）。也就是说，它们和物理实体并没有太大区别。一台机械装置例如怀表、钢琴如何体现自己的状态和行为是显而易见的，然而几乎每样功能性的物体都能够体现自己的状态和行为。即使一个普通的瓶子也会有状态（饼子有多满，瓶子是否已经打开，瓶子所装的液体温度多少）和行为（以不同的流量倒出所装液体，盖上或者打开，加热或者制冷）。

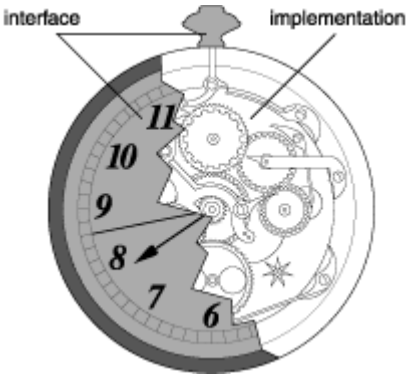
正是这种与实物的相似性赋予了对象强大的能力和吸引力。这些对象不仅可以对现实系统的组件建模，也能够胜任软件系统中的同样的角色。

接口和实现

编写一个程序，您需要对问题进行抽象， 并使用程序将其表达出来。这正是编程语言帮您做的工作。编程语言可以帮助您对抽象后的问题进行编码从而使得程序的编写和设计过程更为简单。它使您只用关注您所写的代码和程序的架构而无需关心表面的细节。

所有的程序语言都提供了对抽象后的问题进行编码的方法。本质上，这些方法就是怎样分离并隐藏实现细节，然后至少在某种程度上，提供给这些细节一个公共的接口—就像机械装置将它的表面和内部的构造分离一样—如图"接口和实现"所示。

图 2-1 接口和实现



随着对该装置的观察角度不同，您所关注的层面也不同。作为一个实现者，您需要关心它是由什么构成并且怎样工作的。作为一个用户，您仅需要关心它是什么并且它能够做什么。您可以略过那些具体的细节而从更高层次来考虑问题。

在 C 语言中，程序的基本单位是结构体和函数，这两者从不同的方面隐藏了具体实现细节。

- 从数据方面，C 中的结构体把数据组合成一个更大的单位从而可以象一个单独的实体一样对它进行操作。尽管有时候代码需要访问结构体内部的数据，但程序一般都将结构体作为一个单独的数据实体来对待—而不是数据的集合。结构体能包含另一个结构体，从而组成复杂的数据结构。

  在现代 C 语言中，结构体中的数据元素位于该结构体的名字空间—也就是说，结构体内的元素名字不会和结构体外的同名元素冲突。 划分名字空间对保证实现细节和接口的分离来说是非常必要的。例如，试想一下，在一个大规模的程序中给每个数据元素赋予一个不同的名字并且保证新的命名不会和已存在的命名冲突是一件多么巨大的工作。

- 从过程方面，函数封装了一些可以重复使用而不用重新实现的行为（操作）。和结构体的数据元素一样，函数中局部变量也处于该函数自己的名字空间。函数可以调用其他的函数，从而组成复杂的逻辑。

  函数是可复用的。一旦函数被定义，则可以不限次的调用而无需再重新实现。一些被广泛使用的函数可能被收集在函数库中并且在不同的应用程序中被重用。用户只需要函数的接口，而不需要知道函数的实现。

  然而，和结构体中数据元素不同的是，函数并没有划分不同的名字空间。每个函数的名字必须唯一。尽管函数是可复用的，但函数名不是。

C 语言中的结构体和函数都能够帮助您对问题进行抽象，但是它们仍然区分了数据和对数据的操作。在面向过程的编程语言中，问题要么被抽象成数据，要么被抽象成操作。您的程序总是按照计算机的工作方式去设计。

面向对象的语言没有抛弃结构体和函数的优点—它更深入了一步，将问题抽象成更高层次的单位，从而隐藏了函数和数据之间的交互。

例如，假设您有一组函数（接口）作用于某个特定的数据结构。您希望这些数据结构和接口无关从而使得这些函数尽可能的易于使用，所以提供了一些额外的函数来管理这些数据。所有对该数据结构的操作—分配内存空间，初始化，获取数据，更改数据，更新数据，释放内存空间—都通过这些额外的函数来进行。用户所需要做的只是以该数据结构为参数调用这些函数。

这样，该数据结构对其他的程序员来说是不透明的，他们无需关心该数据结构的内部构成，从而专注于函数的功能，而不是该数据结构。到现在为止，您已经完成了创建对象的第一步。

下一步就是在编程语言中支持这一想法并且完全隐藏掉数据结构从而无需在函数中作为参数来回传递。该数据结构成为一个内部的实现细节，暴露给用户的是函数接口。因为对象完全封装（隐藏）了内部数据，所以用户只用考虑对象的行为。

有了这一步，这些函数的接口变得更为简单。调用者无需知道函数是怎么实现的（使用了什么数据）。现在我们可以把它称之为"对象"。

被隐藏的数据结构把所有的访问自己的函数联合在一起。所以，一个对象不是一些随机选择的函数的集合，而是数据结构支持的一组相关的行为。要使用一个对象的函数，您必须首先创建一个对象（从而分配了内部数据结构），然后告诉这个对象执行哪个函数。从这时开始，您只需考虑这个对象能做什么，而不用单独的考虑这些函数能做什么。

从函数和数据结构到对象行为的过程就是学习面向对象编程的精髓。可能一开始有点不习惯，但如果您有了一定的面向对象编程经验后，您会发现面向对象更符合我们思考问题的自然方式。日常的编程术语中充满了对现实世界的模拟—表单，容器，表格，控制器，甚至管理者，将这些做为程序对象来实现也是自然而然的事情。

不同的编程语言对问题抽象的方式也不同。您不应关心和当前问题无关的细节。

例如，如果您总是要求相应的过程必须处理相应的数据，则您将需要知道底层的实现细节。尽管您仍然可以在更高的层次抽象问题，然而，从设计到实现的区别可能变得很细微—而且程序规模越大，越复杂，则设计越困难。

通过对问题不同层次的抽象，面向对象语言向您提供了更大的词汇表和更丰富的编程模型。

对象模型

面向对象编程就是在比较高的层次上把状态和行为——数据和对数据的操作——组合到**对象**中，并且提供了编程语言上的支持。对象就是一组相关的函数和为这些函数服务的数据的集合。这些函数被称为对象的**方法**， 数据被称为对象的**实例变量**。对象方法封装了对实例变量的访问，实例变量在对象之外是不可见的，如图 3-1 所示：

图 3-1 对象

如果您曾经应付过一些比较棘手的编程问题，您的程序中可能已经有类似的设计——一组相关的函数作用于特定的数据，这可以称之为没有编程语言支持的隐形"对象"。面向对象使这些函数显式的联系在一起，在程序中作为一个单独的实体来对待。访问一个对象的数据的唯一途径就是通过该对象的方法。

通过把状态和行为组合在一块，对象所体现的作用超过了它的各部分之和。一个对象就是一个在特定的功能领域自给自足的"子程序"，在大型的程序设计中可以充当独当一面的角色。

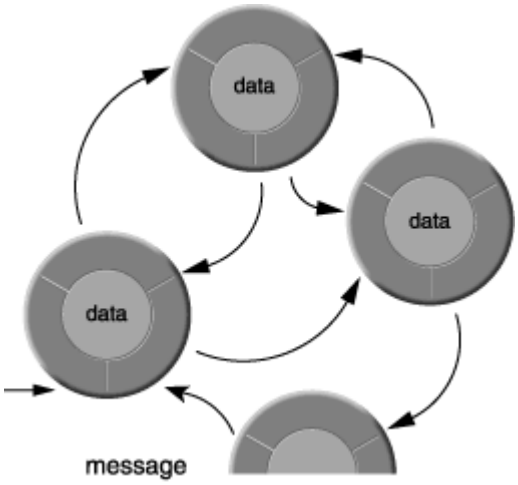
术语： 面向对象的术语随语言的不同而不同。例如，在 C++中，方法被称之为"成员函数"，实例变量被称之为"数据成员"。本文档使用的术语源自 Objective-C 和 Smalltalk。

例如，如果您试图对家庭用水进行编程建模，您可能需要创建各种对象来表示出水系统的各个组件。水龙头对象将会有如下方法：开始放水，停止放水，设置放水速率，返回一定时间内所有流出水的数量等等。相应的，水龙头对象将需要一些实例变量来记录水龙头开关的是打开的还是关闭的，已经使用了的水的数量，以及水的来源。

很明显，程序中的水龙头对象比实际中的水龙头功能要多一些（类似于附加了很多测量仪器的水龙头）。然而和其他的系统组件一样，即使是一个真实的水龙头，它也同时具有状态和行为。您需要使用对象来更有效地对系统建模。

在面向对象编程中，程序是由相互关联的对象构成的对象网络，这些对象可以彼此调用来解决问题。在整个程序设计中，每个对象都有自己特定的角色，对象之间可以通过**消息**来相互通信。

图 3-2 对象网络



对象网络中的对象不会只有一种。例如，对家庭用水进行建模的程序中，除了水龙头对象之外，可能还会有输水的管道对象，控制管道中水流的阀门对象。同时还会有建筑对象，建筑对象中包含很多管道对象，阀门对象，水龙头对象以及各种可能关闭和打开阀门的装置对象—如洗碗机，马桶，洗衣机，还有使用这些装置及水龙头的用户。当建筑对象需要知道用水数量时，它将会通知每个水龙头和阀门对象报告当前的状态。当用户启动某样装置时，该装置则需要打开阀门来获取所需要的水。

本部分包含如下内容：

消息隐喻

每种编程模式都会有自己的术语表和隐喻。面向对象编程也不例外。隐喻能够让您从更专业的角度来理解程序。

例如，有种趋势是将对象看做一个个的角色，并赋予它们类似于人的能力。试想一下，一个对象将”决定“在某种情况做什么事情，”询问“其他的对象来获得信息，”自我检查“来获得需要的信息，”代理“其他的对象，”管理“一个进程，这听起来似乎很有趣。

这些隐喻让您认为对象是在主动"使用“它们的方法，而不是在面向过程的编程中一样，是您调用了函数或方法来完成工作。对象不只是一个状态和行为的容器，而被认为是程序活动的代理。

这些隐喻很有用处。对象在许多方面表现得都很象人类：对象是自包含的，在整个程序设计中具有一个特定的角色，并且它可以独立的和程序的其他角色进行交互，在某种程度上可以自己决定一些行为。当然，就象在舞台上的演员一样，它不能脱离剧本，但是对象角色本身具有多面性和复杂性。

把对象看做角色很好地呼应了面向对象编程的最主要的隐喻—对象通过“消息”来通讯。您需要给一个对象发送消息来执行它的方法而不是直接调用它的函数。

尽管这需要一些时间来习惯，但是这些隐喻有助于我们理解对象和对象的方法。它把方法从操作的数据中分离出来而仅关注方法的行为。例如，在面向对象编程接口中，start 方法会初始化一个操作，而 archive 方式则会将信息归档，draw 方法则会画出一副图片。方法的名字并没有指出具体哪个操作被初始化，哪些信息被归档，哪副图片被画出。不同的对象可能以不同方式来执行这些方法。

因此，方法是对行为的抽象。您必须把方法关联到一个对象来调用这些方法。这些对象被称作消息的“接收者”。被选作接收者的对象将决定具体被初始化的操作，被归档的数据以及画出的图像。

因为方法可以认为是隶属于某个对象的，所以特定的方法只能选择特定的接收者（实现了该方法的对象并且该对象具有该方法操作的数据结构）。不同的接收者对同一方法可能有不同的实现，因此会做不同操作。仅从消息本身以及方法名字不能得出消息的结果，它还依赖于接收消息的对象。

通过把消息（请求的行为）和接收者（能够响应请求的方法的拥有者）分离，消息隐喻完美地把行为和针对行为的实现分离开来。

类

同一程序中可以有同一类型的多个对象。例如，在用水模型的程序中可能有多个水龙头对象，管道对象，装置对象以及用户对象。 同一类型的对象我们认为是同一个**类**的实例。同一个类的实例具有同样的方法，同样的实例变量（实例变量的值可能不同），并且共享同一份类的定义。

从这一点上，类和 C 中的结构体类似，都是定义了一种类型。例如，下面的声明

```
struct key {

    char *word;

    int count;

};
```

定义了 struct key 这种类型。一旦被定义，就可以创建 struct key 的实例：

```
struct key  a, b, c, d;

struct key *p = malloc(sizeof(struct key) * MAXITEMS);
```

上面的第一段代码并没有创建程序能实际使用的变量，而是创建了结构体类型的一个模板。第二段代码才是创建该类型的实例并为它分配内存。

类似的，类的定义只是创建了某种类型的对象的一个模板，这个模板可以被用来创建多个同一类型的对象—类的**实例** 。例如，程序中只会有一份水龙头类的定义。通过这个定义，程序能够创建很多它需要的水龙头对象并为之分配内存。

和结构体的定义一样，类的定义也包括数据元素（实例变量）。每个该类的对象实例都会包含这些数据元素，并为它们分配内存。这些数据元素中保存着各个对象实例独有的数据。

然而，和结构体定义不同的是，类定义同时也包括描述类行为的方法。对象的所属的类随着该对象拥有的方法的不同而不同。两个对象拥有同样的数据元素但是不同的方法则不会属于同一个类。

本节包含如下内容：

模块化

对于 C 程序员来说，模块往往意味着源代码的一个文件。把一个大型的程序（甚至不用很大）的代码拆分到不同的文件中是一种很方便的管理方式。每个文件都可以单独的编译，然后链接在一起就是最后的程序。使用 static 标识符能够使符号的作用域仅限于该符号所在的文件从而减小模块间的耦合性。

然而，这种模块的划分是由文件系统决定的。它更象是源代码的容器而不是语言上的逻辑单位。容器内部的实现对于每个程序员都是可见的。您可以将逻辑上相关的代码组织在一个文件中，但您也可以不这么做，完全没有限制。文件就象一个橱柜的抽屉，您可以一个抽屉放袜子，另一个放内衣，等等，或者您也可以以另外一种分类方式来存放，或者简单的全都放在一起。

对象的方法：和实例变量一样，我们很容易认为方法是对象的一部分。如图 3-1所示，方法围绕在对象实例变量的周围。然而，在内存中并不是如此。每个对象的实例变量都会占据一块内存，但无论创建了多少个该类的对象实例，同一个类的对象的方法在内存中只会存在一份，。
面向对象编程语言既支持利用文件来划分模块，也支持通过类从逻辑上划分模块。通常，这两种模块划分方法是一致的，因为每个类都被定义在不同的文件中。

例如在 Objective-C 中，管道类的定义源文件中可能包含了阀门类和管道类交互的部分，这儿的管道类是按文件划分的模块，而阀门类的定义被划分到好几个文件中，但在整个程序结构中，阀门类仍是一个模块单元——按逻辑划分的模块——无论它被定义到多少个文件中。

关于类定义和逻辑模块的之间的更多细节将在 “抽象机制”一节中进行讨论。

复用性

面向对象编程的一个主要目的就是让您的代码尽可能的被重用——在不同的情景下和不同的应用程序中——从而避免重复实现。代码的可复用性与很多因素有关，包括：

- 代码的可靠性和 bug 的多少
- 文档的清晰程度
- 程序接口是否简单直接

- 代码的性能如何
- 代码的功能如何

显然，这些因素不仅仅作用在对象模型中，它们可以用来衡量任何代码的复用性—无论是标准的 C 的函数还是类的定义。例如，高效的和有着良好的文档的函数的可复用性要大于那些不可靠的和没有文档的函数。

虽然如此，我们仍然可以作一个比较来说明类的可复用性要好于函数。虽然有多种方式可以增强函数的可复用性—例如将数据作为参数传递给函数而不是使用全局变量，然而，仍然只有一小部分函数能被其它的应用程序复用。函数的可复用性的局限主要体现在三个方面：

- 函数名是全局的；每个函数必须有一个唯一的名字（除了被声明为 static 的函数）。这使得开发一个复杂的系统时很难依赖那些函数库。编程接口因此变得难于学习和异常庞大。

  另一方面，类可以很容易的共享编程接口。不同的类可以有同样的方法名，这一命名机制使得许多功能性的实现可以共用一个很小的，易于理解的接口。

- 函数只能挨个从函数库中选择，程序员需要负责从函数库中选取自己需要的函数。

  与之相反，类是作为一组功能整体出现的而不是单独的函数和实例变量。类提供的是集成的服务，所以面向对象的程序员不用为集成库和自己的实现方案而烦恼。

- 函数通常和特定的数据结构绑定在一起。数据和函数的互操作不可避免的使得数据接口也成为接口的一部分，因此函数仅仅对那些使用和函数参数同样数据结构的用户有用。

  类的数据都已得到很好的封装，所以不会有同样的问题。这也是类的可复用性比函数更好的一个最主要的原因。

对象的数据对于程序的其它部分来说是不可见的，因而对象的方法不用进行数据完整性的检查。数据不可能会有逻辑错误或者不合理的状态。因此，对象的数据远比传递给函数的参数可靠，从而可复用的对象方法也比函数更容易设计一些。

此外，由于类的数据是封装在类的实现中的，所以类可以重新实现它的数据结构而不会影响类的接口。使用该类的程序不用修改任何代码就可以使用该类的更新版本。

继承

解释一个新的东西最好的方法是从旧的东西开始。如果您的听众已经知道什么是“帆船”对您解释什么是“斯库纳纵帆船”非常有帮助。如果您想解释一下拨弦键琴是怎么工作的，最好的情况是您的听众已经知道钢琴的内部构造，或者看见过吉他是怎么演奏的，至少，熟悉某种乐器。

定义一个新类也一样，如果能够从现存的类开始，一切都会更加简单。

为此，面向对象的编程语言允许您基于一个现存的类定义一个新类。现存的类被称作**父类**；新类被称作**子类**。子类只需要定义和父类不同的部分。

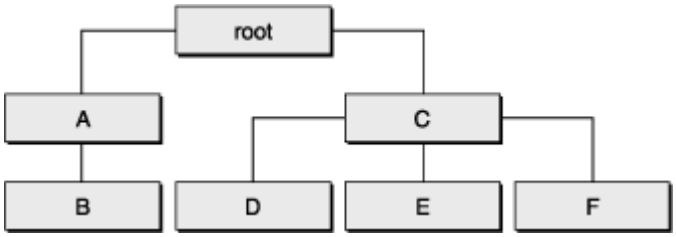
子类并非从父类拷贝而来，而是**继承**了父类的所有方法和实例变量。如果子类的定义是空的（没有定义任何自己的变量和方法），这两个类将是一样的（除了类名）并且共享同一份定义。这就和"fiddle"和"violin"都是表示小提琴一样。然而，定义新类的目的并不是生产同义词，而是因为子类至少有些东西和它的父类不同。

本节将包括如下内容：

类的继承体系

任何类都可以作为一个新类的父类，也可以是一个类的子类同时是另一个类的父类。在继承体系中类的数目并没有限制，如图 3-3 所示：

图 3-3 继承体系



每个继承体系都是从一个没有父类的根类开始向下衍生的。每一个类都继承自它的父类，并通过父类，继承了父类的父类，以此类推，每个类都继承了根类。

每个类都是它的继承链上所有类的定义的累积。如上所示，D 继承自父类 C 和根类，因此 D 拥有三个类 D,C 和 root 定义的所有方法和实例变量。

通常，一个类只有一个父类和任意数目的子类。然而在某些面向对象的编程语言（尽管不是 Objective-C）中，一个类可以继承自多个父类。和图 3-3 中继承体系只能向下衍生不同，多重继承可以有几个继承链最后合并为一个，这几个继承链可以来自不同继承体系（继承自不同的根类）。

子类的定义

子类通常从三个方面对父类的定义做出改动：

- 子类可能通过增加新的方法和实例变量来扩展父类的定义。通常这是需要定义子类的一个最主要的原因。子类总是有一些新的方法，或者新的实例变量。
- 子类通过替换继承自父类的方法来更改该方法的行为。通常，子类简单的实现一个和父类同名的新方法，从而*覆盖*了父类的方法（父类的方法并没有消失，其它的类从父类继承的时候仍可以使用该方法）。
- 子类可能扩展父类的方法但仍然保留着父类方法的行为。子类可以用一个新的方法覆盖父类的方法，然后在新方法中先调用父类的方法，然后再进行子类自己的处理。继承链中的每个类都会构成方法行为的一部分。例如在图 3-3中，D中的一个方法行为可能既包含C的处理行为，同时也包含了根类的处理行为。

通过以上的改动，子类相对父类而言变得更具体化和特殊化。子类一般会增加或者替换父类代码实现而不是减少代码。注意，通常方法不能被禁止继承，并且不能移除或者覆盖实例变量。

继承的用法

关于继承体系比较经典的例子就是动植物分类。例如，松科的子类可能有冷杉，云杉类，松树，铁杉，美洲落叶松，花旗松。松树的子类有软松，硬松。软松的子类又有白松，兰伯多松，硬松的子类有西黄松，斑克松，辐射松，红松。

在实际中很少有程序如此分类，但是用来作例子分析类的继承很合适。 子类一般都是对父类做一些定制，或者对父类进行某方面的扩展，以达到一些特别的目的，而不是为继承而继承。

下面是继承几种典型的用法：

- 代码复用。如果两个类有部分相同的代码，那么相同的部分就可以放在一个独立的类中，让其它的类来继承它，从而相同部分的代码只用实现一次。

  例如，水龙头，阀门和管道类，都需要连接到水源并且记录水流的速度。这些共同部分可以在一个公共类中实现，水龙头，阀门，和管道类分别继承这个类。水龙头从某种意义上说也是一种阀门，所以水龙头类可以继承自阀门类，然后加上水龙头的属性。

- 创建协议类。一个类可以声明一些方法并要求子类实现这些方法。该类可能声明了一些空的方法，或者只有初步实现的方法。无论那种情况，我们都可以认为，该类为它的子类建立了一种**协议**。

  当不同的类可能会实现同名的方法时，程序最好采用多态的设计。构造一个协议类来让子类继承是一种很好的习惯做法。

- 通用性功能的发布。程序员可能定义了一些基本的，通用的，能解决某方面问题的类，但是这个类没有处理各种细节。其它的程序员可以创建一个继承这些类的子类来满足特定的需求。例如，用水系统建模程序中的装置类，程序可能会定义一个通用的用水设备类，然后创建继承自该类的装置子类，在子类中返回具体的设备类型。

  继承能够帮助程序员减少工作量，并且有助于分层实现。

- 需要对现有逻辑作一些轻微的改动。在上述的用法中，都是设计一个类来让其它的类继承。但是，您也可以继承一个不是设计用来作父类的类。例如，有个类在您的程序中工作得很好，但是您想修改其中得某些函数，您就可以继承这个类，并在子类中做出修改。

- 功能预览。子类可以帮助我们进行选择测试。例如，如果一个类设计了一个特定的接口，则在子类中可以设计另外一套接口。每个接口都向用户开放，由用户选择喜欢的接口，并把它合并到父类中去。

动态机制

在编程历史中的某一个时期，一个程序会使用多少内存是在源代码编译和链接的时候就决定了的，既不会增加，也不会缩小。

现在看来，很明显，这是一个严重的局限因素。它不仅仅限制了程序的结构，程序的功能，还限制了程序的设计和编程技术的进步。动态分配内存的函数（例如 malloc）的出现使这些限制不再存在。

之所有成为限制因素是因为程序只能根据编译期和链接期从程序员的源代码中获得的信息做出决定，而不是根据程序的运行环境。

尽管动态分配机制消除了静态内存分配这种限制，但是许多类似的限制仍然存在。例如，编译时期类型匹配，程序的边界地址在链接时期就必须确定。应用程序的每个部分必须组织成一个单独的可执行文件。程序运行时不能加入新的模块和新的类型。

Objective-C 基本上克服了这些限制从而使得程序尽可能的灵活。Objective-C 将很多编译期和链接时需要作的决定挪到了运行时来作，目的就是让程序根据用户运行程序的环境来决定应该发生什么事情而不是根据语言和编译器，链接器的需要。

在面向对象编程中，有三种类型的动态机制：

- 动态类型识别，直到运行时才决定一个对象的类别。
- 动态绑定，直到运行才决定调用哪个方法。
- 动态加载，运行时加入新的模块。

本节包含如下内容：

动态类型识别

通常，如果您在不能转换的类型之间互相赋值，编译器可能会给出这样的警告：

```
incompatible types in assignment

assignment of integer from pointer lacks a cast
```

类型检查有时候是有用的，但是它经常和您从多态中获得的好处冲突，尤其是每个对象的类型必须在编译时确定。

例如，假设您现在需要向一个对象发送 start 消息。和其它数据元素一样，对象也是一个变量。假如变量的类型必须在编译期确定，这就不可能让运行时的因素决定什么类型的对象将会赋给该变量。如果代码中变量的类型是确定，start 消息所调用的方法也就确定了。

从另一方面说，如果变量的类型能在运行时决定，那么任何类型的对象都可以赋给该变量，从而 start 消息所调用的方法也随着变量在运行时类型的不同而不同，并产生不同的结果。

动态类型识别不仅仅是动态绑定（随后将进行介绍）的基础。动态类型识别容许对象之间的关系到运行期才决定，而不用在设计时确定。例如，一个消息可能在参数中传递一个对象而不指明它的类型，然后消息接收者发送消息给该对象，甚至不用考虑对象的类型。因为消息接收者利用传递过来的对象来完成一部分工作，所以在某种意义上，消息接收者的行为是可以由传递的对象的类型来决定的。

动态绑定

在标准 C 语言中，您可以声明一些功能相近的函数，例如标准的字符串比较函数：

```
int strcmp(const char *, const char *);   /* case sensitive */
```

```
int strcasecmp(const char *, const char *); /*case insensitive*/
```

并声明一个具有相同参数和返回类型的函数指针如下：

```
int (* compare)(const char *, const char *);
```

然后，您可以等到运行时再决定将哪个函数赋给这个指针，

```
if ( **argv == 'i' )
    compare = strcasecmp;
else
    compare = strcmp;
```

并通过函数指针调用字符串比较函数：

```
if ( compare(s1, s2) )
    ...
```

以上和面向对象编程的**动态绑定**很相似，都是到程序运行时才决定调用的方法。

尽管不是所有的面向对象语言都支持动态绑定，但是动态绑定通常可以通过消息来实现。您不需要象上面一样声明一个函数指针然后给它赋值，也不用给每个方法都取个不同的名字。

消息并不是直接地调用方法的。每个消息都会关联到一个方法调用。消息机制将会检查消息接收者的类型，并且在消息接收者的类的实现中查找对应消息的方法实现。当这一切在运行时完成时，方法和消息就是动态绑定的。如果是在编译期完成，就是静态绑定。

后期绑定：虽然有些面向对象的编程语言（尤其是 C++）需要在源代码中静态地指定消息接收者的类型，但是消息接收者可以是其指定类型的对象，也可以其子类的对象。因此，编译器并不能确定消息所调用的方法。这种情况下，有两种选择，一种是简单地将消息和方法按照指定的消息接收者的类型绑定，一种是晚些时候解决这个问题。C++中的选择是将这些方法（成员函数）声明为**虚拟函数**。一般称这为“后期绑定”而不是“动态绑定”。和动态绑定发生在运行时不同，后期绑定有严格的类型限制。正如现在所讨论的（以及 objective-C 中所实现的），动态绑定没有限制。
即使没有动态类型识别，动态绑定也是可行的，但是这没有什么意义。如果消息接收者的类能被编译器所确定，则动态绑定没有任何好处，因为编译器链接的方法和运行时绑定的没有任何区别。

然而，如果消息接收者的类型是动态的，则编译器不可能在编译的时候就能链接到正确的方法，只有在运行时，消息接收者的类型已知的情况下，才可能确定应该调用的方法。因此，动态类型识别催生了动态绑定。

动态类型识别使得消息的结果随消息接收者类型的不同而不同成为可能。运行时的因素从而能影响消息接收者的选择和消息的输出。

动态类型识别和动态绑定使得您在代码里可以发送消息一个还未确定的对象。如果该对象的类型可以到运行时再确定，其它的程序员就可以自由的设计自己类和数据类型，并且响应您所发送的消息，只要你们对消息协议而不是数据类型达成一致。

注意： 动态绑定在 Objective-C 随处可见，您无须刻意安排，您的代码也无需为此作特别的改动。

动态加载

历史上，在大部分运行环境中，程序必须被链接成一个执行文件，程序运行时，执行文件会一次性载入到内存中。

面向对象的编程环境允许一个可执行的程序分成几个文件。程序可以在运行后按照需要动态的加载和链接不同的文件。用户的操作将决定哪个文件会被载入到内存中。

一般来说，只有程序的必要部分才会在程序被启动时加载，其它的模块则随用户的请求而加载。没有被加载的模块并不占用系统内存。

动态加载有一些非常有意思的好处。例如，整个程序不需要一次开发完成。您可以分批的发布您的软件，并且每次更新一部分。您可以在程序中提供很多类似的工具—用户可以选择需要的工具并且只有该工具被会加载。

目前来说，动态加载最重要的好处也许是一个程序的可扩展性。您可以允许其它的人对您的程序定制。您需要作的就是在您的程序中提供一个框架让其它的程序员来实现，并且在运行时发现这些实现并动态的加载他们。例如，任何人都可以定制 Interface Build 的插件，Interface Build 也能自动加载这些插件。

动态加载面临的主要问题时如何使新载入的部分和程序已经运行的部分能够工作，尤其是这俩部分由不同的程序员开发。然而，在面向对象环境中，大部分问题都不成为问题，因为面向对象中的代码按逻辑模块的方式来管理，而且接口和实现分离。当类被动态加载时，它不可能破坏程序已经在运行的部分。每个类都封装了自己的实现并且有独立的名字空间。

此外，动态类型识别和动态绑定让其它程序员设计的类能够和您的程序一起工作。一旦类被动态加载，则和其它已经在运行的类没有什么区别。您的代码中可以无差别的给这些类的对象发送消息，无需关心其它程序员实现的到底是什么类，唯一需要的是你们在通信的协议上达成一致。

加载和链接： 动态加载也被叫做动态链接。程序的各部分被链接在一起，并在启动时按需读入到内存中。动态加载一般指的程序的各部分动态链接在一起，并且在运行时动态地载入到内存中的过程。

程序的组织结构

面向对象程序有两种类型的组织结构。一种是在类的继承体系在体现。另一种是在程序运行时由对象间的消息传递来呈现。这些消息揭示了对象间互相连接的网状结构。

- 类继承体系阐述了对象是怎么根据类型相关联的。例如，在用水模型的程序中，水龙头和管道可能是同一类型的对象，区别仅在于水龙头有开关阀门而管道可以和其它管道互相连接。这种相似性可以在设计中通过使水龙头和管道都由一个公共类派生来体现。
- 对象之间的连接网络描述了程序是怎样工作的。例如，装置对象发送需求水的消息给阀门对象，阀门对象再发送给管道对象。然后管道对象和建筑对象通迅，建筑对象再发送消息给其它的阀门对象，水龙头对象和管道对象等，而不是直接和装置对象通讯。如果对象间需要通讯，则必须知道对方的存在。例如装置对象可能需要到阀门的连接，而阀门对象又需要到管道对象的连接，等等。这些连接形成了整个程序的大体结构。

面向对象的程序是通过对象之间行为相互作用的网络以及类的继承体系来组织和设计的，在程序的代码中和程序运行时都能够很明显地体现出这种组织形式。

本部分包含如下内容：

插座变量的连接

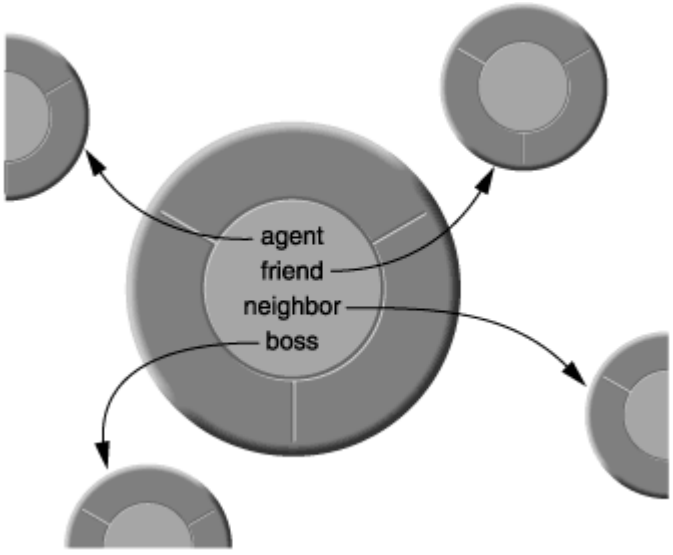
面向对象编程的部分任务就是处理对象构成的网络结构。这种网络结构并不一定是静止的，它可能随着程序的运行而相应的发生改变。对象间的关系也可能根据需而即时产生，对象所扮演的角色也可能随时改变。然后，这一切都会有个规律，就和角色都会有个剧本一样。

有些对象间的连接可能非常短暂。例如，一条消息中包含一个参数，这个参数指向的对象是消息的发送者。对象的接收者可能会发送一条响应消息给参数标识的对象，该响应消息包含了消息接收者本身或者其它需要消息发送者进一步交互的对象。上述对象之间的连接是很短暂的，仅随着消息链产生而产生。

但是并不是所有对象之间的连接都是这么临时产生的， 有一些连接是需要在程序的数据结构中记录的。有很多方法可以实现这一目的。例如可以用一个表格记录对象之间的关系，或者使用一组服务通过名字来标识对象。然而，最简单的方法是给每一个对象赋予一些实例变量来记录需要通讯的其它对象。这些实例变量叫做**插座变量（outlet），**因为它们记录了消息的出口——程序中对象之间最主要的连接关系。

虽然插座变量的命名可以是任意的，但是其名字一般反映了这些插座变量本身所扮演的角色。比如说图 4-1 中一个对象有四个插座变量——"agent"，“friend”，“neighbor“，“boss”。这些变量所指向的对象可能会随时改变，但是它们所扮演的角色是一样的。

图 4-1 插座变量



对象的某些插座变量可能在对象第一次初始化时就设定好了，并不再改变。有一些可能在随后的某些操作中被设置，也有一些可以调用对象提供的方法自由的随时设置。

插座变量被设置后，应用程序的组织结构自然地体现出来。它们使对象连接成一个互相通讯的网络，就跟用水系统因为管道而联系在一起，个体通过社会关系而联系在一起一样。

本节包含如下内容：

内部连接和外部连接

插座变量可以表示对象间多种类型的关系。有时候表示的是在程序中或多或少有一些交互的两个对象，且这两个对象各有自己角色，互不统属。例如，装置对象可能有一个插座变量标识了和该装置对象连接的阀门对象。

有时候表示的两个对象之间是从属的关系。一个对象会被看做另外一个对象的一部分。例如水龙头对象可能会使用一个水表对象来记录使用的水的数量。该水表对象仅服务于该水龙头对象。与装置对象和阀门对象的外部连接相比，水龙头对象和水表对象的连接属于内部连接。

类似的，一个对象也会有插座变量记录所有由该对象统属的对象列表。例如，建筑对象可能有该建筑内所有的管道对象列表，这些管道对象可以被认为是该建筑的一部分，它们间的连接属于内部连接，然而管道和其它管道之间属于外部连接。

内部插座变量和外部插座变量有很大的不同。一个对象被释放或者归档到磁盘上的文件时，该对象的内部插座变量所指向的对象必须随之被释放或者归档。例如，如果水龙头对象被释放了，则服务于该水龙头的水表对象必须随之被释放。如果水龙头对象归档时，它的水表对象没有随之归档，则当它被重新载入到内存中时，该水龙头对象是无意义的（除非它自己可以创建一个新的水表对象）。

而外部插座变量描述的是程序的组织结构，其所指向的对象也是程序中相对独立的组件。例如，当一个装置对象被释放时，它所连接的阀门对象仍然会存在。当一个被归档的装置重新载入到内存中时，它会连接到另外一个阀门对象并开始工作。

对象网络的激活

对象网络一般是由外界因素的促进而启动的。例如，如果您的程序是和用户接口交互的，则需要响应用户对键盘和鼠标的操作。一个对数据进行处理的程序则随着数据的输入而开始运行。有的程序可能会对电话线路的数据，从数据库获得的信息，监视程序运行的进程的状态做出响应。

程序可能会捕捉一系列的**事件**，以及外部活动的状态。例如，具有用户界面的程序会接收到鼠标的点击事件和键盘的按键事件，并对此做出响应。面向对象的程序结构能够比较好的适应这种用户驱动类型的应用。

聚合和分解

面向对象程序设计的另外一部分工作就是设计类之间关系——是通过定义子类而对现有的类进行扩展还是定义一个独立的新类。这个问题可以通过下面这个比较极端的例子来阐述：

- 假设有一个只有一个对象的程序。既然该程序仅有一个对象，那么该对象只能自己给自己发送消息。该程序谈不上什么多态，也不会用上各种类设计的模式，更不会有相互连接的对象网络。程序的真正结构被隐藏于类的定义中。虽然该程序是使用面向对象语言编写，但是和面向对象的关系极其有限。
- 与之相反，一个程序包括成百上千的不同类型的对象，每个对象仅有很少的方法和有限的功能。在这，程序的结构同样丢失在对象迷宫一样的关系中。

显然，避免出现这两种极端状况的最好方法就是既保证对象有基本的功能，同时维持对象的定义仅限于它所扮演的角色。这样程序的结构很容易从对象的网络结构中抽取出来。

然而，总会有是增加现存的类功能还是根据新加的功能定义一个新类这样的问题出现。在用水模型系统中，水龙头需要记录一段时间内的用水量。为此，您可以在水龙头类的定义中实现这一功能，也可以象前面所推荐的一样，定义一个公用的水表类来完成该功能。每个水龙头对象都有插座变量与一个水表对象连接，该水表仅与其对应的水龙头对象交互。

最后的选择往往取决于您的设计目的。如果水表对象同样适用于其它项目，把水表的作用设计到一个单独的类中，将会大大提高代码的可复用性。如果您想把水龙头类的设计的尽量独立于其它类，那么水表的功能就可以设计在水龙头类中。

通常最好的方法就是尽可能的复用代码，避免出现一个类处理太多事情以至于该类仅能应用在它所设计的情景中。当对象被设计成一个组件时，它的复用性更好。因为组件如果能在一个系统中工作往往也能在另一个系统中工作。

按照功能来划分不同的类并没有使编程接口复杂化。如果水龙头类将水表类声明成私有的，那么水表类的接口就不用暴露给水龙头类的用户；水表类就象水龙头类的实例变量一样被封装起来了。

模型和框架

对象是状态和行为的组合，所以和现实世界的物体很相似。因为这种相似性，设计一个面向对象的程序非常类似于构造一个真实的事物——它们能做什么的，怎样去做，如何和其它的事物联系。

当您设计一个面向对象的程序时，实际上，您是把一些事情在计算机上的模拟化放在了一起。对象网络看上去很像真实系统的模型化，其行为也象。面向对象程序也可以被看作一个模型，尽管在现在世界中并没有与之相应的物体。

模型中的每个组件——也就是各种类型的对象——可以从该组件的行为，职责，以及和其它组件的交互等角度来描述。一个对象的接口是由它的行为确定的，而不是它的数据，所以您可以从某个系统组件应该完成什么功能来开始设计，而不是从怎样用数据结构来表示该组件开始。一旦对象的行为决定了，就可以选择合适的数据结构，但这是实现需要考虑的问题，而不是最初设计需要考虑的。

例如，在用水系统程序中，您不能从设计水龙头类的数据结构开始，而是从您需要水龙头类做什么开始——水龙头要能够和管道连接，可以被打开和关闭，能够控制水流大小等等。因此设计并不是从选择数据结构开始。您可以先确定行为，随后再实现数据结构。您也可以随时更改数据结构的实现而不用重新设计。

设计面向对象的程序并不一定需要写大量的代码。类的良好复用性意味着可以基于的现存的类来构建一个程序，甚至完全依赖于现存的类也是有可能的。随着类定义的数目的增长，您可以复用的类也越来越多。

从很多地方都可以得到可复用的类。一个开发项目通常能提供很多可复用的类，甚至有的企业会将一些可复用的类打包出售。面向对象的编程环境和类库密不可分。例如，Cocoa 库中有超过两百个类。这些类中，有提供基础服务（哈希计算，数据存储，远程消息）的类，也有一些提供特定的服务（例如用户界面，视频功能，音频功能）的类。

通常，一组类库中的类组成了一个不完整的程序架构。这些类被称之为软件框架。框架可以用来开发各种不同的应用程序。当您使用框架时，则意味着您接受该框架提供的编程模型，并且基于该框架进行您的设计。下面是使用框架的几种途径：

- 初始化并使用框架类的实例

- 定义框架类的子类
- 定义和框架类协同工作的类

无论哪种途径，不仅是程序适应软件框架的过程，也是通用的软件框架适应特定的应用程序的过程。

框架本质上就是为您的程序建立了部分对象网络和类的继承体系，您的代码可以基于框架来完成整个程序架构。

结构化编程

面向对象不仅能够让程序的组织结构更清晰，对于程序员而言，面向对象也有助于编程任务的结构化。

随着软件所具有的功能越来越强大，程序也随之变得越来越复杂，整个编程过程的管理也越来越困难。越来越多的组件需要协调，越来越多的程序员需要协同工作。面向对象编程具有处理这种复杂关系的优势，这不仅表现在代码的设计上也表现在对整个工作的组织方面。

本部分包含如下内容：

协作

一个复杂的软件需要一个出色的合作团队，每个成员不仅必须要具有个人创造力，而且能够与其它团队成员完美配合。

同一项目同一时间同一地点，努力的程度，人员的多寡，都会影响团队能否达到预期的目标。此外，团队的协作能力还受到时间，空间以及组织结构等方面的约束。

- 代码完成后，很长时间内都会被使用，维护以及改进。程序员在项目上工作的时间可能有先后，不一定都是同一时间工作的，所以他们很难互相讨论或者知道所有的实现细节。
- 即使在该项目中所有的程序员都是同时工作的，他们也可能分处不同的地点，这也限制了他们合作的密切程度。
- 不同团队的程序员各有各的工作侧重点，时间上有各自的安排，却经常要在项目上彼此合作。类似这种跨团队的交流不是一件很容易的事情。

解决问题的办法就是改进设计和编写程序的方式。这种改进不是以层次化的管理，严格的授权为基础，这只会阻碍人们的创造力，而是将协作融入编程本身。

这就是面向对象编程的优势所在。例如，面向对象的代码的良好的复用性，使得即使程序员身处不同的项目，不同的时间，甚至不同的团队都可以高效的合作，他们所需要做的只是在类库种共享他们的代码。这种合作具有广阔的应用前景，因为它不仅能够减轻任务的困难程度，同时也能使那些看上去不可能完成的项目得以完成。

面向对象的工程的组织结构

面向对象降低了在底层的实现细节上协作的需要，并从结构上提高了上层的协作能力，从而使得编程任务可以按照有利于协作的方式重新组织。面向对象的每个特性，从大规模设计的能力到更好的代码复用性，都有助于协作的进行。

本节包含如下内容：

高抽象层级的设计

当对问题的抽象层次比较高的时候，分工也变得比较容易。项目可以直接按照程序的设计逻辑来组织和管理。

面向对象可以保证实现不会偏离设计目标，团队的成员也对自己工作的部分在整个项目的位置有一个比较清晰的认识，从而有助于目标的实现。

接口和实现分离

在面向对象的程序中，各组件之间的联系细节在设计过程的早期，开始实现之前，就已经得到了很好的定义。

在开发过程中，只有这些组件间的接口而不是实现细节需要考虑协作的问题，因为每个类都封装了类的实现并且处于独立的名字空间。需要协作的部分越少，事情就会越简单。

工作模块化

面向对象编程中的模块化意味着一个大型程序的各个逻辑组件可以被单独地实现，不同的人完成不同的类。各自的编程任务都是独立的。

这样做不仅是为了组织和管理开发工作，也是为了更好的修正代码中错误。因为实现都是被封装在类中，所以代码出了问题可以很容易的定位。

这种按类来划分责任范围意味着类的实现可以独立的更新，例如性能的优化，采用新的技术等。只要类的接口没有变化，程序员可以随时改进类的实现。

接口简单化

类的多态性使得程序的接口更加简单，因为类之间可以重用同样的方法名字。这样导致的结果是整个程序的工作机制更易于被学习，被理解，并且更容易在开发过程中分工协作。

运行时决定

面向对象的程序更多的是在运行时做出决定，因此在编译期（源代码中）只需要做很少的工作就可以让两部分的代码很好的一块工作。相应地，分工合作变得更加容易，出错的可能性也更小。

代码继承

继承是代码复用的一种方式。如果您仅需要继承并特化一些公有类，您的设计和编程过程无疑变得非常简单，因为不同层次的实现之间的关系可以由类的继承体系确定，并且非常容易理解。

继承同样增加了代码的可复用性和可靠性。父类中的代码在子类中也会被测试到，所以类库中那些通用类将会在它们的子类中经过多次测试，这些子类可能是由不同的程序员所写，甚至是在不同的应用程序中。

重用测试过代码

如果您重用别的程序中的部分越多，您自身的工作也越简单。在面向对象中，重用的过程更为容易一些，因为代码本身的可复用性更高。如果项目的任务不重，程序员之间的协作也更容易。

面向对象的类库中的类和框架能够帮助减轻您的编程任务。例如，如果您使用了苹果公司提供的软件框架，您就可以和苹果公司的程序员合作。这些框架通常构成了您的程序中比较基础的部分，而您可以把精力放在您最擅长的方面，其它的事情让类库的开发人员去做。因此，您的项目可以更快的做出原型系统，更快的完成，协作方面的问题也更少。

面向对象中代码良好的可复用性也提高了其自身的可靠性。类库中的类可能被广泛的用于各种应用程序和应用情景。类被使用得越多，潜在的问题就越可能被发现和修正。在您的程序中看起来很奇怪或者很难发现的问题很可能已经被别的程序员发现并排除了。