克隆的反攻

(Attack of the Clones*)

James Noble
Computer Science
Victoria University of Wellington
New Zealand
kjx@mcs.vuw.ac.nz

Brian Foote
209W. Iowa
Urbana, IL 61801
U.S.A.
foote@laputan.org

翻译：Shining Ray @ Nirvana Studio

摘要

Self是一个基于原型的编程语言，常常被说成“比Smalltalk还Smalltalk”的。本文将展示Self中的一些设计模式的实现，并且指出Self中实现和其他面向对象编程语言之间的区别。

19. A language that doesn't affect
the way you think about programming,
is not worth knowing.
Alan Perlis, *Epigrams on Programming*, (Perlis 1982)

简介

. . . 很不幸，Self并不是最后一个面向对象编程语言。不过Self还是有一些最后一个*有意思的*面向对象编程语言，或者至少说是最后一个有趣的*现代*OO语言——相比大多数后来出现的语言，C++、 Java、 C#、 Cb，它们都是换汤不换药，就好像可乐和百事：你喝得不是饮料，而是喝的广告宣传（Twitchell 2000）。在Self中，至少可以给你带来前所未有的震惊。

作为一个编程语言，Self本质上是最先进的，事实上它是最小化了Smalltalk的新式变种，在很大程度上和Scheme与Common Lisp的关系一样，一个是经典Lisp系统的最小集，另一个是最大集（Smith & Ungar 1997, Ungar & Smith 1991, Goldberg & Robson 1983, Moon, Stallman &Weinreb 1984）。Self的实现和编程环境又是另一种情况：完全令人疯狂的编译器技术和一个超现实主义的用户界面，这些都让Self系统独树一帜，比Smalltalk、Lisp和几乎任何其他同时代的更加前卫。

本文旨在通过演示Self中的对《设计模式》(*Design Patterns:* Gamma, Helm, Johnson & Vlissides 1994) 的一些节选模式的实现，来展现Self语言的独特风格。我们将关注于“实现”（学术上，解solutions (Noble & Biddle 2002)），而不是模式的目的，只是因为我们要展示的是已经存在的模式是如何特别地在Self中实现的。一般来说，我们会选择那些我们喜欢、我们认为有趣的或者是有着特别“巧妙”的Self实现的模式。我们不会向实践中的Self程序员对于这些模式的实际使用（或者模式的实现）要求任何权利：本文更多的是对于可能的模式的一种假想的探究而非程序员们那种正儿八经的描述。

形式和内容

文章大部分内容将展示我们用Self实现的模式的解决方法。这些模式的组织形式和顺序都和《设计模式》中对应的模式一致。我们将从创建型模式——原型（Prototype）和单例（Singleton）开始，接着展示结构型模式——适配器（Adaptor）、桥模式（Bridge）、装修器模式（Decorator）和代理模式（Proxy），最后以响应链、状态和策略三个行为型模式结束。本文包含了一个模式目录（也就是至少一个模式解决方案目录），而不是一整个系统或者是模式语言：实际上，它更像一个翻译字典，展示了常见的《设计模式》中的词汇如何转换成一个不常见的语言。

由于空间的原因，我们使用了一种精简的形式来展示模式。每个模式都一个名称、关于目的的一句话，和关于问题的一句话，这是从《设计模式》相应的章节中学来的。*解决方法*的结构、参与者、协作和实现都跟在后面，会基于一些源代码的例子。对于已知的使用方法不会再做进一步讨论。对于那些还不是全球Self程序员社群中一员的读者来说（至少有10%不是吧？），附录中添加了一份对于Self的简介。

鸣谢

十分感谢KoalaPLoP作者工坊的成员——Pronab Ganguly、Pradip Sarkar、Marie-Christine Fauvet——当然，还要感谢广大的Self支持者。

代表

我们相信很多**设计模式**都抓住了存在于面向对象软件设计中深层的普遍性质。本文是一个实验来确定模式是否（怎样）出现在Dave和Randy的杰出的探险成果中的：这有点奇怪的、极其简洁的、总是可爱地要命

的面向对象编程语言——Self。

原型模式（Prototype）

目的

使用样板实例来指明要新建的某一类对象，并通过复制这个原型来创建新的对象。

问题

你常常需要创建某种对象来完成特定的任务。你如何才能创建正确类型的新对象呢？

例子

下面一段代码展示了一个独立的可用作原型的点对象。clone方法是利用底层的原语操作_Clone来实现的。这就是实现这样一个对象所需要的全部东西，你可以根据需要添加消息（比如这里列出的等于）。

```
Point = ( |
        x.
        y.
        = aPoint = ((x = aPoint x) && (y = aPoint y)).
        clone = (_Clone).
| )
```

实现

事实上在Self的实践中，你不会这样去实现一个对象。你要把方法移到一个单独的对象中（这样他们就会被继承而不是每次都要被复制），并且继承系统对象 traits clonable。

该traits clonable对象提供了对于clone方法的精确定义，加上一些标准的打印、比较和其他消息。

```
Point = ( |
        parent*= traits point.
        x.
        y.
| )
traits point = ( |
        parent* = traits clonable.
        = aPoint = ((x = aPoint x) && (y = aPoint y)).
| )
traits clonable = ( |
        "这里删除了很多其他方法"
        clone = (_Clone).
| )
```

讨论

这个模式已经内置到Self中了——其实，这是Self程序唯一的创建新对象的途径。正因为如此，所以Self才被称为基于原型的编程语言。在Self中，我们总是使用clone消息来复制一个已经存在的对象来创建新对象的，而这个消息最终是通过系统级原语_Clone来实现的。

要让一个对象的原型在程序中可以被访问，你要把这种类别其中一个对象放入全局名空间对象中，在这里程序可以根据需要获取并克隆它。在Self中，这些全局名空间只不过是一些标准对象，如果有些对象需要访问里面的对象，只要继承他们就行了：无论何时用到原型的名字都可以通过继承来找到它。

单例模式（Singleton）

目的

确保一个类只有一个实例，并提供一个全局的访问点。

问题

有些类有且只有一个实例。例如，代表真实的硬件资源的对象，如磁盘、打印机或者特殊的系统对象如回收站，也只能一次出现一个。

解决方法

Self对象只能通过clone方法来创建。所以你通过将clone重新定义为什么也不做的方法——也就是直接返回对象自身，就可以让任何对象变成单例。

例子

在Self中重新定义clone十分简单。例如，你可以让一个特殊的点对象代表原点：给这个对象发送clone消息将返回它自身：

```
origin = ( |
        clone = (self).
        x = 0.
        y = 0.
| )
```

实现

Self提供了一个 traits oddballs （和 traits clonable 类似），通过继承它可以获得一个单例。同样的，traits oddballs还有很多其他比较的方法，如等于和其他可以正确应用在单例上的方法。

```
Origin = ( |
        parent* = traits oddballs.
        X = 0.
        y = 0.
| )

traits oddballs = ( |
        "删掉了很多方法"
        clone = (self).
| )
```

讨论

《设计模式》*Design Patterns* 描述了如何泛化单例模式来控制可以创建的对象的数量：你同样可以在Self中完成这个功能。例如，要限制一个给定原型只能有六个范例，只要维护一个cloneCount变量。这个变量要在所有克隆出的对象中可访问，所以放置他的最佳位置就是原型所继承的**特性**对象。

```
SillyPoint = ( |
        parent*= traits sillyPoint.
        X <- 0.
        y <- 0.
| )
traits sillyPoint = ( |
        parent*= traits clonable. "我们可以克隆这些东西!"
        cloneCount <-  0.
        clone = ((cloneCount < 6)
                ifTrue: [cloneCount: cloneCount succ.
                                resend.clone]
                False: [error: 'Attack of the Clones!'])
| )
```

适配器模式（Adaptor）

目的

把一个类的接口转换成另一个用户期望的接口。适配器模式让本来由于不兼容的接口而不能一起工作的类变得可以一起工作。

问题

先不看基于类的模式语言的目的陈述。我们可以把对象看作他们自己的世界，每个都有他们自己的文化、语言和风格。系统常常需要连接几个不同的世界，而其中的对象的接口（他们所理解的消息）可能会极为不同。

解决方法

产生一个用于把*被调整对象*（*Adaptee*）的接口匹配为*目标*（*target*）接口的适配器对象。

例子

例如，你可以把点对象（在单例模式中出现的那个）调整为可以提供极坐标的点，类似于以下的东西：

```
polarAdaptor = ( |
        adapteePoint.
        theta = ((adapteePoint y / adapteePoint x) arctan).
        "需要修正。我们不进行数学运算."
        r = ((adapteePoint x squared + adapteePoint y squared) sqrt).
| )
```

实现

一般的方法会把这个适配器分成一个原型和一个特性对象。你可能需要注意像`clone`之类的消息是否正确处理（例如，克隆应该复制被适配的对象并返回一个新的适配器加上这个对象）。

讨论

还有一些可选的利用继承来完成更多的事情的方式。例如，我们可以继承被适配的对象：

```
polarAdaptor = ( |
        adapteePoint*.
        "从adaptee继承"
        theta = ((y / x) arctan).
        r = ((x squared + y squared) sqrt).
| )
```

这里给出了一个类似于基于类的适配器模式的设计，但是是使用了对被适配者的单一继承。因为Self是动态类型的（类似于Smalltalk），所以对于适配器没有必要继承某种目标接口（Alpert, Brown & Woolf 1988）。Smalltalk Companion 也列出了很多不同其他交织的Smalltalk中十分有用的可参数化的适配器，带来的最关键的好处是他们可以进行弹性适配调整而不用创建一个新的Smalltalk类（Alpert et al. 1988）。由于在Self中对象都是自定义的，我们不用担心大量出现的单一使用的类：单一使用的对象在这里正好。

桥模式（Bridge）

目的

把一个抽象从他的实现中分离，达到解耦的目的，这样使两者可以独立变化。

问题

有时候你需要一个类库。例如，一个集合库可以提供一系列不同的集合抽象（像set、bag、dictionary），同时每个抽象又有一系列不同的实现（以数组、链表、散列表等的方式）。

解决方法

产生两个对象的层次结构——一个针对抽象，另一个则是实现。抽象部分的对象为实现部分的对象代理了一些或者全部的行为。

例子

在Self中，抽象对象可以使用一个可变的parent槽来继承他们的实现，这样如果需要的话就可以改变实现形式。我们来看集合的例子，我们可以有一些抽象：

```
bagAbstraction = ( |
        myImplementation*. "可变parent槽"
        add: element = (append: element). "继承append"
| )
setAbstraction = ( |
        myImplementation*. "可变parent槽"
        add: element = ((contains: element) ifFalse: [append: element]).
        "继承contains和append"
| )
```

同时可能的使用他们的实现是：

```
listImplementation = ( |
        myList.
        append: element = (myList insertAtEnd: element).
        contains: element = ( ... ).
        ...
| )
vectorImplementation = ( |
        myVector. myIndex.
        append: element = (vector at: myIndex Put: element.
                myIndex: myIndex succ).
        contains: element = ( ... ).
| )
```

抽象提供了对用户级方法如add之类的定义，而他们又是根据通过myImplementation*可变parent槽继承的实现级方法（如append和contains）来写的。

实现

一个正式的实现会使用多继承把抽象和实现对象分离到不同的层次结构中，这样一个抽象对象可以同时从他的抽象特性和他的实现对象中继承。很奇怪让人想起了《设计模式》中描述的C#多重继承适配器，不过，当然是通过对象而非类来完成的。

讨论

这是在Self中使用动态继承的一个非常好的例子：如果一个抽象需要替换它的实现，那么他可以这样做，先克隆一个新的实现，根据原来的对它进行初始化，然后替换原来的（代码写起来比写英文还少）。

```
setAbstraction = ( |
        useListImplementation = (
                myImplementation: listImplementation clone
                        initialiseFrom: myImplementation).
| )
```

装饰器模式（Decorator）

目的

动态地给一个对象附加额外的职责。装饰器提供了弹性的扩展功能的候选。

问题

一些对象很烦人也不负责任，把阴影错误地涂成灰棕色，不过其他的还是正常的。

解决方法

使用装修器来重新装饰你的对象！产生一个特殊的提供你所需要的全新的行为的对象，并把它包裹在破旧的对象外面。

例子

我们有一个平淡无奇的房间对象：

```
room = ( |
       colour = 'brown'.
       carpet = 'beige'.
       printString = ('A ', colour, ' room with ', carpet, ' carpet.').
| )
```

我们可以用一个装饰器来修饰它

```
puceColourDecorator = ( |
       parent*= room.
       colour = 'puce'.
| )
```

或者用两个

```
lighteningCarpetDecorator = ( |
       parent*= puceCarpetDecorator.
       carpet = ('light ', resend.carpet).
| )
```

这时墙壁的颜色是深褐色的，窗帘则是浅灰棕色的。极大的改进啊。

实现

Self常常关于“自身问题”（self problem, Lieberman 1986）“做正确的事情”（do the right things）；所以，比如对一个完全装饰过的房间对象调用`printString`将输出“A puce room with light beige carpet.”。而使用绑定到对象身份的原语则会失败，也就是，你可以区分普通房间和一个被装饰过的房间。而通过在装饰器中重新定义一致性比较消息会让你混淆这个问题：

```
== x = (parent == x). "和我的父对象比较，不是和我比"
```

（但如果你已经能设法解析这段代码，那么你已经对本文将要叙述的所有问题都已经了解了）

如果你有很多代码很简单的装饰器，你可以将它们进行重构，用独立的特性对象（就像我们为适配器模式描述的那样）放到一个继承层次中。但是，像很多美国总统，装饰器是由他们所*不是*的东西来定义的——由于继承，他们只要描述自己的职责——所以常常不值得。

讨论

这是Self中另一个好看又自然的动态继承的例子。你也可以像Smalltalk一样使用`doesNotUnderstand`——不过*Smalltalk Companion*并没有提到他因为，呵呵，他们只是“懦夫”（“wimps”, Alpert et al. 1988）。你完全可以使用Rappers在Smalltalk中做完全一样的东西（Brant, Foote, Johnson & Roberts 1998）。

代理模式（Proxy）

目的

为另一个对象提供一个替身或者占位符来提供对他的访问。

问题

一些对象当你想用他们的时候，往往不存在——也许他们在磁盘上、也许他们在另一台机器上……多麻烦啊！

解决方法

为远程或者是昂贵的对象生成一个代理对象来事先站好位置。把代理放在任何你本来要放原来对象但不能放的地方，同时让代理的行为看起来和原来的对象一样。

例子

Self中的代理（对于本地对象）可以和装修器模式一样用：

```
doNothingProxy = ( |
        "什么也不做的代理".
        parent*= realSubject.
| )
```

当然，会进行某些处理的代理更加有意思，例如这个虚拟代理，当它收到一个消息的时候就从磁盘上载入一个对象：

```
virtualProxy = ( |
        parent* <-  ( |
                "占位符"
                loadObjectFrom: filename = ( ... ).
                someMessage = (
                        (parent: loadObjectFrom: myFilename).
                        resend.someMessage).
        | )
        myFilename <- 'object.self'.
| )
```

这种情况下我们已经使用了占位符父对象（类似一种特殊的Null Object（Woolf 1998）），并安排它当收到消息时就从磁盘上载入对象——Self的继承规则会给不同的名字进行正确的排序，这样`parent`就会被替换成`loadObjectFrom`消息的结果。当然你可以使用`undefinedSelector`陷阱来一次捕获所有的消息，而不用定义他们：

```
virtualProxy = ( |
        parent* <- ( | | ). "空对象，empty object, not even nil"
        myFilename = 'object.self'.
        undefinedSelector: msg = (
                parent: loadObjectFrom: myFilename.
                parent perform: msg).
| )
```

（注意正式的实现需要处理消息参数——这个例子简化了`undefinedSelector`消息）。

实现

当代理本地对象时，通过继承的实现是可以的。对于远程对象，处理可能要更复杂，例如要捕获`undefinedSelector`异常（类似Smalltalk的`doesNotUnderstand`）因为你需要代理的对象可能还不是他们。一般来说，有三个策略可以用来实现这种东西：手工代理消息，使用（动态）继承，和捕获异常——从最直观的到最奇怪的。

讨论

你要变得多牛？

你要浪费多少程序时间（lifetime）呢？

职责链（Chain of Responsibility）

目的

通过给予多个对象处理请求的机会，来避免把请求的发送者和它的接受者耦合起来。把接收对象连接起来并沿着这个路径传递请求直到有一个对象对他进行了处理。

问题

有些时候一个对象的处理能力并不足以正确相应一个消息。也许职责是被几个协作的对象分享的，或者也许对象已经被组成了一个结构，同时是这个结构指明了要如何处理这个消息，而不是单个对象。例如，在图形化用户界面中请求一个上下文相关帮助也许无法由任何单独的对象所响应，它可能需要从一个按钮把消息传到面板再传到窗口。

解决方法

结合一个额外的描述职责链的继承链接。然后，确保一些请求是沿着链接传送的，而不是被单个处理了。

例子

我们可以安排一个单独的沿着从一个窗口指向他的构造容器的container链接的继承链。

```
buttonWindow = ( |
        parent*= traits buttonWindow. "一般继承"
        container<- panel. "结构的包含情况"
        windowTitle <- "Push Me".
| )
traits buttonWindow = ( |
        display = ( ... ).
        "不要在这里处理帮助消息"
| )
panel = ( |
        parent* = traits panel. "一般继承"
        container <-  topLevelWindow. "结构的包含情况"
        help = "Cutesy help message".
        display = ( ... ).
| )
```

如果buttonWindow收到了一个help消息，他会消息交给它的容器panel处理。

实现

这个设计从理论上看上去很优雅，但不幸的是在实践中存在着一些问题。问题是，可能会有很多消息，而消息在结构链和一般分类层次关系上产生歧义——给buttonWindow发送display消息会引发一个错误，因为Self无法辨别panel中的display方法还是traits buttonWindow中的display方法。

要让这个能完全正常工作，你必须选择哪个消息可以在哪个链上找到，然后通过某种方式确保没有歧义的定义，或者如果有必要就从对象中删除或更名方法。如果假定一个叫做help的消息要通过结构container链接进行响应，那么就要确保help不会通过分类上的parent链接实现：不过也许可以重命名分类结构上的help方法为defaultHelp，当调用根对象resend时，发送这个消息。

讨论

Wow! 有些东西实际上使用了动态多重继承。可惜的是运行得还不是很好：歧义的错误会让他在实际运用中行不通。老的版本，Self 2.0，结合优先级继承的方式却工作正常（只要把结构的父对象的优先级设置得比继承的父对象低就性了），直到Randy让Dave出局并删除了优先级(Smith & Ungar 1997)。

我猜特别资深的Self程序员会考虑使用对于未定义的选择器异常的代理操作（或者，甚至是处理歧义选择器异常）来正确处理这个事情。

其它语言可以这样做的唯一途径是构造特殊的方式来做（像NewtonScript（Smith 1995）或者Dewan的双向继承模式（Dewan 1991））。Self也是没有额外代价就完成了这个事情。

观察者模式（Observer）

目的

在对象之间定义一对多的依赖，以便当一个对象的状态变化时，所有依赖他的对象都可以自动被提醒和更新。

问题

这个模式提出了两个问题：你怎样才能检测一个对象什么时候变化了，以及如何通知其他对自身变化感兴趣的其他对象。Self队第二个问题没有提供特别的支持，但是却让确定对象什么时候产生了变化更加简单。

解决方法

重命名一个对象的数据槽并把它替换成访问器方法。并让写入器方法自动提醒观察者。

例子

这里是我们修改了的点对象这样他会在任何一个槽被赋值的时候引发一个更改。

```
subjectPoint = ( |
        privateX.
        privateY.
        x = (privateX).
        y = (privateY).
        x: newX = (signalChangeOf: 'x' to: newX. privateX: newX).
        x: newY = (signalChangeOf: 'y' to: newY. privateY: newY).
| )
```

当然他们需要实现“`signalChangeOf:to:`”消息来处理这个变化。

实现

这里关键的地方是：某个叫做`foo`的槽，它是通过访问器消息`foo`来读取并通过设置器消息`foo:`来写入的。一个外部的对象是无法分辨出是获取器和设置器方法实现了这些消息，还是仅仅是一个数据槽。这两个方法（也就是`foo`和`foo:`）对于所有的客户来说和某个叫做`foo`的变量完全一样。

讨论

想想就觉得有趣，C#给语言引入了大量CRUD操作（属性properties）只是为了处理这类问题——而Self的精简的设计则不用任何额外代价就可以解决它。

状态模式（State）

目的

让一个对象当内部状态更改的时候改变它的行为。而看上去这个对象好像改变了他的类。

问题

你现在要处理一个自身行为依赖于它内部状态的对象，而它用了很多条件代码来达到这个目的。

解决方法

状态模式把每个的状态分支放在一个单独的对象中。这让你将对象的状态看作是可以独立于其它对象自己进行变化的对象。

例子

事实上某些Self的集合对象就是这么做的，在他们为空和不空时提供不同的行为。

```
list = ( |
       parent* =  traits emptyList.
       ... "剩下的槽放在这里!"
| )
traits emptyList = ( |
       parent*=traits list.
       add: e = (parent: traits fullList. ... "随便它做什么")
       ...
| )
traits fullList = ( |
       parent*= traits list.
       removeAll = ("随便它做什么". parent: traits emptyList)
       ...
| )
```

列表对象更具它是否为空的状态选择性地继承两个状态对象中的一个：而这两个又继承了包含了所有的状态无关的行为的`traits list`对象。

实现

这里上下文对象（`list`）只有一个状态对象，所以我们让唯一的父槽成为动态的。我们也可以使用一个静态父槽并继承一个使用动态继承的状态对象（就像在策略模式中例子所演示的那样）。

讨论

状态模式和策略模式的解决方法很大程度上是一样的（除了状态对象通常是封装在他们的上下文中，同时上下文可以自动更改他们，而策略对于客户是可见的同时不一般不由他们的上下文进行更改）。这两个模式之间的区别主要在于目的上，而不是解决方法的结构（Noble & Biddle 2002, Alpert et al. 1988）。

策略模式（Strategy）

目的

定义一个算法族，并使得他们可以相互更改。

问题

你有一个对象需要可以使用几种备选的算法（同时可能要在他们之间进行切换）每一个都有自己的接口和可资定义的参数。

解决方法

用一个独立的策略对象来代表每一个算法，并且把算法的参数放入策略对象的属性中。让使用算法的上下文对象继承合适的策略对象。

例子

我们来看一个需要在不同的渲染算法中进行切换的图形窗口。我们可以让每个算法放入一个单独的渲染策略对象中并且当我们需要更改算法的时候切换对象。Self的好处是我们可以使用多重动态继承来把上下文链接到策略对象上。

```
contextWindow = ( |
        parent*= traits contextWindow. "标准继承层次"
        open = ( ... ).
        close =( ... ).
        x. y. width. height.
        renderingStrategy* <- renderingStrategy. "继承策略"
| )
traits contextWindow = ( |
        display = (render).
| )
renderingStrategy = ( |
        render = (drawRectangleX: x Y: y Width: w: Height: h. ...)
| )
```

请求`contextWindows`显示自己的时候会通过继承调用`renderingStrategy`的`render`方法。不管什么时候我们要更改策略的时候，我们只要给`renderingStrategy`槽分配一个新的对象。

实现

由于Self的继承规则运行的方式（和Smalltalk或Java很相似，不过是在对象之间而不是在类之间）策略的方法可以通过给自己发送消息来访问上下文中的所有的槽和方法（上下文的行为就像策略对象的子类，所以策略发送的消息可以被上下文中的所覆盖）。继承有效地结合了所有在上下文和组件之间传送信息的标准方法（Noble 2002）。

讨论

这又是另一个结合了动态继承和多重继承的例子。因为状态对象的接口是如此受限制的，所以你可以避开职责链中使用多重继承可能出现的歧义问题来完成它（甚至是一个单个对象有多个相交的策略时）。

SELF

本节将介绍SELF编程语言(The Self Group 1992, Ungar& Smith 1991)同时旨在为读者提供足够的背景知识以便理解在前面章节中所展示的那些Self的例子。本章以对象和表达式的描述开始，介绍了Self程序的组成形式，然后展示一个面向对象的栈的例子。

对象

SELF对象是一个命名槽的集合，并且是按照一个槽的列表来书写的，用“(|”和“|)”围起来。每一个槽都和一个消息选择器相关联，并且可以存放一个常数、变量或者一个方法。例如，下面的例子定义了一个叫做“fred”的对象，其中定义了一个常数槽“pi”，以及带一个参数的关键词消息“circ:”还有一个数据（或变量）槽“size”。

```
fred = ( |
       pi = 3.14159265. "常数"
       circ: r = (2 * pi * r). "方法"
       size <-  3. "变量"
| )
 "注释是放在双引号中的"
```

对象创建

在一个正在执行的程序中，新的对象是通过*克隆*（*clone*）——逐个槽进行复制——创建的。每一个对象都有一个唯一的*身份*（*identity*）。两个对象可以有一样的槽，同时这些槽里面的有相同的值，但是如果这两个对象是分开创建的，他们仍然可以通过他们最根本的身份来进行区别。特别的，如果一个对象的变量槽的值变化了，不会影响到任何其他的对象（Baker 1993）。

作为一种模式来使用来进行克隆的那些对象是就是被称作为*原型*（*prototype*）的对象。和Lisp以及Smalltalk一样，Self使用了一个垃圾收集器这样无需明确地进行对象的销毁。

某些类型的对象可以直接使用简单的语法来进行创建，这些包括数字、字符串和块。

表达式

SELF的表达式语法是直接从Smalltalk中派生出来的。由于Self是一个纯粹的面向对象语言，几乎所有的表达式要么是直接量要么是给*接收器*（*receiver*）对象发送消息。有三种消息类型：一元、二元和关键字消息，消息的类型是由他的选择器和它需要的参数的个数决定的。

一元消息（Unary messages） 仅仅命名了一个操作，除了消息接收器对象外不提供任何参数。例如：“top”、“isEmpty”和“isFull”都是一元消息。

二元消息（Binary messages） 除了接收器之外还提供了一个参数。他们的接收器必须是由非字母的字符的组合。“+”、“-”和“*”都是二元消息，与加法、减法和乘法相对应。类似的，“@”从两个数字中创建一个点，同时“##”则从两个点创建一个矩形。

关键字消息（Keyword messages） 是Smalltalk和Self语法中最奇特的地方。他们提供了带一个或多个参数的消息。一个关键字消息有一个特定的数量，同时消息选择器是被分成很多部分的。一个关键字消息是写成把参数值散步在其中的关键字的序列。例如：“at:Put:”是一个有两个参数的关键词消息用于数组赋值。Pascal代码：

```
a[i] := j;
```

写成Self就是

```
a at: i Put: j.
```

可以直接给由其他消息返回的结果发送消息。括号可以用来进行分组。例如：

```
draw = (style drawLineFrom: start negated To: finish negated).
c = ( (a squared + b squared) squareRoot ).
```

隐含的Self

唯一既不是直接量对象或者消息发送的表达式是关键字“self”。这个表达式将给出当前的对象，也就是，当前正在执行的方法的接收器。给“self”发送的消息已经被广泛了解为“self-send”。因为“self”在Self中的使用很普遍（特别是如下面要描述的进行变量访问的消息），所以只要可能的地方，都可以把它从语法中省略掉。而这个语言被命名为“SELF”正是为了纪念这个无处不在却很多情况下看不见的关键字。

变量访问

变量槽和常数槽都是通过发送一个与其名字一样的一元消息来读取的，同时一个变量槽通过带一个参数的

关键字消息来进行存储，同样与变量槽的名字一样。SELF 代码如下:

```
foo: 43.
bar: foo.
```

当“foo”和“bar”是变量的时候，大致等同于Pascal中的：

```
foo := 43;
bar := foo;
```

而当“foo”和“bar”是方法的时候，它等同于：

```
foo(43);
bar(foo);
```

使用消息访问变量是Self和Smalltalk之间的一个主要的区别。

块（Block）

块代表了lambda表达式。例如，表达式 $\lambda xy.x+y$ 在Self中写成这样：

```
[ | :x. :y|  x + y].
```

一个块可以有参数和临时变量：这些在块的开头进行声明，放在两个“|”符号中间。参数前有冒号“:”，而临时变量没有。块是用来实现控制结构的，同时要与关键字消息同时使用。例如：

```
n isEven ifTrue: ['n is even!' printLine]
           False: ['n is odd!' printLine].
```

关于“ifTrue:False:”关键字消息并没有什么特别的地方，但是与Lisp不同，它不是一个特殊的form或macro。他的参数必须封闭在块内来避免过早地被计算：“ifTrue:False:”会自己计算合适的参数块。

迭代子（Iterator）

块可以用来映射函数和迭代子。例如，集合提供了一个“do:”消息，它可以将某个带一个参数的块应用到每一个集合元素上。例如，一个集合中所有元素的总和可以通过给“do:”消息传送一个把每个元素相加的块来进行计算：

```
total: 0.
collection do: [ | :item. | total: total + item].
```

返回值

The “^” 前置操作符是用来提前从一个方法中返回。他的语义本质上和C的return语句是一样的。一个针对字符串进行集合线性搜索“'foo'”的过程可以结合一个迭代子和一个返回操作符来完成。

```
collection do: [ | :item. | item = 'foo' ifTrue: [^true]].
^false.
```

继承

由于 SELF 没有类，所以对象可以直接通过使用parent槽来直接继承其他对象。声明一个parent槽的方法是在名称后面加上一个星号“*”。当给一个对象发送消息时，会使用一个*消息查找*算法来找到可以执行的方法或者是可以访问的变量。消息查找算法查找消息的接收器，然后递归查找接收器的所有父对象。如果无法找到消息的任何一个实现，那么会抛出一个“*undefined selector*”（未定义的选择器）异常。

```
foo = ( |
       fred = ('Implemented in foo' printLine).
| ).
bar = ( |
       parent*= foo.
       nigel = ('Implemented in bar' printLine).
| ).
```

例如，在上面的两个对象中，给“bar”对象发送“fred”或者“nigel”消息会成功执行。而给“foo”发送“nigel”或者随便给他们中任意一个发送“thomas”都回导致一个未定义选择器的异常。

特性（Traits）和原型（Prototype）

继承常用来把对象分成两个部分——*原型*和*特性*。通常特性对象包含了由所有通过原型克隆出来的对象所共享的行为（方法槽），同时原型包含了每一个克隆对象的“实例”变量和一个指向特性对象的parent槽。原型大体上对应于Smalltalk的实例，而特性则对应于Smalltalk的类（Ungar, Chambers, Chang & H¨olzle 1991, Chambers, Ungar, Chang & H¨olzle 1991）。

多继承和动态继承

一个对象可以包含多个parent槽来获得多重继承：如果一个方法查找算法找到了多个匹配的方法，那么就会引发一个“*ambiguous lookup*”（查找中有歧义）异常。Parent槽即可以是常量也可以是变量：这可以提供动态继承，它能让继承层次在运行时进行改变。例如，一个二叉树节点可以使用一个可变parent槽和两个备选特性对象来实现。一个特性对象是用于表达空树节点，另一个表示包含值的树节点。一个节点在

创建的时候为空，是用空节点特性对象。当一个节点存入了值，它就切换parent槽，这样它现在就继承于一个非空的树节点特性对象。

Resend

一个方法可以使用resend操作（前置于一个消息接收器）来调用方法的一个继承的版本。在下面的例子中，如果给“bar”发送“fred”的话，会打印出“Implemented in bar”和“Implemented in foo”。

```
foo = ( |
        fred = ('Implemented in foo' printLine).
| ).
bar = ( |
        parent*= foo.
        fred = ('Implemented in bar' printLine.
                resend.fred).
| ).
```

SELF 库

SELF拥有一个包含了超过两百个原型对象的库。这些对象分成集中不同类别：

控制结构 诸如block、true和false之类的对象提供了像“ifTrue:False:”之类的消息实现了基本控制结构。

Numbers数字 SELF 同时包含了整型和浮点数.

集合 这是Self对象中最大的分类，集合对象是用来存储其他对象的。Self的容器包括定长数组vector和byteVector；针对字符的特殊集合string；哈希表或者树来实现的set和dictionary；双向的list；以及可以用来同步多个进程的shareQueue。

几何 像point、extent和rectangle等对象提供了基本的二维几何运算。

镜像 SELF 结构上是反射的（也称自省）。这个功能是通过mirror对象来提供的，它可以反射出其他对象的结构。每一个mirror和另一个对象——mirror的被反射者（*reflectee*）——相联系。镜像可以处理names（返回所有槽的名子）、localDataSlots（返回所有的数据槽）和localAssignmentSlots（返回所有的赋值槽）等消息。

外部代理 各种不同的代理对象提供了对由C或者C++写的函数和对象的访问能力。Tarraing´ım使用代理来提供到**X**窗口系统的图形输出。

范例 SELF 程序

例1和例2包含了一个Self版本的栈的实现。

例1包含了stack对象的定义。它被分成了两个对象，traits stack包含了方法声明，以及一个stack原型对象，它继承了traits stack。Push和pop方法是公有地从traits stack中导出的，同时在stack原型中，所有的槽都应该是私有的。因为原型直接继承特性对象，所以在traits stack中定义的方法可以直接访问定义在原型中的数据槽。特性对象类似地从traits collection继承了额外的行为。Traits stack对象提供了一个clone的定义来创建新的stack。它使用了resend操作来调用定义在collection中的clone方法，然后克隆了contents向量，它不会在不同的栈之间共享。例2显示了一个使用Self版本的栈的程序。块被广泛使用来实现循环控制结构。

例3展示了一个快速排序的Self实现。

正如你可以从这个例子中所见的，Self的语法的确是一个优美的东西，但是……

```
"栈的定义"
traits stack = ( |
        parent*= traits collection.
        push: c = (contents at: index Put: c. index: index + 1).
        pop = (index: index . contents at: index).
        isEmpty = (index = 0).
        clone = (resend.clone contents: contents clone).
| )
stack = ( |
        parent*= traits stack.
        Contents <- vector copySize: 80.
        index .
| )
```

例1：栈对象的Self定义

```
main = ( |
       lines <- 0.
       s <-  stack.
       handleLine = (
              [eoln] whileFalse: [s push: read].
              [s isEmpty] whileFalse: [s pop write].
              lines: lines + 1).
       initialise = (s: stack clone).
       reverse = (
              initialise.
              [eof] whileFalse: [handleLine].
              ('Reversed: ',lines,' lines\n') printLine).
| )
```

例2：使用栈对象的Self程序

```
quick = ( |
       quickSort = (quickSortFrom: 0 To: size predecessor).
       quickSortFrom: l To: r = ( |  i. j. x. |
              i: l.
              j: r.
              x: at: (l + r) / 2.
              [
                     [(at: i) < x] whileTrue: [i: i successor].
                     [x < (at: j)] whileTrue: [j: j predecessor].
                     i <=  j ifTrue: [
                            i = j ifFalse: [swap: i And: j].
                            i: i successor.
                            j: j predecessor.
                     ].
              ] untilFalse: [ i <= j ].
              l < j ifTrue: [quickSortFrom: l To: j].
              i < r ifTrue: [quickSortFrom: i To: r].
              self).
| )
```

例3：Self中的快速排序

参考资料

- Alpert, S. R., Brown, K. & Woolf, B. (1988), *The Design Patterns Smalltalk Companion*, Addison-Wesley.
- Baker, H. G. (1993), 'Equal rights for functional objects or, the more things change, the more they are the same', *OOPS Messenger* **4**(4).
- Brant, J., Foote, B., Johnson, R. E. & Roberts, D. (1998), Wrappers to the rescue, *in* 'ECOOP Proceedings', pp. 396 – 417.
- Chambers, C., Ungar, D., Chang, B.-W. & H¨olzle, U. (1991), 'Parents are shared parts of objects: inheritance and encapsulation in Self', *Lisp And Symbolic Computation* **4**(3).
- Dewan, P. (1991), 'An inheritance model for supporting flexible displays of data structures', *Software—Practice and Experience* **21**(7), 719 – 738.
- Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. (1994), *Design Patterns*, Addison-Wesley.
- Goldberg, A. & Robson, D. (1983), *Smalltalk-80: The Language and its Implementation*, Addison-Wesley.
- Lieberman, H. (1986), Using prototypical objects to implement shared behavior in object oriented systems, *in* 'OOPSLA Proceedings'.
- Moon, D., Stallman, R. & Weinreb, D. (1984), *The Lisp Machine Manual*, M.I.T. A I Laboratory.
- Noble, J. (2002), Need to know: Patterns for coupling contexts and components, *in* 'Proceedings of the Second Annual Asian Pacific Conference on Pattern Languages of Programs', pp. 196 – 206.
- Noble, J. & Biddle, R. (2002), Patterns as signs, *in* 'ECOOP Proceedings'.
- Perlis, A. (1982), 'Epigrams on programming', *ACM SIGPLAN Notices* **17**(9).

- Smith, R. B. & Ungar, D. (1997), Programming as an experience: The inspiration for Self, *in* J. Noble, A. Taivalsaari & I. Moore, eds, 'Prototype-Based Programming: Conecepts, Languages, Applications', Springer-Verlag.
- Smith,W. R. (1995), Using a prototype-based language for user interface: The Newton project's experience, *in* 'OOPSLA Proceedings'.
- The Self Group (1992), *Self Programmer's Reference Manual*, 2.0alpha edn, Sun Microsystems and Stanford University.
- Twitchell, J. B. (2000), *twenty ADS that shook the WORLD*, Three Rivers Press.
- Ungar, D., Chambers, C., Chang, B.-W. & H¨olzle, U. (1991), 'Organizing programs without classes', *Lisp And Symbolic Computation* **4**(3).
- Ungar, D. & Smith, R. B. (1991), 'SELF: the Power of Simplicity', *Lisp And Symbolic Computation* **4**(3).
- Woolf, B. (1998), Null object, *in* R. C. Martin, D. Riehle & F. Buschmann, eds, 'Pattern Languages of Program Design', Vol. 3, Addison-Wesley.