IOS设计模式浅析之原型模式(Prototype)--copy

作者：dlm_211314 (http://my.csdn.net/dlm_211314)

IOS设计模式浅析之原型模式(Prototype)

原文地址：http://www.cnblogs.com/eagle927183/p/3462439.html

感谢Gof Lee

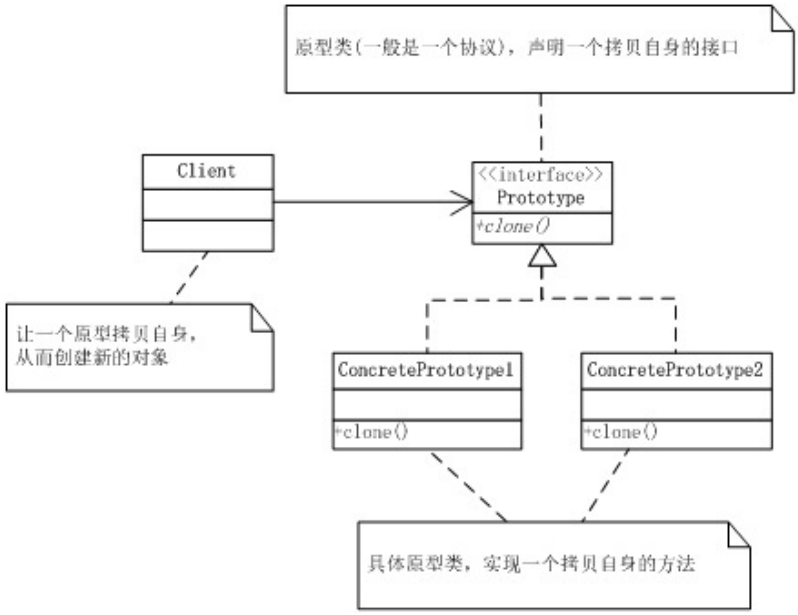
原型模式的定义

“使用原型实例指定创建对象的种类，并通过复制这个原型创建新的对象”。最初的定义出现于《设计模式》(Addison-Wesley,1994)。

简单来理解就是根据这个原型创建新的对象，而且不需要知道任何创建的细节。打个比方，以前生物课上面，有一个知识点叫细胞分裂，细胞在一定条件下，由一个分裂成2个，再由2个分裂成4个……,分裂出来的细胞基于原始的细胞(原型)，这个原始的细胞决定了分裂出来的细胞的组成结构。这种分裂过程，可以理解为原型模式。

结构

图

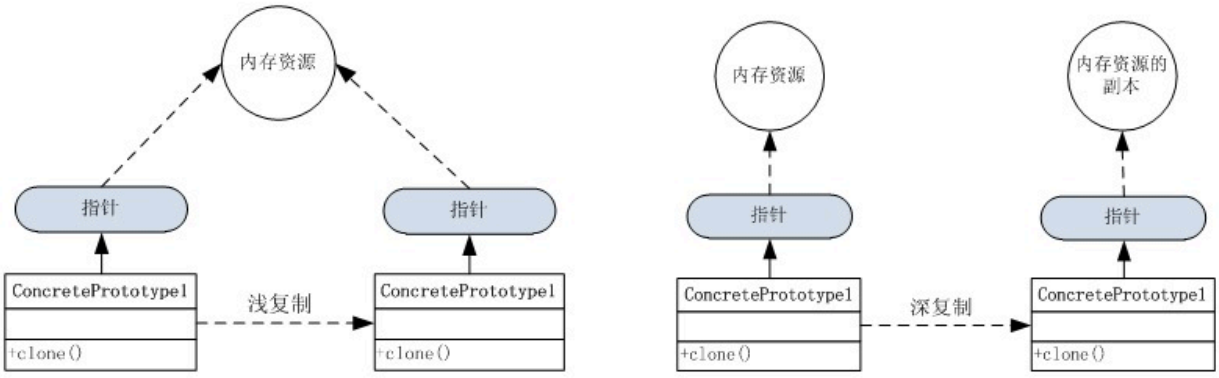


从上图可以看到，Prototype类中包括一个clone方法，Client调用其拷贝方法clone即可得到实例，不需要手工去创建实例。ConcretePrototype1和ConcretePrototype2为Prototype的子类，实现自身的clone方法，如果Client调用ConcretePrototype1的clone方法，将返回ConcretePrototype1的实例。

浅复制与深复制

浅复制：只复制了指针值，并没有复制指针指向的资源(即没有创建指针指向资源的副本)，复制后原有指针和新指针共享同一块内存。
深复制：不仅复制了指针值，还复制了指针指向的资源。

下面的示意图左边为浅复制，右边为深复制。



Cocoa Touch框架为NSObject的派生类提供了实现深复制的协议，即NSCopying协议，提供深复制的NSObject子类，需要实现NSCopying协议的方法(id)copyWithZone:(NSZone *)zone。NSObject有一个实例方法(id)copy，这个方法默认调用了[self copyWithZone:nil]，对于引用了NSCopying协议的子类，必须实现(id)copyWithZone:(NSZone *)zone方法，否则将引发异常，异常信息如下：

*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '-[Prototype copyWithZone:]: unrecognized selector sent to instance 0x100114d50'

原型模式的示例

新建Prototype类，Prototype.h如下：

```
1 @interface Prototype : NSObject<NSCopying>
2
3 //设置一个属性，用来检测复制的变化
4
5 @property(nonatomic, strong) NSString *name;
6
7 @end
```

实现深复制，Prototype.m文件如下：

```
#import "Prototype.h"

@implementation Prototype

- (id)init

{

    if (self = [superinit])

    {

        //初始化Prototype类时，将name设置如下

        self.name = @"My name is Prototype";

    }

    returnself;

}

//实现NSCopying中的方法

- (id)copyWithZone:(NSZone *)zone

{

    //调用allocWithZone方法，复制一个新对象

    return [[[selfclass] allocWithZone:zone] init];

}

@end
```

测试代码如下：

```
1          // 创建Prototype实例 prototype
2
3          Prototype *prototype = [[Prototypealloc] init];
4
5          // 通过prototype深复制出一个新的对象prototypeCopy
6
7          Prototype *prototypeDeepCopy = [prototype copy];
8
9          // 通过prototype直接赋值，其实就是复制了指针(可以理解为取了个别名)，属于浅复制，引用计数不
变
10
11         Prototype *prototypeShallowCopy = prototype;
12
13         // 打印
14
15         NSLog(@"修改前========");
16
17         NSLog(@"原始对象:%p,%@",prototype, prototype.name);
18
19         NSLog(@"浅复制对象:%p,%@",prototypeShallowCopy, prototypeShallowCopy.name);
20
21         NSLog(@"深复制对象:%p,%@",prototypeDeepCopy,prototypeDeepCopy.name);
22
23         prototype.name = @"My name is new Prototype";
24
25         // 打印
26
27         NSLog(@"修改后========");
28
29         NSLog(@"原始对象:%p,%@",prototype, prototype.name);
30
31         NSLog(@"浅复制对象:%p,%@",prototypeShallowCopy, prototypeShallowCopy.name);
32
33         NSLog(@"深复制对象:%p,%@",prototypeDeepCopy,prototypeDeepCopy.name);
```

输出结果如下（省略时间及项目名）：

修改前========

原始对象:0x1001143f0,My name is Prototype

浅复制对象:0x1001143f0,My name is Prototype

深复制对象:0x1001155a0,My name is Prototype

修改后========

原始对象:0x1001143f0,My name is new Prototype

浅复制对象:0x1001143f0,My name is new Prototype

深复制对象:0x1001155a0,My name is Prototype

【结论】：

我们使用copyWithZone:(NSZone *)zone方法实现了深复制，通过copy方法(该方法默认调用copyWithZone方法)复制得到prototypeDeepCopy，从输出结果来看，内存地址与prototype是不一样的，另外深复制得到prototypeDeepCopy后，修改prototype的name，对prototypeDeepCopy的name值没有影响，可判断为深复制；
使用直接赋值得到的prototypeShallowCopy，内存地址与prototype一样，只是简单的指针复制，另外从修改了prototype的name值同时也影响了prototypeShallowCopy的name值也可以看出，这种为浅复制。

【说明】：大家看完这个例子，可能感觉怎么和原型模式的结构图不太一样？实际上是一样的，这里的Prototype类相当于是结构图里面的ConcretePrototype，NSCopying相当于是结构图里面的Prototype。

下载源码 (http://pan.baidu.com/s/1GfEbZ)

assign、copy 和 retain

我们还是通过一个示例来说明这三者的区别，定义一个类，类里面只有三个属性，如下所示：

```
1 @interface Test : NSObject
2
3
4
5 @property (nonatomic, copy)NSMutableString *strName;
6
7 @property (nonatomic, assign)NSMutableString *strName1;
8
9 @property (nonatomic, retain)NSMutableString *strName2;
```

调用代码：

```
 1         Test *t = [[Testalloc] init];
 2
 3         NSMutableString *strTest = [[NSMutableStringalloc] initWithString:@"abc"];
 4
 5         NSLog(@"strTest retainCount:%ld strTest:%p %@",[strTest retainCount],strTest,s
trTest);
 6
 7         t.strName1 = strTest;  // assign
 8
 9         NSLog(@"after assign:  strTest retainCount:%ld t.strName1:%p %@ ",[strTest ret
ainCount],t.strName1,t.strName1);
10
11         t.strName = strTest;  // copy
12
13         NSLog(@"after copy:  strTest retainCount:%ld t.strName:%p %@ ",[strTest retain
Count],t.strName,t.strName);
14
15         t.strName2 = strTest;  // retain
16
17         NSLog(@"after retain:  strTest retainCount:%ld t.strName2:%p %@ ",[strTest ret
ainCount],t.strName2,t.strName2);
```

输出结果如下所示(省略时间及项目名)：

start: strTest retainCount:1 strTest:0x1001157f0 abc

after assign: strTest retainCount:1 t.strName1:0x1001157f0 abc

after copy: strTest retainCount:1 t.strName:0x100400460 abc

after retain: strTest retainCount:2 t.strName2:0x1001157f0 abc

首先，咱们分析一下这行代码：NSMutableString *strTest = [[NSMutableStringalloc] initWithString:@"abc"];这行代码实际上进行了两个操作：

在栈上分配一段内存用来存储strTest，比如地址为0xAAAA，内容为0x1001157f0；
在堆上分配一段内存用来存储@"abc"，地址为0x1001157f0，内容为abc。

现在，咱们针对刚才示例的输出结果来分别对assign、copy和retain进行说明：

assign：默认值，应用assign后，t.strName1和strTest具有相同的内容0x1001157f0，并且retainCount没有增加，可以理解t.strName1是strTest的别名；

copy：应用copy后，会在堆上重新分配一段内存来存储@"abc"，地址为0x100400460，同时也会在栈上分配一段内存用来存储t.strName，比如地址为0xBBBB，内容为0x100400460，这时strTest管理0x1001157f0这段内存；t.strName管理0x100400460这段内存。t.strName和strTest的retainCount均为1。

retain：应用retain后，可以看到retainCount增加了1，说明在栈上重新分配了一段内存来存储t.strName2，比如地址为0xCCCC，内容为0x1001157f0。此时，strTest和t.strName2共同管理0x1001157f0这段内存。

想必这样介绍完，大家对于这三个属性应该是了解的比较清楚了。这里再顺便说一下atomic和nonatomic，这两个属性用来决定编译器生成的getter和setter是否为原子操作。

atomic：默认值，提供多线程安全。在多线程环境下，原子操作是必要的，否则有可能引起错误的结果。加了atomic，setter函数在操作前会加锁。

nonatomic：禁用多线程的变量保护，提高性能。

atomic是OC中使用的一种线程保护技术，用来防止在写操作未完成的时候被另外一个线程读取，造成数据错误。但是这种机制是耗费系统资源的，所以如果没有使用多线程的通讯编程，那么nonatomic是一个非常好的选择。

【小思考】：将本示例中的所有NSMutableString替换成NSString后，结果是不一样的，大家可以试验一下，然后思考这是为什么？(答案在下一小节会有解说)

下载源码 (http://pan.baidu.com/s/1OiHa)

IOS中的深复制

像NSString、NSDictionary这些类，本身已经实现了copyWithZone:(NSZone *)zone方法，直接使用如[NSString copy]调用即可。在复制后得到的副本，又可以分为可变副本(mutable copy)和不可变副本(immutable copy)。通常在NSCopying协议规定的方法copyWithZone中返回不可变副本，在NSMutableCopying协议规定的方法mutableCopyWithZone中返回可变副本，然后调用copy和mutableCopy方法来得到相应的不可变和可变副本。

NSString类已经遵循NSCopying协议及NSMutableCopying协议，下面还是通过示例来进行测试。

示例一：

```
1         NSString *strSource = [NSStringstringWithFormat:@"I am %@",@"ligf"];
2
3         // 使用copy方法,strSource和strCopy内存地址一致,strSource引用计数加1
4
5         NSString *strCopy = [strSource copy];
6
7         NSLog(@"原始字符串:%p,%@",strSource,strSource);
8
9         NSLog(@"复制字符串:%p,%@",strCopy,strCopy);
```

输出结果：

原始字符串:0x1001156c0,I am ligf

复制字符串:0x1001156c0,I am ligf

【结论】：

由[strSource copy]得到的strCopy，两者内存地址一致，由于copy返回的是不可变副本，系统只生成一份内存资源，此时的copy只是浅复制，和retain作用一样。(上一小节小思考里面留下的问题就是这个原因)

示例二：

```
1         NSString *strSource = [NSStringstringWithFormat:@"I am %@",@"ligf"];
2
3         // 使用mutableCopy方法,strSource和strCopy内存地址不一致,两者的引用计数均为1
4
5         NSString *strCopy = [strSource mutableCopy];
6
7         NSLog(@"原始字符串:%p,%@",strSource,strSource);
8
9         NSLog(@"复制字符串:%p,%@",strCopy,strCopy);
```

输出结果：

原始字符串:0x1001156c0,I am ligf

复制字符串:0x100114fb0,I am ligf

【结论】：

由[strSource mutableCopy]得到的strCopy，两者内存地址不一致，由于mutableCopy返回的是可变副本，系统生成了新的内存资源，此时的mutableCopy是深复制。

【示例三】：

```
 1         NSMutableString *strSource = [NSMutableStringstringWithFormat:@"I am %@",@"lig
f"];
 2
 3         // NSMutableString使用copy方法,strSource和strCopy内存地址不一致,两者的引用计数均为1
 4
 5         NSMutableString *strCopy = [strSource copy];
 6
 7         NSLog(@"原始字符串:%p,%@",strSource,strSource);
 8
 9         NSLog(@"复制字符串:%p,%@",strCopy,strCopy);
10
11         [strCopy appendString:@"hello"];
```

输出结果：

原始字符串:0x100115470,I am ligf

复制字符串:0x100115690,I am ligf

*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: 'Attempt to mutate immutable object with appendString:'

【结论】：

由[strSource copy]得到的strCopy，两者内存地址不一致，即是copy对NSMutableString类型进行了深复制，当尝试修改strCopy里面的值时，发现报错了，无法修改，可以确定副本strCopy是不可变副本。

【总的结论】：

对于系统中已经实现的同时支持NSCopying协议和NSMutableCopying协议的NSString、NSDictionary等，copy总是返回不可变副本，mutableCopy总是返回可变副本。

何时用原型模式

需要创建的对象应独立于其类型与创建方式。
要实例化的类是在运行时决定的。
不想要与产品层次相对应的工厂层次。
不同类的实例间的差异仅是状态的若干组合。因此复制相应数量的原型比手工实例化更加方便。
类不容易创建，比如每个组件可以把其他组件作为子节点的组合对象。复制已有的组合

对象并对副本进行修改会更加容易。

以下两种特别常见的情形，我们会想到用原型模式：

有很多的相关的类，其行为略有不同，而且主要差异在于内部属性，如名称等；
需要使用组合（树）对象作为其他对象的基础，比如，使用组合对象作为组件来构建另一个组合对象。

查看原文>> (http://blog.csdn.net/dlm_211314/article/details/38556615)

1

看过本文的人也看了：

- iOS知识结构图 (http://lib.csdn.net/base/ios/structure)
- ios设计模式 (http://lib.csdn.net/article/ios/42118)
- 【我们都爱Paul Hegarty】斯坦福IOS8... (http://lib.csdn.net/article/ios/44749)
- 设计模式深入学习IOS版（9）工程依赖... (http://lib.csdn.net/article/ios/42128)
- iOS KVO & KVC (http://lib.csdn.net/article/ios/42123)
- IOS开发中常用的设计模式 (http://lib.csdn.net/article/ios/42111)

发表评论

输入评论内容

发表

0个评论