Object-C基础知识（分类知多少）

1.分类的概念：

Category是Objective-C 2.0之后添加的**语言特性**，Category的主要作用是为已经存在的类添加方法。

2.分类的形式：

```
@interface 待扩展的类（分类的名称）
end
@implementation 待扩展的名称（分类的名称）
@end
```

3.分类的作用

```
1.给已经存在的类添加方法(常用，方便给系统类添加方法)
2.把类的实现分开在几个不同文件中（大文件的拆分）
  2.1.减少单个文件的体积
  2.1.把不同功能组织到不同的category里
  2.2.多个开发者共同完成一个类
  2.3.按照需求加载想要的category
3.声明私有方法
4.模仿多继承（拓展）
5.把framework的私有方法分开（拓展）
```

4.分类的特点

```
1.运行时决议（跟扩展的一个显著的不同点）
  1.1 在编译阶段并没有把分类添加到宿主类中，宿主类没有对应分类方法
  1.2 在运行时，通过runtime，把分类中添加的内容真实的添加到对应的宿主上
2.可以为系统类添加分类（扩展无法实现）
3.分类添加的方法可以"覆盖"原类方法
4.同名分类方法谁能生效取决于编译顺序
5.名字相同的分类会引起编译报错
6.分类添加的新方法可以被子类继承
```

5.Category源码：

```
struct category_t {
    const char *name;//分类的名称
    classref_t cls;//分类所属的宿主类
    struct method_list_t *instanceMethods; // 实例方法列表
    struct method_list_t *classMethods; // 类方法列表
    struct protocol_list_t *protocols; // 协议列表
    struct property_list_t *instanceProperties; // 属性列表
    // Fields below this point are not always present on disk.
    struct property_list_t *_classProperties;

    method_list_t *methodsForMeta(bool isMeta) {
        if (isMeta) return classMethods;
        else return instanceMethods;
    }

    property_list_t *propertiesForMeta(bool isMeta) {
        if (isMeta) return nil;//classProperties;
        else return instanceProperties;
    }
};
//从源码基本可以看出我们平时使用categroy的方式，对象方法，类方法，协议，和属性都可以找到对应的存储方式。并且我们发现分类结构体中是不存在成员变量的，因此分类中是不允许添加成员变量的。分类中添加的属性并不会帮助我们自动生成成员变量，只会生成get set方法的声明，需要我们自己去实现
```

6.分类中都可以添加哪些内容

```
6.1 实例方法
6.2 类方法
6.3 协议
6.4 属性(只是声明了get方法和set方法，并没有在分类中添加了实例变量)
6.5 分类添加实例变量，是通过关联对象实现的(后面有例子)
```

7.分类的加载调用栈

---

category_t 结构体

```
struct _category_t {
    const char *name;
    struct _class_t *cls;
    const struct _method_list_t *instance_methods;
    const struct _method_list_t *class_methods;
    const struct _protocol_list_t *protocols;
    const struct _prop_list_t *properties;
};
```

结构体赋值

```
extern "C" __declspec(dllimport) struct _class_t OBJC_CLASS_$_Preson;

static struct _category_t _OBJC_$_CATEGORY_Preson_$_Test __attribute__ ((used, section ("__DATA,__objc_const"))) =
{
    "Preson",
    0, // &OBJC_CLASS_$_Preson,
    (const struct _method_list_t *)&_OBJC_$_CATEGORY_INSTANCE_METHODS_Preson_$_Test,
    (const struct _method_list_t *)&_OBJC_$_CATEGORY_CLASS_METHODS_Preson_$_Test,
    (const struct _protocol_list_t *)&_OBJC_CATEGORY_PROTOCOLS_$_Preson_$_Test,
    (const struct _prop_list_t *)&_OBJC_$_PROP_LIST_Preson_$_Test,
};
static void OBJC_CATEGORY_SETUP_$_Preson_$_Test(void ) {
    _OBJC_$_CATEGORY_Preson_$_Test.cls = &OBJC_CLASS_$_Preson;
}
```

分类源码中是将我们定义的对象方法，类方法，属性等都存放在catagory_t 结构体中
最后cls指针指向的是分类的主类类对象的地址

```
    // Discover categories.
    for (EACH_HEADER) {
        category_t **catlist =
            _getObjc2CategoryList(hi, &count);
        bool hasClassProperties = hi->info()->hasCategoryClassProperties();

        for (i = 0; i < count; i++) {
            category_t *cat = catlist[i];
            Class cls = remapClass(cat->cls);

            if (!cls) {
                // Category's target class is missing (probably weak-linked).
                // Disavow any knowledge of this category.
                catlist[i] = nil;
                if (PrintConnecting) {
                    _objc_inform("CLASS: IGNORING category \?\?\?(%s) %p with "
                                 "missing weak-linked target class",
                                 cat->name, cat);
                }
                continue;
            }

            // Process this category.
            // First, register the category with its target class.
            // Then, rebuild the class's method lists (etc) if
            // the class is realized.
            bool classExists = NO;
            if (cat->instanceMethods ||  cat->protocols
                ||  cat->instanceProperties)
            {
                addUnattachedCategoryForClass(cat, cls, hi);
                if (cls->isRealized()) {
                    remethodizeClass(cls);
                    classExists = YES;
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category -%s(%s) %s",
                                 cls->nameForLogging(), cat->name,
                                 classExists ? "on existing class" : "");
                }
            }

            if (cat->classMethods  ||  cat->protocols
                ||  (hasClassProperties && cat->_classProperties))
            {
                addUnattachedCategoryForClass(cat, cls->ISA(), hi);
                if (cls->ISA()->isRealized()) {
                    remethodizeClass(cls->ISA());
                }
                if (PrintConnecting) {
                    _objc_inform("CLASS: found category +%s(%s)",
                                 cls->nameForLogging(), cat->name);
                }
            }
        }
    }
```

从上述代码中我们可以知道这段代码是用来查找有没有分类的。通过_getObjc2CategoryList函数获取到分类列表之后，进行遍历，获取其中的方法，协议，属性等。可以看到最终都调用了remethodizeClass(cls)

---

remethodizeClass

```
/***********************************************************************
* remethodizeClass
* Attach outstanding categories to an existing class.
* Fixes up cls's method list, protocol list, and property list.
* Updates method caches for cls and its subclasses.
* Locking: runtimeLock must be held by the caller
**********************************************************************/
static void remethodizeClass(Class cls)
{
    category_list *cats;
    bool isMeta;

    runtimeLock.assertWriting();

    isMeta = cls->isMetaClass();

    // Re-methodizing: check for more categories
    if ((cats = unattachedCategoriesForClass(cls, false/*not realizing*/))) {
        if (PrintConnecting) {
            _objc_inform("CLASS: attaching categories to class '%s' %s",
                         cls->nameForLogging(), isMeta ? "(meta)" : "");
        }

        attachCategories(cls, cats, true /*flush caches*/);
        free(cats);
    }
}
```

通过上述代码我们发现attachCategories函数接收了类对象cls和分类数组cats，如我们一开始写的代码所示，一个类可以有多个分类。之前我们说到分类信息存储在category_t结构体中，那么多个分类则保存在category_list中

attachCategorires方法说明

```
// Attach method lists and properties and protocols from categories to a class.
// Assumes the categories in cats are all loaded and sorted by load order,
// oldest categories first.
static void
attachCategories(Class cls, category_list *cats, bool flush_caches)
{
    if (!cats) return;
    if (PrintReplacedMethods) printReplacements(cls, cats);

    bool isMeta = cls->isMetaClass();

    // fixme rearrange to remove these intermediate allocations
    method_list_t **mlists = (method_list_t **)
        malloc(cats->count * sizeof(*mlists));
    property_list_t **proplists = (property_list_t **)
        malloc(cats->count * sizeof(*proplists));
    protocol_list_t **protolists = (protocol_list_t **)
        malloc(cats->count * sizeof(*protolists));

    // Count backwards through cats to get newest categories first
    int mcount = 0;
    int propcount = 0;
    int protocount = 0;
    int i = cats->count;
    bool fromBundle = NO;
    while (i--) {
        auto& entry = cats->list[i];

        method_list_t *mlist = entry.cat->methodsForMeta(isMeta);
        if (mlist) {
            mlists[mcount++] = mlist;
            fromBundle |= entry.hi->isBundle();
        }

        property_list_t *proplist =
            entry.cat->propertiesForMeta(isMeta, entry.hi);
        if (proplist) {
            proplists[propcount++] = proplist;
        }

        protocol_list_t *protolist = entry.cat->protocols;
        if (protolist) {
            protolists[protocount++] = protolist;
        }
    }

    auto rw = cls->data();

    prepareMethodLists(cls, mlists, mcount, NO, fromBundle);
    rw->methods.attachLists(mlists, mcount);
    free(mlists);
    if (flush_caches  &&  mcount > 0) flushCaches(cls);

    rw->properties.attachLists(proplists, propcount);
    free(proplists);

    rw->protocols.attachLists(protolists, protocount);
    free(protolists);
}
```

根绝每个分类中方法列表，属性列表，协议列表分配内存

遍历拿到每一个分类

将所有分类中的所有方法存入mlist数组中

将所有分类中的所有属性存入proplists数组

将所有分类中所有协议存入protolists数组中

rw：class_rw_t结构体，class结构体中用来存储对象方法，属性，协议的结构体

将mlists数组传入rw->methods的attachLists函数，之后释放mlist

将proplists数组传入rw->properties的attachLists函数，之后释放prolists

将protolists传入rw->protocols的attachLists函数，之后释放protplists

上述源码中可以看出，首先根据方法列表，属性列表，协议列表，malloc分配内存，根据多少个分类以及每一块方法需要多少内存来分配相应的内存地址。之后从分类数组里面往三个数组里面存放分类数组里面存放的分类方法，属性以及协议放入对应mlist、proplists、protolosts数组中，这三个数组放着所有分类的方法，属性和协议。

之后通过类对象的data()方法，拿到类对象的class_rw_t结构体rw，class_rw_t中存放着类对象的方法，属性和协议等数据，rw结构体通过类对象的data方法获取，所以rw里面存放这类对象里面的数据。

之后分别通过rw调用方法列表、属性列表、协议列表的attachList函数，将所有的分类的方法、属性、协议列表数组传进去，大致可以猜想到在attachList方法内部将分类和本类相应的对象方法，属性，和协议进行了合并。

---

attachLists

```
void attachLists(List* const * addedLists, uint32_t addedCount) {
    if (addedCount == 0) return;

    if (hasArray()) {
        // many lists -> many lists
        uint32_t oldCount = array()->count;
        uint32_t newCount = oldCount + addedCount;
        setArray((array_t *)realloc(array(), array_t::byteSize(newCount)));
        array()->count = newCount;
        memmove(array()->lists + addedCount, array()->lists,
                oldCount * sizeof(array()->lists[0]));
        memcpy(array()->lists, addedLists,
               addedCount * sizeof(array()->lists[0]));
    }
    else if (!list  &&  addedCount == 1) {
        // 0 lists -> 1 list
        list = addedLists[0];
    }
    else {
        // 1 list -> many lists
        List* oldList = list;
        uint32_t oldCount = oldList ? 1 : 0;
        uint32_t newCount = oldCount + addedCount;
        setArray((array_t *)malloc(array_t::byteSize(newCount)));
        array()->count = newCount;
        if (oldList) array()->lists[addedCount] = oldList;
        memcpy(array()->lists, addedLists,
               addedCount * sizeof(array()->lists[0]));
    }
}
```

array()->lists：原来的列表数组

addedList：分类的列表数组

memmove：内存移动，将array()->lists的内存移动oldCount * sizeof(array()->lists[0])个内存到array()->lista + addedCound中

memcpy：内存复制，将addedLists的内存复制addedCount * sizeof(array()->lists[0])个内存到array()->lists中

array()->lists： 类对象原来的方法列表，属性列表，协议列表。
addedLists：传入所有分类的方法列表，属性列表，协议列表。
attachLists函数中最重要的两个方法为memmove内存移动和memcpy内存拷贝。我们先来分别看一下这两个函数

```
// memmove ： 内存移动。
/*  __dst ： 移动内存的目的地
*   __src ： 被移动的内存首地址
*   __len ： 被移动的内存长度
*   将__src的内存移动__len块内存到__dst中
*/
void    *memmove(void *__dst, const void *__src, size_t __len);

// memcpy ： 内存拷贝。
/*  __dst ： 拷贝内存的拷贝目的地
*   __src ： 被拷贝的内存首地址
*   __n ： 被移动的内存长度
*   将__src的内存移动__n块内存到__dst中
*/
void    *memcpy(void *__dst, const void *__src, size_t __n);
```

```
//通过上面的源码我们发现，分类的方法，协议，属性等确实是存放在Categroy结构体里面的，下面简单说明加载过程
1._objc_init(runtime 初始化)
  2.map_2_images (images: 镜像)
    3.map_images_nolock
      4._read_images (加载可执行文件到内存中)
        5.remethodizeClass(分类的加载的开始 )
          5.1 获取cls中的所有分类分类列表(cats)
          5.2 将分类cats拼接到cls上
          5.3 method_list_t,property_list_t,protocol_list_t
          //每个list就是一个二维数组，类似这样的结构：[[method_t,...],[method_t],[method_t,...]]
          //内部每个小的数组就是一个分类，每个method_t就是一个分类的中方法
          //获取宿主类中分类的总数cats->count
          i := cats->count
          while(i--) {
            //cats中最后一个元素是最后参加编译的
            //倒序遍历，最先访问最后编译的分类
            //如果两个分类有同名的方法，那个方法会最终生效，取决于那个分类最后被编译
            //最后编译的分类最先添加到分类的数组中，最后编译的分类方法会最终生效
            //获取宿主类中的方法列表,把mcount个方法的mlists二维数组列表添加到method_list总（分类是运行时决议）
            rw->method.attachList(mlists,mcount)
            //attachList实现
            1.获取列表中原有元素的总数 oldCount = 2
            2.计算拼接之后的总数 newCount = 5
            3.根据总数重新分配内存
            4.重新设置元素总数
            5.内存移动:[[],[],[],[origin1],[origin2]]
            6.内容copy:
            [
              A--->[addedLists中的第一个元素]
              B--->[addedLists中的第二个元素]
              C--->[addedLists中的第二个元素]
              [origin1]
              [origin2]
            ]
            //这个就是分类方法会"覆盖"宿主类方法的原因，宿主类方法实际上仍然存在，但是在消息方法查找，是根据选择器名称来查找的，一旦查找到对应的实现方法就会返回，如果分类中有跟宿主类同名的方法，分类的方法会被优先实现
          }
          //分类中的对象方法依然是存储在类对象中的，同本类对象方法在同一个地方，调用步骤也同调用对象方法一样。如果是类方法的话，也同样是存储在元类对象中
```

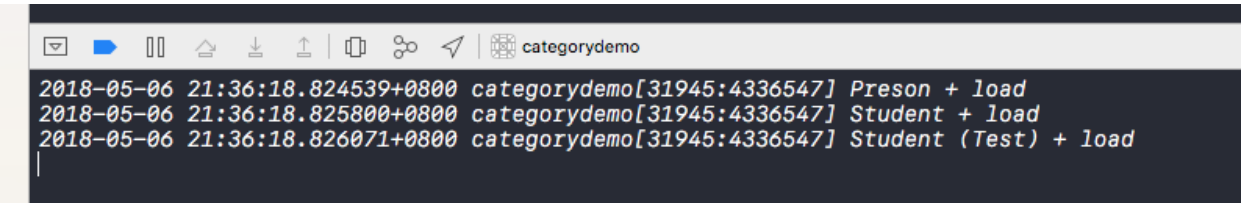
8.load 和 initialize

load方法会在程序启动就会调用，当装载类信息的时候就会调用。
调用顺序看一下源代码。

```
void call_load_methods(void)
{
    static bool loading = NO;
    bool more_categories;

    loadMethodLock.assertLocked();

    // Re-entrant calls do nothing; the outermost call will finish the job.
    if (loading) return;
    loading = YES;

    void *pool = objc_autoreleasePoolPush();

    do {
        // 1. Repeatedly call class +loads until there aren't any more
        while (loadable_classes_used > 0) {
            call_class_loads();
        }

        // 2. Call category +loads ONCE
        more_categories = call_category_loads();

        // 3. Run more +loads if there are classes OR more untried categories
    } while (loadable_classes_used > 0  ||  more_categories);

    objc_autoreleasePoolPop(pool);

    loading = NO;
}
```

先调用类的load方法

之后再调用分类的load方法




```
    // Call all +loads for the detached list.
    for (i = 0; i < used; i++) {
        Category cat = cats[i].cat;
        load_method_t load_method = (load_method_t)cats[i].method;
        Class cls;
        if (!cat) continue;

        cls = _category_getClass(cat);
        if (cls  &&  cls->isLoadable()) {
            if (PrintLoading) {
                _objc_inform("LOAD: +[%s(%s) load]\n",
                             cls->nameForLogging(),
                             _category_getName(cat));
            }
            (*load_method)(cls, SEL_load);
            cats[i].cat = nil;
        }
    }
```

可以看到分类中也是通过直接拿到load方法的地址进行调用。因此正如我们之前试验的一样，分类中重写load方法，并不会优先调用分类的load方法，而不调用本类中的load方法了

为Preson、Student 、Student+Test 添加initialize方法。
我们知道当类第一次接收到消息时，就会调用initialize，相当于第一次使用类的时候就会调用initialize方法。调用子类的initialize之前，会先保证调用父类的initialize方法。如果之前已经调用过initialize，就不会再调用initialize方法了。当分类重写initialize方法时会先调用分类的方法。但是load方法并不会被覆盖，首先我们来看一下initialize的源码

```
void callInitialize(Class cls)
{
    ((void(*)(Class, SEL))objc_msgSend)(cls, SEL_initialize);
    asm("");
}
```

上图中说明，initialize是通过消息发送机制调用的，消息发送机制通过isa指针找到对应的方法与实现，因此先找到分类方法中的实现，会优先调用分类方法中的实现。

9.思考

1.Category的实现原理，以及Category为什么只能加方法不能加属性。

> 分类的实现原理是将category中的方法，属性，协议数据放在category_t 结构体中，然后将结构体内的方法列表拷贝到类对象的方法列表中。
> Category可以添加属性，但是并不会自动生成成员变量及set/get方法。因为category_t 结构体中并不存在成员变量。通过之前对对象的分析我们知道成员变量是存放在实例对象中的，并且编译的那一刻就已经决定好了。而分类是在运行时才去加载的。那么我们就无法再程序运行时将分类的成员变量中添加到实例对象的结构体中。因此分类中不可以添加成员变量。

2.Category中有load方法吗？load方法是什么时候调用的？load 方法能继承吗？

> Category中有load方法，load方法在程序启动装载类信息的时候就会调用。load方法可以继承。调用子类的load方法之前，会先调用父类的load方法

3.load、initialize的区别，以及它们在category重写的时候的调用的次序。

> 区别在于调用方式和调用时刻
> 调用方式：load是根据函数地址直接调用，initialize是通过objc_msgSend调用
> 调用时刻：load是runtime加载类、分类的时候调用（只会调用1次），initialize是类第一次接收到消息的时候调用，每一个类只会initialize一次（父类的initialize方法可能会被调用多次）
>
> 调用顺序：先调用类的load方法，先编译那个类，就先调用load。在调用load之前会先调用父类的load方法。分类中load方法不会覆盖本类的load方法，先编译的分类优先调用load方法。initialize先初始化父类，之后再初始化子类。如果子类没有实现+initialize，会调用父类的+initialize（所以父类的+initialize可能会被调用多次），如果分类实现了+initialize，就覆盖类本身的+initialize调用。

10.虽然不能在分类（类别）中定义成员变量，但是有办法也可以让它支持添加成员变量

```
    一种常见的办法是通过runtime.h中objc_getAssociatedObject / objc_setAssociatedObject来访问和生成关联对象。通过这种方法来模拟生成变量。
```

“NSObject+SpecialName.h”文件：

```
@interface NSObject (SpecialName)
@property (nonatomic, copy) NSString *specialName;
@end
```

“NSObject+SpecialName.m”文件：

```
#import "NSObject+Extension.h"
#import <objc/runtime.h>

static const void *SpecialNameKey = &SpecialNameKey;
@implementation NSObject (SpecialName)
@dynamic specialName;

- (NSString *)specialName {
    //如果属性值是非id类型，可以通过属性值先构造OC的id对象，再通过对象获取非id类型属性
    return objc_getAssociatedObject(self, SpecialNameKey);
}

- (void)setSpecialName:(NSString *)specialName{
    //如果属性值是非id类型，可以通过属性值先构造OC的id对象，再通过对象获取非id类型属性
    objc_setAssociatedObject(self, SpecialNameKey, specialName, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
@end
```