iOS 开发：『Runtime』详解（三）Category 底层原理

- 本文首发于我的个人博客：『不羁阁』
- 文章链接：传送门
- 本文更新时间：2019年07月24日20:15:36

本文用来介绍 iOS 开发中『Runtime』中的 **Category 底层原理**。通过本文，您将了解到：

1. Category （分类）简介
2. Category 的实质
3. Category 的加载过程
4. Category（分类）和 Class（类）的 +load 方法
5. Category 与关联对象

文中示例代码在： bujige / **YSC-Category-Demo**

1. Category （分类）简介

1.1 什么是 Category（分类）？

Category（分类） 是 Objective-C 2.0 添加的语言特性，主要作用是为已经存在的类添加方法。Category 可以做到在既不子类化，也不侵入一个类的源码的情况下，为原有的类添加新的方法，从而实现扩展一个类或者分离一个类的目的。在日常开发中我们常常使用 Category 为已有的类扩展功能。

虽然继承也能为已有类增加新的方法，而且还能直接增加属性，但继承关系增加了不必要的代码复杂度，在运行时，也无法与父类的原始方法进行区分。所以我们可以优先考虑使用自定义 Category（分类）。

通常 Category（分类）有以下几种使用场景：

- 把类的不同实现方法分开到不同的文件里。
- 声明私有方法。
- 模拟多继承。
- 将 framework 私有方法公开化。

1.2 Category（分类）和 Extension（扩展）

Category（分类）看起来和 Extension（扩展）有点相似。Extension（扩展）有时候也被称为 **匿名分类**。但两者实质上是不同的东西。 Extension（扩展）是在编译阶段与该类同时编译的，是类的一部分。而且 Extension（扩展）中声明的方法只能在该类的 `@implementation` 中实现，这也就意味着，你无法对系统的类（例如 NSString 类）使用 Extension（扩展）。

而且和 Category（分类）不同的是，Extension（扩展）不但可以声明方法，还可以声明成员变量，这是 Category（分类）所做不到的。

> 为什么 Category（分类）不能像 Extension（扩展）一样添加成员变量?
>
> 因为 Extension（扩展）是在编译阶段与该类同时编译的，就是类的一部分。既然作为类的一部分，且与类同时编译，那么就可以在编译阶段为类添加成员变量。
>
> 而 Category（分类）则不同， Category（分类）的特性是：**可以在运行时阶段动态地为已有类添加新行为**。 Category（分类）是在运行时期间决定的。而成员变量的内存布局

> 已经在编译阶段确定好了，如果在运行时阶段添加成员变量的话，就会破坏原有类的内存布局，从而造成可怕的后果，所以 Category（分类）无法添加成员变量。

---

2. Category 的实质

2.1 Category 结构体简介

在第一篇 iOS 开发：『Runtime』详解（一）基础知识 中我们知道了：`Object（对象）` 和 `Class（类）` 的实质分别是 `objc_object 结构体` 和 `objc_class 结构体`，这里 Category 也不例外，在 `objc-runtime-new.h` 中，Category（分类）被定义为 `category_t 结构体`。`category_t 结构体` 的数据结构如下：

```C++
typedef struct category_t *Category;

struct category_t {
    const char *name;                                   // 类名
    classref_t cls;                                     // 类，在运行时阶段通过 clasee_name（类
    struct method_list_t *instanceMethods;              // Category 中所有添加的对象方法列表
    struct method_list_t *classMethods;                 // Category 中所有添加的类方法列表
    struct protocol_list_t *protocols;                  // Category 中实现的所有协议列表
    struct property_list_t *instanceProperties;         // Category 中添加的所有属性
};
```

从 Category（分类）的结构体定义中也可以看出， Category（分类）可以为类添加对象方法、类方法、协议、属性。同时，也能发现 Category（分类）无法添加成员变量。

2.2 Category 的 C++ 源码

想要了解 Category 的本质，我们需要借助于 Category 的 C++ 源码。 首先呢，我们需要写一个继承自 NSObject 的 Person 类，还需要写一个 Person+Additon 的分类。在分类中添加对象方法，类方法，属性，以及代理。

例如下边代码中这样：

Objc

```
/********************** Person+Addition.h 文件 **********************/

#import "Person.h"

// PersonProtocol 代理
@protocol PersonProtocol <NSObject>

- (void)PersonProtocolMethod;

+ (void)PersonProtocolClassMethod;

@end

@interface Person (Addition) <PersonProtocol>

/* name 属性 */
@property (nonatomic, copy) NSString *personName;

// 类方法
+ (void)printClassName;

// 对象方法
- (void)printName;

@end

/********************** Person+Addition.m 文件 **********************/

#import "Person+Addition.h"

@implementation Person (Addition)

+ (void)printClassName {
    NSLog(@"printClassName");
}

- (void)printName {
    NSLog(@"printName");
}

#pragma mark - <PersonProtocol> 方法

- (void)PersonProtocolMethod {
    NSLog(@"PersonProtocolMethod");
}
```

```
+ (void)PersonProtocolClassMethod {
    NSLog(@"PersonProtocolClassMethod");
}
```

> Category 由 OC 转 C++ 源码方法如下：
>
> 1. 在项目中添加 Person 类文件 Person.h 和 Person.m，Person 类继承自 NSObject 。
> 2. 在项目中添加 Person 类的 Category 文件 Person+Addition.h 和 Person+Addition.m，并在 Category 中添加的相关对象方法，类方法，属性，以及代理。
> 3. 打开『终端』，执行 `cd XXX/XXX` 命令，其中 `XXX/XXX` 为 Category 文件 所在的目录。
> 4. 继续在终端执行 `clang -rewrite-objc Person+Addition.m`
> 5. 执行完命令之后，Person+Addition.m 所在目录下就会生成一个 Person+Addition.cpp 文件，这就是我们需要的 Category（分类） 相关的 C++ 源码。

当我们得到 Person+Addition.cpp 文件之后，就会神奇的发现：这是一个 3.7M 大小，拥有近 10W 行代码的庞大文件。

不用慌。Category 的相关 C++ 源码在文件的最底部。我们删除其他无关代码，只保留 Category 有关的代码，大概就会剩下差不多 200 多行代码。下边我们根据 Category 结构体 的不同结构，分模块来讲解一下。

2.2.1 『Category 结构体』

C++

```
// Person 类的 Category 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;
};

// Person 类的 Category 结构体赋值
static struct _category_t _OBJC_$_CATEGORY_Person_$_Addition __attribute__ ((used, secti
{
        "Person",
        0, // &OBJC_CLASS_$_Person,
        (const struct _method_list_t *)&_OBJC_$_CATEGORY_INSTANCE_METHODS_Person_$_Addit
        (const struct _method_list_t *)&_OBJC_$_CATEGORY_CLASS_METHODS_Person_$_Addition
        (const struct _protocol_list_t *)&_OBJC_CATEGORY_PROTOCOLS_$_Person_$_Addition,
        (const struct _prop_list_t *)&_OBJC_$_PROP_LIST_Person_$_Addition,
};

// Category 数组，如果 Person 有多个分类，则 Category 数组中对应多个 Category
static struct _category_t *L_OBJC_LABEL_CATEGORY_$ [1] __attribute__((used, section ("__
        &_OBJC_$_CATEGORY_Person_$_Addition,
};
```

从『Category 结构体』源码中我们可以看到：

1. Categor 结构体。
2. Category 结构体的赋值语句。
3. Category 结构体数组。

第一个 Categor 结构体和 **2.1 Category 结构体简介** 中的结构体其实质是一一对应的。可以看做是同一个结构体。第三个 Category 结构体数组中存放了 Person 类的相关分类，如果有多个分类，则数组中存放对应数目的 Category 结构体。

2.2.2 Category 中『对象方法列表结构体』

C++

```
// - (void)printName; 对象方法的实现
static void _I_Person_Addition_printName(Person * self, SEL _cmd) {
    NSLog((NSString *)&__NSConstantStringImpl__var_folders_ct_0dyw1pvj6k16t5z8t0j0_ghw00
}

// - (void)personProtocolMethod; 方法的实现
static void _I_Person_Addition_personProtocolMethod(Person * self, SEL _cmd) {
    NSLog((NSString *)&__NSConstantStringImpl__var_folders_ct_0dyw1pvj6k16t5z8t0j0_ghw00
```

```
}

// Person 分类中添加的『对象方法列表结构体』
static struct /*_method_list_t*/ {
        unsigned int entsize;  // sizeof(struct _objc_method)
        unsigned int method_count;
        struct _objc_method method_list[2];
} _OBJC_$_CATEGORY_INSTANCE_METHODS_Person_$_Addition __attribute__ ((used, section ("__
        sizeof(_objc_method),
        2,
        {{(struct objc_selector *)"printName", "v16@0:8", (void *)_I_Person_Addition_pri
        {(struct objc_selector *)"personProtocolMethod", "v16@0:8", (void *)_I_Person_Ad
};
```

> 从『对象方法列表结构体』源码中我们可以看到：
>
> 1. `- (void)printName;` 对象方法的实现。
> 2. `- (void)personProtocolMethod;` 方法的实现。
> 3. 对象方法列表结构体。
>
> 只要是 Category 中 **实现了** 的对象方法（包括代理中的对象方法）。都会添加到 **对象方法列表结构体** `_OBJC_$_CATEGORY_INSTANCE_METHODS_Person_$_Addition` 中来。如果只是在 Person.h 中定义，而没有实现，则不会添加。

2.2.3 Category 中『类方法列表结构体』

C++

```
// + (void)printClassName; 类方法的实现
static void _C_Person_Addition_printClassName(Class self, SEL _cmd) {
    NSLog((NSString *)&__NSConstantStringImpl__var_folders_ct_0dyw1pvj6k16t5z8t0j0_ghw00
}

// + (void)personProtocolClassMethod; 方法的实现
static void _C_Person_Addition_personProtocolClassMethod(Class self, SEL _cmd) {
    NSLog((NSString *)&__NSConstantStringImpl__var_folders_ct_0dyw1pvj6k16t5z8t0j0_ghw00
}

// Person 分类中添加的『类方法列表结构体』
static struct /*_method_list_t*/ {
        unsigned int entsize;  // sizeof(struct _objc_method)
        unsigned int method_count;
        struct _objc_method method_list[2];
} _OBJC_$_CATEGORY_CLASS_METHODS_Person_$_Addition __attribute__ ((used, section ("__DAT
```

```
        sizeof(_objc_method),
        2,
        {{(struct objc_selector *)"printClassName", "v16@0:8", (void *)_C_Person_Additio
        {(struct objc_selector *)"personProtocolClassMethod", "v16@0:8", (void *)_C_Pers
};
```

> 从『类方法列表结构体』源码中我们可以看到：
>
> 1. `+ (void)printClassName;` 类方法的实现。
> 2. `+ (void)personProtocolClassMethod;` 类方法的实现。
> 3. 类方法列表结构体。
>
> 只要是 Category 中 **实现了** 的类方法（包括代理中的类方法）。都会添加到 **类方法列表结构体** `_OBJC_$_CATEGORY_CLASS_METHODS_Person_$_Addition` 中来。如果只是在 Person.h 中定义，而没有实现，则不会添加。

2.2.4 Category 中『协议列表结构体』

C++

```
// Person 分类中添加的『协议列表结构体』
static struct /*_protocol_list_t*/ {
        long protocol_count;  // Note, this is 32/64 bit
        struct _protocol_t *super_protocols[1];
} _OBJC_CATEGORY_PROTOCOLS_$_Person_$_Addition __attribute__ ((used, section ("__DATA,__
        1,
        &_OBJC_PROTOCOL_PersonProtocol
};

// 协议列表 对象方法列表结构体
static struct /*_method_list_t*/ {
        unsigned int entsize;  // sizeof(struct _objc_method)
        unsigned int method_count;
        struct _objc_method method_list[1];
} _OBJC_PROTOCOL_INSTANCE_METHODS_PersonProtocol __attribute__ ((used, section ("__DATA,
        sizeof(_objc_method),
        1,
        {{(struct objc_selector *)"personProtocolMethod", "v16@0:8", 0}}
};

// 协议列表 类方法列表结构体
static struct /*_method_list_t*/ {
        unsigned int entsize;  // sizeof(struct _objc_method)
        unsigned int method_count;
```

```
        struct _objc_method method_list[1];
} _OBJC_PROTOCOL_CLASS_METHODS_PersonProtocol __attribute__ ((used, section ("__DATA,__o
        sizeof(_objc_method),
        1,
        {{(struct objc_selector *)"personProtocolClassMethod", "v16@0:8", 0}}
};

// PersonProtocol 结构体赋值
struct _protocol_t _OBJC_PROTOCOL_PersonProtocol __attribute__ ((used)) = {
        0,
        "PersonProtocol",
        (const struct _protocol_list_t *)&_OBJC_PROTOCOL_REFS_PersonProtocol,
        (const struct method_list_t *)&_OBJC_PROTOCOL_INSTANCE_METHODS_PersonProtocol,
        (const struct method_list_t *)&_OBJC_PROTOCOL_CLASS_METHODS_PersonProtocol,
        0,
        0,
        0,
        sizeof(_protocol_t),
        0,
        (const char **)&_OBJC_PROTOCOL_METHOD_TYPES_PersonProtocol
};
struct _protocol_t *_OBJC_LABEL_PROTOCOL_$_PersonProtocol = &_OBJC_PROTOCOL_PersonProtoc
```

> 从『协议列表结构体』源码中我们可以看到：
>
> 1. 协议列表结构体。
> 2. 协议列表 对象方法列表结构体。
> 3. 协议列表 类方法列表结构体。
> 4. PersonProtocol 协议结构体赋值语句。

2.2.5 Category 中『属性列表结构体』

C++

```
// Person 分类中添加的属性列表
static struct /*_prop_list_t*/ {
        unsigned int entsize;  // sizeof(struct _prop_t)
        unsigned int count_of_properties;
        struct _prop_t prop_list[1];
} _OBJC_$_PROP_LIST_Person_$_Addition __attribute__ ((used, section ("__DATA,__objc_cons
        sizeof(_prop_t),
        1,
        {{"personName","T@\"NSString\",C,N"}}
};
```

> 从『属性列表结构体』源码中我们看到：
>
> 只有 Person 分类中添加的 **属性列表结构体** `_OBJC_$_PROP_LIST_Person_$_Addition`，没有成员变量结构体 `_ivar_list_t 结构体` 。更没有对应的 `set 方法 / get 方法` 相关的内容。这也直接说明了 Category 中不能添加成员变量这一事实。

2.3 Category 的实质总结

> 下面我们来总结一下 **Category 的本质**：
>
> Category 的本质就是 `_category_t 结构体` 类型，其中包含了以下几部分：
>
> 1. `_method_list_t` 类型的『对象方法列表结构体』；
> 2. `_method_list_t` 类型的『类方法列表结构体』；
> 3. `_protocol_list_t` 类型的『协议列表结构体』；
> 4. `_prop_list_t` 类型的『属性列表结构体』。
>
> `_category_t 结构体` 中不包含 `_ivar_list_t` 类型，也就是不包含『成员变量结构体』。

---

3. Category 的加载过程

3.1 dyld 加载大致流程

之前我们谈到过 Category（分类）是在运行时阶段动态加载的。而 Runtime（运行时） 加载的过程，离不开一个叫做 dyld 的动态链接器。

在 MacOS 和 iOS 上，动态链接加载器 dyld 用来加载所有的库和可执行文件。而加载 Runtime（运行时） 的过程，就是在 dyld 加载的时候发生的。

dyld 的相关代码可在苹果开源网站上进行下载。 链接地址：dyld 苹果开源代码

dyld 加载的流程大致是这样：

1. 配置环境变量；
2. 加载共享缓存；
3. 初始化主 APP；
4. 插入动态缓存库；
5. 链接主程序；
6. 链接插入的动态库；
7. 初始化主程序：OC, C++ 全局变量初始化；
8. 返回主程序入口函数。

本文中，我们只需要关心的是第 7 步，因为 Runtime（运行时） 是在这一步初始化的。加载 Category（分类）自然也是在这个过程中。

> 初始化主程序中，Runtime 初始化的调用栈如下：
>
> `dyldbootstrap::start` ---> `dyld::_main` ---> `initializeMainExecutable` ---> `runInitializers` ---> `recursiveInitialization` ---> `doInitialization` ---> `doModInitFunctions` ---> `_objc_init`

最后调用的 `_objc_init` 是 `libobjc` 库中的方法， 是 Runtime 的初始化过程，也是 Objective-C 的入口。

运行时相关的代码可在苹果开源网站上进行下载。 链接地址： objc4 苹果开源代码

在 `_objc_init` 这一步中： `Runtime` 向 `dyld` 绑定了回调，当 `image` 加载到内存后， `dyld` 会通知 `Runtime` 进行处理， `Runtime` 接手后调用 `map_images` 做解析和处理，调用 `_read_images` 方法把 `Category（分类）` 的对象方法、协议、属性添加到类上，把 `Category（分类）` 的类方法、协议添加到类的 `metaclass` 上；接下来 `load_images` 中调用 `call_load_methods` 方法，遍历所有加载进来的 `Class`，按继承层级和编译顺序依次调用 `Class` 的 `load` 方法和其 `Category` 的 `load` 方法。

> 加载Category（分类）的调用栈如下：
>
> `_objc_init` ---> `map_images` ---> `map_images_nolock` ---> `_read_images（加载分类）` ---> `load_images` 。

既然我们知道了 Category（分类）的加载发生在 `_read_images` 方法中，那么我们只需要关注

`_read_images` 方法中关于分类加载的代码即可。

3.2 Category（分类） 加载过程

3.2.1 `_read_images` 方法

忽略 `_read_images` 方法中其他与本文无关的代码，得到如下代码：

```C++
// 获取镜像中的分类数组
category_t **catlist =
    _getObjc2CategoryList(hi, &count);
bool hasClassProperties = hi->info()->hasCategoryClassProperties();

// 遍历分类数组
for (i = 0; i < count; i++) {
    category_t *cat = catlist[i];
    Class cls = remapClass(cat->cls);
    // 处理这个分类
    // 首先，使用目标类注册当前分类
    // 然后，如果实现了这个类，重建类的方法列表
    bool classExists = NO;
    if (cat->instanceMethods ||  cat->protocols
        ||  cat->instanceProperties)
    {
        addUnattachedCategoryForClass(cat, cls, hi);
        if (cls->isRealized()) {
            remethodizeClass(cls);
            classExists = YES;
        }
    }

    if (cat->classMethods  ||  cat->protocols
        ||  (hasClassProperties && cat->_classProperties))
    {
        addUnattachedCategoryForClass(cat, cls->ISA(), hi);
        if (cls->ISA()->isRealized()) {
            remethodizeClass(cls->ISA());
        }
    }
}
```

主要用到了两个方法：

- `addUnattachedCategoryForClass(cat, cls, hi);` 为类添加未依附的分类
- `remethodizeClass(cls);` 重建类的方法列表

通过这两个方法达到了两个目的：

1. 把 `Category（分类）` 的对象方法、协议、属性添加到类上；
2. 把 `Category（分类）` 的类方法、协议添加到类的 `metaclass` 上。

下面来说说上边提到的这两个方法。

3.2.2 `addUnattachedCategoryForClass(cat, cls, hi);` 方法

```C++
static void addUnattachedCategoryForClass(category_t *cat, Class cls,
                                          header_info *catHeader)
{
    runtimeLock.assertLocked();

    // 取得存储所有未依附分类的列表：cats
    NXMapTable *cats = unattachedCategories();
    category_list *list;
    // 从 cats 列表中找到 cls 对应的未依附分类的列表：list
    list = (category_list *)NXMapGet(cats, cls);
    if (!list) {
        list = (category_list *)
            calloc(sizeof(*list) + sizeof(list->list[0]), 1);
    } else {
        list = (category_list *)
            realloc(list, sizeof(*list) + sizeof(list->list[0]) * (list->count + 1));
    }
    // 将新增的分类 cat 添加 list 中
    list->list[list->count++] = (locstamped_category_t){cat, catHeader};
    // 将新生成的 list 添加重新插入 cats 中，会覆盖旧的 list
    NXMapInsert(cats, cls, list);
}
```

`addUnattachedCategoryForClass(cat, cls, hi);` 的执行过程可以参考代码注释。执行完这个方法之后，系统会将当前分类 `cat` 放到该类 `cls` 对应的未依附分类的列表 `list` 中。这句话有点拗口，简而言之，就是：**把类和分类做了一个关联映射。**

实际上真正起到添加加载作用的是下边的 `remethodizeClass(cls);` 方法。

3.2.3 `remethodizeClass(cls);` 方法

```C++
static void remethodizeClass(Class cls)
{
    category_list *cats;
    bool isMeta;

    runtimeLock.assertLocked();

    isMeta = cls->isMetaClass();

    // 取得 cls 类的未依附分类的列表：cats
    if ((cats = unattachedCategoriesForClass(cls, false/*not realizing*/))) {
        // 将未依附分类的列表 cats 附加到 cls 类上
        attachCategories(cls, cats, true /*flush caches*/);
        free(cats);
    }
}
```

`remethodizeClass(cls);` 方法主要就做了一件事：调用 `attachCategories(cls, cats, true);` 方法将未依附分类的列表 cats 附加到 cls 类上。所以，我们就再来看看 `attachCategories(cls, cats, true);` 方法。

3.2.4 `attachCategories(cls, cats, true);` 方法

我发誓这是本文中加载 Category（分类）过程的最后一段代码。不过也是最为核心的一段代码。

```C++
static void
attachCategories(Class cls, category_list *cats, bool flush_caches)
{
    if (!cats) return;
    if (PrintReplacedMethods) printReplacements(cls, cats);

    bool isMeta = cls->isMetaClass();

    // 创建方法列表、属性列表、协议列表，用来存储分类的方法、属性、协议
    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;            // 记录方法的数量
int propcount = 0;         // 记录属性的数量
int protocount = 0;        // 记录协议的数量
int i = cats->count;       // 从分类数组最后开始遍历，保证先取的是最新的分类
bool fromBundle = NO;      // 记录是否是从 bundle 中取的
while (i--) { // 从后往前依次遍历
    auto& entry = cats->list[i];  // 取出当前分类

    // 取出分类中的方法列表。如果是元类，取得的是类方法列表；否则取得的是对象方法列表
    method_list_t *mlist = entry.cat->methodsForMeta(isMeta);
    if (mlist) {
        mlists[mcount++] = mlist;            // 将方法列表放入 mlists 方法列表数组中
        fromBundle |= entry.hi->isBundle();  // 分类的头部信息中存储了是否是 bundle，将其
    }

    // 取出分类中的属性列表，如果是元类，取得的是 nil
    property_list_t *proplist =
        entry.cat->propertiesForMeta(isMeta, entry.hi);
    if (proplist) {
        proplists[propcount++] = proplist;
    }

    // 取出分类中遵循的协议列表
    protocol_list_t *protolist = entry.cat->protocols;
    if (protolist) {
        protolists[protocount++] = protolist;
    }
}

// 取出当前类 cls 的 class_rw_t 数据
auto rw = cls->data();

// 存储方法、属性、协议数组到 rw 中
// 准备方法列表 mlists 中的方法
prepareMethodLists(cls, mlists, mcount, NO, fromBundle);
// 将新方法列表添加到 rw 中的方法列表中
rw->methods.attachLists(mlists, mcount);
// 释放方法列表 mlists
free(mlists);
// 清除 cls 的缓存列表
if (flush_caches  &&  mcount > 0) flushCaches(cls);

// 将新属性列表添加到 rw 中的属性列表中
rw->properties.attachLists(proplists, propcount);
```

```
    // 释放属性列表
    free(proplists);

    // 将新协议列表添加到 rw 中的协议列表中
    rw->protocols.attachLists(protolists, protocount);
    // 释放协议列表
    free(protolists);
}
```

从 `attachCategories(cls, cats, true);` 方法的注释中可以看出这个方法就是存储分类的方法、属性、协议的核心代码。

> 但是需要注意一些细节问题：
>
> - Category（分类）的方法、属性、协议只是添加到原有类上，并没有将原有类的方法、属性、协议进行完全替换。 举个例子说明就是：假设原有类拥有 `MethodA` 方法，分类也拥有 `MethodA` 方法，那么加载完分类之后，类的方法列表中会拥有两个 `MethodA` 方法。
> - Category（分类）的方法、属性、协议会被添加到原有类的方法列表、属性列表、协议列表的最前面，而原有类的方法、属性、协议则被移动到了列表后面。 因为在运行时查找方法的时候是顺着方法列表的顺序依次查找的，所以 Category（分类）的方法会先被搜索到，然后直接执行，而原有类的方法则不被执行。这也是 Category（分类）中的方法会覆盖掉原有类的方法的最直接原因。

---

4. Category（分类）和 Class（类）的 +load 方法

Category（分类）中的的方法、属性、协议附加到类上的操作，是在 `+ load` 方法执行之前进行的。也就是说，在 `+ load` 方法执行之前，类中就已经加载了 Category（分类）中的的方法、属性、协议。

而 Category（分类）和 Class（类）的 `+ load` 方法的调用顺序规则如下所示：

1. 先调用主类，按照编译顺序，顺序地根据继承关系由父类向子类调用；
2. 调用完主类，再调用分类，按照编译顺序，依次调用；ıíí
3. `+ load` 方法除非主动调用，否则只会调用一次。

通过这样的调用规则，我们可以知道：主类的 `+ load` 方法调用一定在分类 `+ load` 方法调用

之前。但是分类 `+ load` 方法调用顺序并不不是按照继承关系调用的，而是依照编译顺序确定的，这也导致了 `+ load` 方法的调用顺序并不一定确定。一个顺序可能是：`父类 -> 子类 -> 父类类别 -> 子类类别`，也可能是 `父类 -> 子类 -> 子类类别 -> 父类类别`。

---

5. Category 与关联对象

之前我们提到过，在 Category 中虽然可以添加属性，但是不会生成对应的成员变量，也不能生成 `getter`、`setter` 方法。因此，在调用 Category 中声明的属性时会报错。

那么就没有办法使用 Category 中的属性了吗?

答案当然是否定的。

我们可以自己来实现 `getter`、`setter` 方法，并借助关联对象（Objective-C Associated Objects）来实现 `getter`、`setter` 方法。关联对象能够帮助我们在运行时阶段将任意的属性关联到一个对象上。具体需要用到以下几个方法：

```
// 1. 通过 key : value 的形式给对象 object 设置关联属性
void objc_setAssociatedObject(id object, const void *key, id value, objc_AssociationPoli

// 2. 通过 key 获取关联的属性 object
id objc_getAssociatedObject(id object, const void *key);

// 3. 移除对象所关联的属性
void objc_removeAssociatedObjects(id object);
```

下面讲解一个示例。

5.1 UIImage 分类中增加网络地址属性

```Objc
/********************** UIImage+Property.h 文件 **********************/

#import <UIKit/UIKit.h>

@interface UIImage (Property)
```

```
/* 图片网络地址 */
@property (nonatomic, copy) NSString *urlString;

// 用于清除关联对象
- (void)clearAssociatedObjcet;

@end

/********************** UIImage+Property.m 文件 **********************/

#import "UIImage+Property.h"
#import <objc/runtime.h>

@implementation UIImage (Property)

// set 方法
- (void)setUrlString:(NSString *)urlString {
    objc_setAssociatedObject(self, @selector(urlString), urlString, OBJC_ASSOCIATION_COP
}

// get 方法
- (NSString *)urlString {
    return objc_getAssociatedObject(self, @selector(urlString));
}

// 清除关联对象
- (void)clearAssociatedObjcet {
    objc_removeAssociatedObjects(self);
}

@end
```

测试代码：

```Objc
UIImage *image = [[UIImage alloc] init];
image.urlString = @"http://www.image.png";

NSLog(@"image urlString = %@",image.urlString);

[image clearAssociatedObjcet];
NSLog(@"image urlString = %@",image.urlString);
```

> 打印结果： 2019-07-24 18:36:31.051789+0800 YSC-Category[74564:17944298]

> image urlString = www.image.png 2019-07-24 18:36:31.051926+0800 YSC-Category[74564:17944298] image urlString = (null)

可以看到：借助关联对象，我们成功的在 UIImage 分类中为 UImage 类增加了 urlString 关联属性，并实现了 `getter`、`setter` 方法。

> 注意：使用 `objc_removeAssociatedObjects` 可以断开所有的关联。通常情况下不建议使用，因为它会断开所有的关联。如果想要断开关联可以使用 `objc_setAssociatedObject`，将关联对象传入 nil 即可。

参考资料

- 美团技术团队：深入理解Objective-C：Category
- CJS_：iOS分类底层实现原理小记
- 梧雨北辰：Runtime-iOS运行时应用篇
- objc4 苹果开源代码 | 文中参考：objc4-750 版本
- dyld 苹果开源代码 | 文中参考：dyld-635.2 版本

最后

最后说一句，其实一开始只想随便写写关于 Category 与关联对象。结果不小心触碰到了 Category 的底层知识。。。然后就不小心写多了。心累。。。

文中如若有误，烦请指正，感谢。

---

iOS 开发：『Runtime』详解 系列文章：

- iOS 开发：『Runtime』详解（一）基础知识
- iOS 开发：『Runtime』详解（二）Method Swizzling
- iOS 开发：『Runtime』详解（三）Category 底层原理

尚未完成：

- iOS 开发：『Runtime』详解（四）获取类详细属性及应用
- iOS 开发：『Runtime』详解（五）Crash 防护系统
- iOS 开发：『Runtime』详解（六）Objective-C 2.0 结构解析
- iOS 开发：『Runtime』详解（七）KVO 底层实现
