objc4源码分析

理解runtime做了什么，对象到底是什么，很多问题的答案就会变得非常清晰，比如：

- 为什么Category不能添加ivar?
- 方法查找的本质是什么?
- Category的同名方法为什么会覆盖原类的方法？为什么Protocol不会？属性重名会怎么样?
- 子类实现了和父类同名的私有方法会覆盖父类的实现吗？为什么?
- 类对象、元类对象是什么?
- 为什么可以给类对象和元类对象发送消息?
- IMP、Method、SEL的区别?
- Method Swizzling做了什么?
- associatedObject到底是什么?

这些问题在阅读过源码后就会变得很容易理解

前言

之前的问题中，基本上所有都和runtime有关，所以runtime是什么？运行时又是什么？runtime和运行时有什么关系?

- runtime是一个动态连接库，被dyld加载到进程的内存空间，这个动态连接库包含了一系列的运行时能力，包括类对象的定义和加载、消息发送、方法查找、动态方法解析、反射等等
- 运行时：广义上是指**程序从启动到退出之间的整个生命周期**，即**程序在操作系统加载后被 CPU 执行的过程**
- runtime和运行时：runtime时运行时阶段所依赖的核心库，为Objective-C程序在运行时提供元数据管理、对象模型，消息分发等所有底层能力

1. 类对象

1.1 类对象是什么?

代码块

```
1   id object = [[NSObject alloc] init];
```

- [[NSObject alloc] init]创建了一个实例，也就是一块NSObject的对象布局的内存地址，而object是一个指针，指向这一块内存地址的头部，这也就是我们所说的持有了这个对象的引用
- 在objective-c中，基本上所有对象都继承自两个基类NSObject和NSProxy，而对象在运行时是这样定义的

代码块

```
typedef struct objc_class *Class;

/// Represents an instance of a class.
struct objc_object {
    Class _Nonnull isa  OBJC_ISA_AVAILABILITY;
};

/// A pointer to an instance of a class.
typedef struct objc_object *id;
```

- 所以说，所有对象都是一个objc_object，而id就是一个objc_object类型的指针，所以可以指向任何对象
- 我们都说，有isa的就是对象，因为有完整的isa链就能收发消息

super关键字

代码块

```
struct objc_super {
    /// Specifies an instance of a class.
    __unsafe_unretained _Nonnull id receiver;

    /// Specifies the particular superclass of the instance to message.
    __unsafe_unretained _Nonnull Class super_class;

    /* super_class is the first class to search */
};
```

- 在objc4的定义中，super就是一个结构体，存储着实例和它的superClass，调用super的时候，会第一个搜索super_class而不是receiver

objc_class

代码块

```
struct objc_class : objc_object {
  objc_class(const objc_class&) = delete;
  objc_class(objc_class&&) = delete;
```

```
4     void operator=(const objc_class&) = delete;
5     void operator=(objc_class&&) = delete;
6       // Class ISA;
7       Class superclass;
8       cache_t cache;             // formerly cache pointer and vtable
9       class_data_bits_t bits;    // class_rw_t * plus custom rr/alloc flags
10
11      Class getSuperclass() const {
12  #if __has_feature(ptrauth_calls)
13  #   if ISA_SIGNING_AUTH_MODE == ISA_SIGNING_AUTH
14          if (superclass == Nil)
15              return Nil;
16
17  #if SUPERCLASS_SIGNING_TREAT_UNSIGNED_AS_NIL
18          void *stripped = ptrauth_strip((void *)superclass, ISA_SIGNING_KEY);
19          if ((void *)superclass == stripped) {
20              void *resigned = ptrauth_sign_unauthenticated(stripped,
    ISA_SIGNING_KEY, ptrauth_blend_discriminator(&superclass,
    ISA_SIGNING_DISCRIMINATOR_CLASS_SUPERCLASS));
21              if ((void *)superclass != resigned)
22                  return Nil;
23          }
24  #endif
25
26          void *result = ptrauth_auth_data((void *)superclass, ISA_SIGNING_KEY,
    ptrauth_blend_discriminator(&superclass,
    ISA_SIGNING_DISCRIMINATOR_CLASS_SUPERCLASS));
27          return (Class)result;
28
29  #   else
30          return (Class)ptrauth_strip((void *)superclass, ISA_SIGNING_KEY);
31  #   endif
32  #else
33          return superclass;
34  #endif
35      }
36
37      class_rw_t *data() const {
38          return bits.data();
39      }
```

- objc_class的结构体：继承自objc_object，所以说Class也有isa，是一个对象，它也可以接收+转发消息
- 类对象isa元类对象，而元类对象也是一个Class，不同的只是元类对象存储的是类方法

- 类对象和元类对象在dyld之后被runtime加载并建立isa、superclass关系，在进入main()函数之前被全部加载和注册
- cache：存储最近查找过的methods(这里不准确，后面有解释)和protocols
- bits：可以看到，著名的rw_t就是bits.data()

class_rw_t

代码块

```
struct class_rw_t {
    Class firstSubclass;
    Class nextSiblingClass;

private:
    const class_ro_t *ro() const {
        auto v = get_ro_or_rwe();
        if (slowpath(v.is<class_rw_ext_t *>())) {
            return v.get<class_rw_ext_t *>(&ro_or_rw_ext)->ro;
        }
        return v.get<const class_ro_t *>(&ro_or_rw_ext);
    }

    const method_array_t methods() const {
        auto alternates = methodAlternates();
        if (auto *array = alternates.array)
            return *array;
        if (auto *list = alternates.list)
            return method_array_t{list};
        if (auto *relativeList = alternates.relativeList)
            return method_array_t{relativeList};
        return method_array_t{};
    }

    const property_array_t properties() const {
        auto v = get_ro_or_rwe();
        if (v.is<class_rw_ext_t *>()) {
            return v.get<class_rw_ext_t *>(&ro_or_rw_ext)->properties;
        } else {
            auto &baseProperties = v.get<const class_ro_t *>(&ro_or_rw_ext)-
>baseProperties;
            return property_array_t{baseProperties};
        }
    }

    const protocol_array_t protocols() const {
        auto v = get_ro_or_rwe();
        if (v.is<class_rw_ext_t *>()) {
```

```
            return v.get<class_rw_ext_t *>(&ro_or_rw_ext)->protocols;
        } else {
            auto &baseProtocols = v.get<const class_ro_t *>(&ro_or_rw_ext)->baseProtocols;
            return protocol_array_t{baseProtocols};
        }
    }
};
```

- class_rw_t：也就是class_readwrite_t，存储在运行时动态添加的方法，协议，属性，可能来自于Category或者运行时方法
- 这就是为什么Category不能添加实例变量的原因，因为ivar在编译期就已经确定，被添加进了ro_t
- setAssociatedObject并不在rw_t中，关联对象的的名字起的很好，associatedObject是通过一个全局map的形式，把变量和objc_object关联

class_ro_t

代码块

```
struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
#ifdef __LP64__
    uint32_t reserved;
#endif

    union {
        const uint8_t * ivarLayout;
        Class nonMetaclass;
    };

    explicit_atomic<const char *> name;
    objc::PointerUnion<method_list_t, relative_list_list_t<method_list_t>, method_list_t::Ptrauth, method_list_t::Ptrauth> baseMethods;
    objc::PointerUnion<protocol_list_t, relative_list_list_t<protocol_list_t>, PtrauthRaw, PtrauthRaw> baseProtocols;
    const ivar_list_t * ivars;

    const uint8_t * weakIvarLayout;
    objc::PointerUnion<property_list_t, relative_list_list_t<property_list_t>, PtrauthRaw, PtrauthRaw> baseProperties;

    // This field exists only when RO_HAS_SWIFT_INITIALIZER is set.
```

```
    _objc_swiftMetadataInitializer __ptrauth_objc_method_list_imp
_swiftMetadataInitializer_NEVER_USE[0];

    _objc_swiftMetadataInitializer swiftMetadataInitializer() const {
        if (flags & RO_HAS_SWIFT_INITIALIZER) {
            return _swiftMetadataInitializer_NEVER_USE[0];
        } else {
            return nil;
        }
    }
};
```

- 成员：baseMethods、baseProtocols、ivars、baseProperties
- 在编译期就已经确定

1.1.1 方法协议查询产生的问题

先复习一下方法和协议查询，不提后续isa链和消息转发的查询流程是这样

- objc_object通过isa找到自己的类对象，先查cache_t，再查rw_t，再查ro_t，找不到就会继续isa链，找了就缓存进cache_t然后返回

我们都知道，Category会覆盖原类的方法

- 原因也很简单，Category中的协议，属性，方法都会被runtime塞进rw_t中，因为查询顺序的原因，先查rw_t，找到就返回，所以并不是方法覆盖，只是有多个方法，找到就返回了

这样会引出一个问题，protocol为什么不影响，属性会不会影响?

- 协议的查询：conformsToProtocol会在找到后就返回，所以不会受影响，只要有就行，反正是一样的
- 属性的查询：

代码块

```
struct property_t {
    const char *name;
    const char *attributes;
};
```

属性的结构体只有一个name和attributes，attributes中S和G字符串表示这个属性的getter和setter具体是哪个selector

- 类和Category实现了同样name的属性编译器不会报错
- runtime查询属性实现的本质其实就是查询方法，所以如果有同名属性，并且属性的getter或setter方法名相同，也会影响方法查询，谁后加载就用谁

1.1.2 selector到底是什么?

使用clang -rewrite-objc LabObject.m之后，可以看到方法调用的转换

代码块

```
((LabObject *(*)(id, SEL))(void *)objc_msgSend)((id)((LabObject *(*)(id, SEL))
(void *)objc_msgSend)((id)objc_getClass("LabObject"),
sel_registerName("alloc")), sel_registerName("init"))
```

- sel_registerName将会返回一个SEL，SEL是一个objc_selector结构体，而objc_selector的本质是一个字符串指针
- 相同的SEL都指向同一块地址，在发送消息时，唯一不同的就是接收对象，msgSend会根据SEL和object找到类对象的method，调用imp
- 对比SEL的本质是对比字符串指针，相同的SEL总是指向同一块地址，这是runtime对查询的一种优化，对比指针的速度是极快的

这里要注意的是Objective-C并不关心参数类型，即使参数类型不同，只要名称一样，就会被认为是同一个SEL

代码块

```
- (void)someMethod:(NSObject *)object;
- (void)someMethod:(NSProxy *)proxy;
```

Objective-C不会关心参数类型，甚至不会关心参数名，这两个方法会被认为是同一个SEL，后面定义的method会覆盖前面的，这也就是Objective-C不支持重载的原因

2. Method Swizzling的本质是什么

代码块

```
SEL originalSelector = NSSelectorFromString(selectors[i]);
SEL swizzledSelector = NSSelectorFromString(swizzleds[i]);

Method originalMethod = class_getInstanceMethod(class, originalSelector);
Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector);

if (!originalMethod || !swizzledMethod) continue;

BOOL didAddMethod =
class_addMethod(class,
                originalSelector,
                method_getImplementation(swizzledMethod),
                method_getTypeEncoding(swizzledMethod));
```

```
14  if (didAddMethod) {
15      class_replaceMethod(class,
16                          swizzledSelector,
17                          method_getImplementation(originalMethod),
18                          method_getTypeEncoding(originalMethod));
19  } else {
20      method_exchangeImplementations(originalMethod, swizzledMethod);
21  }
```

具体逻辑简要分析如下

1. 通过SEL找到Method
2. 尝试将swizzlingMethod的实现添加到originalSelector上
3. 如果添加上了，替换sizzledMethod的实现
4. 如果没有，交换两个方法的实现

所以最关键的方法就是class_addMethod、class_replaceMethod、method_exchangeImplementations

代码块

```
1   BOOL
2   class_addMethod(Class cls, SEL name, IMP imp, const char *types)
3   {
4       if (!cls) return NO;
5
6       mutex_locker_t lock(runtimeLock);
7       return ! addMethod(cls, name, imp, types ?: "", NO);
8   }
9
10
11  IMP
12  class_replaceMethod(Class cls, SEL name, IMP imp, const char *types)
13  {
14      if (!cls) return nil;
15
16      mutex_locker_t lock(runtimeLock);
17      return addMethod(cls, name, imp, types ?: "", YES);
18  }
```

可以看到addMethod和replaceMethod调用了同一个方法，只是最后传入的参数不同

然而在objc-runtime-new.mm文件中，似乎并没有addMethod这个方法的准确实现，只有定义

```
static IMP addMethod(Class cls, SEL name, IMP imp, const char *types, bool replace);
```

method_exchangeImplementations

代码块

```
void method_exchangeImplementations(Method m1Signed, Method m2Signed)
{
    if (!m1Signed  ||  !m2Signed) return;

    method_t *m1 = _method_auth(m1Signed);
    method_t *m2 = _method_auth(m2Signed);

    mutex_locker_t lock(runtimeLock);

    IMP imp1 = m1->imp(false);
    IMP imp2 = m2->imp(false);
    SEL sel1 = m1->name();
    SEL sel2 = m2->name();

    m1->setImp(imp2);
    m2->setImp(imp1);

    flushCaches(nil, __func__, [sel1, sel2, imp1, imp2](Class c){
        return c->cache.shouldFlush(sel1, imp1) || c->cache.shouldFlush(sel2, imp2);
    });

    adjustCustomFlagsForMethodChange(nil, m1);
    adjustCustomFlagsForMethodChange(nil, m2);
}
```

- 可以看到方法交换的本质就是IMP交换
- 最后会更新缓存，因为缓存里的SEL可能还是指向原来的imp
- 所以由此可知，objc_class中的cache_t实际上缓存的是SEL到IMP的映射，而不是Method
- 最后会更新方法的flag，flag会表示该method的实现是不是被runtime替换过

3. AssociatedObject的本质是什么?

关联对象其实并不复杂

代码块

```
id
objc_getAssociatedObject(id object, const void *key)
{
    return _object_get_associative_reference(object, key);
}

void
objc_setAssociatedObject(id object, const void *key, id value,
objc_AssociationPolicy policy)
{
    _object_set_associative_reference(object, key, value, policy);
}
```

- 所以重点就是这两个方法_object_get_associative_reference和 _object_set_associative_reference

_object_get_associative_reference

代码块

```
id
_object_get_associative_reference(id object, const void *key)
{
    ObjcAssociation association{};

    {
        AssociationsManager manager;
        AssociationsHashMap &associations(manager.get());
        AssociationsHashMap::iterator i = associations.find((objc_object
*)object);
        if (i != associations.end()) {
            ObjectAssociationMap &refs = i->second;
            ObjectAssociationMap::iterator j = refs.find(key);
            if (j != refs.end()) {
                association = j->second;
                association.retainReturnedValue();
            }
        }
    }

    return association.autoreleaseReturnedValue();
}
```

先了解一下基本的数据结构的定义

```
class ObjcAssociation {
    uintptr_t _policy;
    id _value;
}

typedef DenseMap<const void *, ObjcAssociation> ObjectAssociationMap;

typedef DenseMap<DisguisedPtr<objc_object>, ObjectAssociationMap>
AssociationsHashMap;
```

- ObjcAssociation：代表一条关联对象的记录，包含policy和value
- AssociationsHashMap：object到ObjectAssociationMap之间的映射，是一个全局map
- ObjectAssociationMap：key到ObjectAssociation之间的映射

接下来分析源码

- 声明一个空的ObjcAssociation结构体
- `AssociationsManager` 是一个 RAII（自动资源管理）对象，进入作用域就加锁，析构时自动解锁，确保线程安全。
- 从manager中获取全局的关联对象哈希表，这里存储在所有关联对象的映射

代码块

```
AssociationsHashMap::iterator i = associations.find((objc_object *)object);
```

- 查找关联对象在不在map里，i != end代表找到了，后面的i->second相关的操作都是c++里的取value

代码块

```
class ObjcAssociation {
    uintptr_t _policy;
    id _value;
public:
    inline void retainReturnedValue() {
        if (_value && (_policy & OBJC_ASSOCIATION_GETTER_RETAIN)) {
            objc_retain(_value);
        }
    }

    inline id autoreleaseReturnedValue() {
        if (slowpath(_value && (_policy &
OBJC_ASSOCIATION_GETTER_AUTORELEASE))) {
            return objc_autorelease(_value);
```

```
        }
        return _value;
    }
};

typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,           /**< Specifies an unsafe unretained reference to the associated object. */
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, /**< Specifies a strong reference to the associated object.
                                            *   The association is not made atomically. */
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,   /**< Specifies that the associated object is copied.
                                            *   The association is not made atomically. */
    OBJC_ASSOCIATION_RETAIN = 01401,       /**< Specifies a strong reference to the associated object.
                                            *   The association is made atomically. */
    OBJC_ASSOCIATION_COPY = 01403          /**< Specifies that the associated object is copied.
                                            *   The association is made atomically. */
};
```

- 内存管理语义，如果set的时候指定了retain，找到后先增加引用计数，防止返回前对象被释放
- `slowpath(...)`：这是 clang/gcc 的 branch prediction 宏，提示编译器这个条件不常走（性能优化，不影响逻辑）
- atomic修饰的属性会默认返回autorelease对象，所以在使用set管理对象的时候，policy就已经决定了get关联对象需不需要返回autorelease对象

总结

- getAssociatedObject本质是就是在一个全局的map中根据object找到这个object关联的所有对象，然后根据key返回这个关联对象的值，如果是atomic修饰符修饰的会返回autorelease对象

_object_set_associative_reference

代码块

```
void
_object_set_associative_reference(id object, const void *key, id value, uintptr_t policy)
{
    // This code used to work when nil was passed for object and key. Some code
```

```
    // probably relies on that to not crash. Check and handle it explicitly.
    // rdar://problem/44094390
    if (!object && !value) return;

    Class objectClass = object->getIsa();
    objectClass->realizeIfNeeded();

    if (objectClass->forbidsAssociatedObjects())
        _objc_fatal("objc_setAssociatedObject called on instance (%p) of class %s which does not allow associated objects", object, object_getClassName(object));

    DisguisedPtr<objc_object> disguised{(objc_object *)object};
    ObjcAssociation association{policy, value};

    // retain the new value (if any) outside the lock.
    association.acquireValue();

    bool isFirstAssociation = false;
    {
        AssociationsManager manager;
        AssociationsHashMap &associations(manager.get());

        if (value) {
            auto refs_result = associations.try_emplace(disguised, ObjectAssociationMap{});
            if (refs_result.second) {
                /* it's the first association we make */
                isFirstAssociation = true;
            }

            /* establish or replace the association */
            auto &refs = refs_result.first->second;
            auto result = refs.try_emplace(key, std::move(association));
            if (!result.second) {
                association.swap(result.first->second);
            }
        } else {
            auto refs_it = associations.find(disguised);
            if (refs_it != associations.end()) {
                auto &refs = refs_it->second;
                auto it = refs.find(key);
                if (it != refs.end()) {
                    association.swap(it->second);
                    refs.erase(it);
                    if (refs.size() == 0) {
                        associations.erase(refs_it);
```

```
49
50                      }
51                  }
52              }
53          }
54      }
55
56      // Call setHasAssociatedObjects outside the lock, since this
57      // will call the object's _noteAssociatedObjects method if it
58      // has one, and this may trigger +initialize which might do
59      // arbitrary stuff, including setting more associated objects.
60      if (isFirstAssociation)
61          object->setHasAssociatedObjects();
62
63      // release the old value (outside of the lock).
64      association.releaseHeldValue();
65  }
```

- 确保类已经realize：获取对象的实际类，并确保 class 已经被 runtime realize（比如 lazy-load 类）。不 realize 可能导致 class 的元数据（如 associated flag、property table）未初始化
- 如果某些对象禁止关联对象，报错
- 根据policy和value初始化ObjcAssociation
- association.acquireValue()根据策略retain或者copy新值
- 区分value有没有值的情况处理设置或者删除
- 有值：直接写入，先在全局表查找/插入 object 的 map，如果是新 object（第二返回 true），标记首个关联。然后再插入/查找这个 object 的 key-value，如果已有，直接 swap，回收旧值。
  - `try_emplace` 是 C++17 map 的优化用法：只有 key 不存在才会 emplace，省一次构造和查找。
- 没有值：把旧的ObjcAssociation取出来，swap到临时变量，后续回收，如果该object的关联对象全部回收了，删除object相关的关联表
- 如果是第一个关联对象，设置runtime状态位，通知对象
- 释放旧的关联对象