

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是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

```

1  typedef struct objc_class *Class;
2
3  /// Represents an instance of a class.
4  struct objc_object {
5      Class _Nonnull isa OBJC_ISA_AVAILABILITY;
6  };
7
8  /// A pointer to an instance of a class.
9  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

```

1  struct objc_super {
2      /// Specifies an instance of a class.
3      __unsafe_unretained _Nonnull id receiver;
4
5      /// Specifies the particular superclass of the instance to message.
6      __unsafe_unretained _Nonnull Class super_class;
7
8      /* super_class is the first class to search */
9  };

```

- 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

```

1  struct objc_class : objc_object {
2      objc_class(const objc_class&) = delete;
3      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 getClass() 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,
21 ISA_SIGNING_KEY, ptrauth_blend_discriminator(&superclass,
22 ISA_SIGNING_DISCRIMINATOR_CLASS_SUPERCLASS));
23         if ((void *)superclass != resigned)
24             return Nil;
25     }
26 #endif
27     void *result = ptrauth_auth_data((void *)superclass, ISA_SIGNING_KEY,
28 ptrauth_blend_discriminator(&superclass,
29 ISA_SIGNING_DISCRIMINATOR_CLASS_SUPERCLASS));
30     return (Class)result;
31
32 # else
33     return (Class)ptrauth_strip((void *)superclass, ISA_SIGNING_KEY);
34 # endif
35 #else
36     return superclass;
37 #endif
38 }
39
40
41     class_rw_t *data() const {
42         return bits.data();
43     }

```

- `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

```

1  struct class_rw_t {
2      Class firstSubclass;
3      Class nextSiblingClass;
4
5  private:
6      const class_ro_t *ro() const {
7          auto v = get_ro_or_rwe();
8          if (slowpath(v.is<class_rw_ext_t *>())) {
9              return v.get<class_rw_ext_t *>(&ro_or_rw_ext)->ro;
10         }
11         return v.get<const class_ro_t *>(&ro_or_rw_ext);
12     }
13
14     const method_array_t methods() const {
15         auto alternates = methodAlternates();
16         if (auto *array = alternates.array)
17             return *array;
18         if (auto *list = alternates.list)
19             return method_array_t{list};
20         if (auto *relativeList = alternates.relativeList)
21             return method_array_t{relativeList};
22         return method_array_t{};
23     }
24
25     const property_array_t properties() const {
26         auto v = get_ro_or_rwe();
27         if (v.is<class_rw_ext_t *>()) {
28             return v.get<class_rw_ext_t *>(&ro_or_rw_ext)->properties;
29         } else {
30             auto &baseProperties = v.get<const class_ro_t *>(&ro_or_rw_ext)-
>baseProperties;
31             return property_array_t{baseProperties};
32         }
33     }
34
35     const protocol_array_t protocols() const {
36         auto v = get_ro_or_rwe();
37         if (v.is<class_rw_ext_t *>()) {

```

```

38             return v.get<class_rw_ext_t *>(&ro_or_rw_ext)->protocols;
39     } else {
40         auto &baseProtocols = v.get<const class_ro_t *>(&ro_or_rw_ext)-
>baseProtocols;
41         return protocol_array_t{baseProtocols};
42     }
43 }
44 };

```

- 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

```

1  struct class_ro_t {
2      uint32_t flags;
3      uint32_t instanceStart;
4      uint32_t instanceSize;
5 #ifdef __LP64__
6      uint32_t reserved;
7 #endif
8
9     union {
10         const uint8_t * ivarLayout;
11         Class nonMetaClass;
12     };
13
14     explicit_atomic<const char *> name;
15     objc::PointerUnion<method_list_t, relative_list_list_t<method_list_t>,
method_list_t::Ptrauth, method_list_t::Ptrauth> baseMethods;
16     objc::PointerUnion<protocol_list_t, relative_list_list_t<protocol_list_t>,
PtrauthRaw, PtrauthRaw> baseProtocols;
17     const ivar_list_t * ivars;
18
19     const uint8_t * weakIvarLayout;
20     objc::PointerUnion<property_list_t, relative_list_list_t<property_list_t>,
PtrauthRaw, PtrauthRaw> baseProperties;
21
22     // This field exists only when RO_HAS_SWIFT_INITIALIZER is set.

```

```

23     _objc_swiftMetadataInitializer __ptrauth_objc_method_list_imp
24     _swiftMetadataInitializer_NEVER_USE[0];
25
26     _objc_swiftMetadataInitializer swiftMetadataInitializer() const {
27         if (flags & RO_HAS_SWIFT_INITIALIZER) {
28             return _swiftMetadataInitializer_NEVER_USE[0];
29         } else {
30             return nil;
31         }
32     };

```

- 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

```

1 struct property_t {
2     const char *name;
3     const char *attributes;
4 };

```

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

```
1 ((LabObject *(*) (id, SEL)) (void *)objc_msgSend) ((id) ((LabObject *(*) (id, SEL))  
2 (void *)objc_msgSend) ((id)objc_getClass("LabObject"),  
3 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

```
1 - (void)someMethod:(NSObject *)object;  
2 - (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

```
1 SEL originalSelector = NSSelectorFromString(selectors[i]);  
2 SEL swizzledSelector = NSSelectorFromString(swizzles[i]);  
3  
4 Method originalMethod = class_getInstanceMethod(class, originalSelector);  
5 Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector);  
6  
7 if (!originalMethod || !swizzledMethod) continue;  
8  
9 BOOL didAddMethod =  
10 class_addMethod(class,  
11                 originalSelector,  
12                 method_getImplementation(swizzledMethod),  
13                 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swizzledMethod的实现添加到originalSelector上
3. 如果添加上了，替换swizzledMethod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

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

method_exchangeImplementations

代码块

```
1 void method_exchangeImplementations(Method m1Signed, Method m2Signed)  
2 {  
3     if (!m1Signed || !m2Signed) return;  
4  
5     method_t *m1 = _method_auth(m1Signed);  
6     method_t *m2 = _method_auth(m2Signed);  
7  
8     mutex_locker_t lock(runtimeLock);  
9  
10    IMP imp1 = m1->imp(false);  
11    IMP imp2 = m2->imp(false);  
12    SEL sel1 = m1->name();  
13    SEL sel2 = m2->name();  
14  
15    m1->setImp(imp2);  
16    m2->setImp(imp1);  
17  
18    flushCaches(nil, __func__, [sel1, sel2, imp1, imp2](Class c){  
19        return c->cache.shouldFlush(sel1, imp1) || c->cache.shouldFlush(sel2,  
imp2);  
20    });  
21  
22    adjustCustomFlagsForMethodChange(nil, m1);  
23    adjustCustomFlagsForMethodChange(nil, m2);  
24 }
```

- 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

```

1  id
2  objc_getAssociatedObject(id object, const void *key)
3  {
4      return _object_get_associative_reference(object, key);
5  }
6
7  void
8  objc_setAssociatedObject(id object, const void *key, id value,
9   objc_AssociationPolicy policy)
10 {
11     _object_set_associative_reference(object, key, value, policy);
12 }
```

- 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

```

1  id
2  _object_get_associative_reference(id object, const void *key)
3  {
4      ObjcAssociation association{};
5
6      {
7          AssociationsManager manager;
8          AssociationsHashMap &associations(manager.get());
9          AssociationsHashMap::iterator i = associations.find((objc_object
*)object);
10         if (i != associations.end()) {
11             ObjectAssociationMap &refs = i->second;
12             ObjectAssociationMap::iterator j = refs.find(key);
13             if (j != refs.end()) {
14                 association = j->second;
15                 association.retainReturnedValue();
16             }
17         }
18     }
19
20     return association.autoreleaseReturnedValue();
21 }
```

先了解一下基本的数据结构的定义

```

1 class ObjcAssociation {
2     uintptr_t _policy;
3     id _value;
4 }
5
6 typedef DenseMap<const void *, ObjcAssociation> ObjectAssociationMap;
7
8 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ObjcAssociation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

```
1 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

```

1 class ObjcAssociation {
2     uintptr_t _policy;
3     id _value;
4 public:
5     inline void retainReturnedValue() {
6         if (_value && (_policy &_OBJC_ASSOCIATION_GETTER_RETAIN)) {
7             objc_retain(_value);
8         }
9     }
10
11    inline id autoreleaseReturnedValue() {
12        if (slowpath(_value && (_policy &
_OBJC_ASSOCIATION_GETTER_AUTORELEASE))) {
13            return objc_autorelease(_value);

```

```

14         }
15     return _value;
16 }
17 };
18
19 typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
20    _OBJC_ASSOCIATION_ASSIGN = 0,           /***< Specifies an unsafe unretained
reference to the associated object. */
21    _OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, /***< Specifies a strong reference
to the associated object.
22                                         * The association is not made
atomically. */
23    _OBJC_ASSOCIATION_COPY_NONATOMIC = 3,   /***< Specifies that the associated
object is copied.
24                                         * The association is not made
atomically. */
25    _OBJC_ASSOCIATION_RETAIN = 01401,        /***< Specifies a strong reference
to the associated object.
26                                         * The association is made
atomically. */
27    _OBJC_ASSOCIATION_COPY = 01403          /***< Specifies that the associated
object is copied.
28                                         * The association is made
atomically. */
29 };

```

- 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

```

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

```

```

5   // probably relies on that to not crash. Check and handle it explicitly.
6   // rdar://problem/44094390
7   if (!object && !value) return;
8
9   Class objectClass = object->getIsa();
10  objectClass->realizeIfNeeded();
11
12  if (objectClass->forbidsAssociatedObjects())
13      _objc_fatal("objc_setAssociatedObject called on instance (%p) of class
%s which does not allow associated objects", object,
14      object_>getClassName(object));
15
16  DisguisedPtr<objc_object> disguised{(objc_object *)object};
17  ObjcAssociation association{policy, value};
18
19  // retain the new value (if any) outside the lock.
20  association.acquireValue();
21
22  bool isFirstAssociation = false;
23  {
24      AssociationsManager manager;
25      AssociationsHashMap &associations(manager.get());
26
27      if (value) {
28          auto refs_result = associations.try_emplace(disguised,
29          ObjectAssociationMap{});
30          if (refs_result.second) {
31              /* it's the first association we make */
32              isFirstAssociation = true;
33          }
34
35          /* establish or replace the association */
36          auto &refs = refs_result.first->second;
37          auto result = refs.try_emplace(key, std::move(association));
38          if (!result.second) {
39              association.swap(result.first->second);
40          }
41      } else {
42          auto refs_it = associations.find(disguised);
43          if (refs_it != associations.end()) {
44              auto &refs = refs_it->second;
45              auto it = refs.find(key);
46              if (it != refs.end()) {
47                  association.swap(it->second);
48                  refs.erase(it);
49                  if (refs.size() == 0) {
50                      associations.erase(refs_it);
51                  }
52              }
53          }
54      }
55  }
56
57  if (isFirstAssociation)
58      association.releaseValue();
59
60  return association;
61
62  else {
63      auto &refs = refs_it->second;
64      auto it = refs.find(key);
65      if (it != refs.end()) {
66          association.swap(it->second);
67          refs.erase(it);
68          if (refs.size() == 0) {
69              associations.erase(refs_it);
70          }
71      }
72  }
73
74  return association;
75
76  else {
77      auto &refs = refs_it->second;
78      auto it = refs.find(key);
79      if (it != refs.end()) {
80          association.swap(it->second);
81          refs.erase(it);
82          if (refs.size() == 0) {
83              associations.erase(refs_it);
84          }
85      }
86  }
87
88  return association;
89
90  else {
91      auto &refs = refs_it->second;
92      auto it = refs.find(key);
93      if (it != refs.end()) {
94          association.swap(it->second);
95          refs.erase(it);
96          if (refs.size() == 0) {
97              associations.erase(refs_it);
98          }
99      }
100 }
101
102 return association;
103
104  else {
105      auto &refs = refs_it->second;
106      auto it = refs.find(key);
107      if (it != refs.end()) {
108          association.swap(it->second);
109          refs.erase(it);
110          if (refs.size() == 0) {
111              associations.erase(refs_it);
112          }
113      }
114  }
115
116  return association;
117
118  else {
119      auto &refs = refs_it->second;
120      auto it = refs.find(key);
121      if (it != refs.end()) {
122          association.swap(it->second);
123          refs.erase(it);
124          if (refs.size() == 0) {
125              associations.erase(refs_it);
126          }
127      }
128  }
129
130  return association;
131
132  else {
133      auto &refs = refs_it->second;
134      auto it = refs.find(key);
135      if (it != refs.end()) {
136          association.swap(it->second);
137          refs.erase(it);
138          if (refs.size() == 0) {
139              associations.erase(refs_it);
140          }
141      }
142  }
143
144  return association;
145
146  else {
147      auto &refs = refs_it->second;
148      auto it = refs.find(key);
149      if (it != refs.end()) {
150          association.swap(it->second);
151          refs.erase(it);
152          if (refs.size() == 0) {
153              associations.erase(refs_it);
154          }
155      }
156  }
157
158  return association;
159
160  else {
161      auto &refs = refs_it->second;
162      auto it = refs.find(key);
163      if (it != refs.end()) {
164          association.swap(it->second);
165          refs.erase(it);
166          if (refs.size() == 0) {
167              associations.erase(refs_it);
168          }
169      }
170  }
171
172  return association;
173
174  else {
175      auto &refs = refs_it->second;
176      auto it = refs.find(key);
177      if (it != refs.end()) {
178          association.swap(it->second);
179          refs.erase(it);
180          if (refs.size() == 0) {
181              associations.erase(refs_it);
182          }
183      }
184  }
185
186  return association;
187
188  else {
189      auto &refs = refs_it->second;
190      auto it = refs.find(key);
191      if (it != refs.end()) {
192          association.swap(it->second);
193          refs.erase(it);
194          if (refs.size() == 0) {
195              associations.erase(refs_it);
196          }
197      }
198  }
199
200  return association;
201
202  else {
203      auto &refs = refs_it->second;
204      auto it = refs.find(key);
205      if (it != refs.end()) {
206          association.swap(it->second);
207          refs.erase(it);
208          if (refs.size() == 0) {
209              associations.erase(refs_it);
210          }
211      }
212  }
213
214  return association;
215
216  else {
217      auto &refs = refs_it->second;
218      auto it = refs.find(key);
219      if (it != refs.end()) {
220          association.swap(it->second);
221          refs.erase(it);
222          if (refs.size() == 0) {
223              associations.erase(refs_it);
224          }
225      }
226  }
227
228  return association;
229
230  else {
231      auto &refs = refs_it->second;
232      auto it = refs.find(key);
233      if (it != refs.end()) {
234          association.swap(it->second);
235          refs.erase(it);
236          if (refs.size() == 0) {
237              associations.erase(refs_it);
238          }
239      }
240  }
241
242  return association;
243
244  else {
245      auto &refs = refs_it->second;
246      auto it = refs.find(key);
247      if (it != refs.end()) {
248          association.swap(it->second);
249          refs.erase(it);
250          if (refs.size() == 0) {
251              associations.erase(refs_it);
252          }
253      }
254  }
255
256  return association;
257
258  else {
259      auto &refs = refs_it->second;
260      auto it = refs.find(key);
261      if (it != refs.end()) {
262          association.swap(it->second);
263          refs.erase(it);
264          if (refs.size() == 0) {
265              associations.erase(refs_it);
266          }
267      }
268  }
269
270  return association;
271
272  else {
273      auto &refs = refs_it->second;
274      auto it = refs.find(key);
275      if (it != refs.end()) {
276          association.swap(it->second);
277          refs.erase(it);
278          if (refs.size() == 0) {
279              associations.erase(refs_it);
280          }
281      }
282  }
283
284  return association;
285
286  else {
287      auto &refs = refs_it->second;
288      auto it = refs.find(key);
289      if (it != refs.end()) {
290          association.swap(it->second);
291          refs.erase(it);
292          if (refs.size() == 0) {
293              associations.erase(refs_it);
294          }
295      }
296  }
297
298  return association;
299
300  else {
301      auto &refs = refs_it->second;
302      auto it = refs.find(key);
303      if (it != refs.end()) {
304          association.swap(it->second);
305          refs.erase(it);
306          if (refs.size() == 0) {
307              associations.erase(refs_it);
308          }
309      }
310  }
311
312  return association;
313
314  else {
315      auto &refs = refs_it->second;
316      auto it = refs.find(key);
317      if (it != refs.end()) {
318          association.swap(it->second);
319          refs.erase(it);
320          if (refs.size() == 0) {
321              associations.erase(refs_it);
322          }
323      }
324  }
325
326  return association;
327
328  else {
329      auto &refs = refs_it->second;
330      auto it = refs.find(key);
331      if (it != refs.end()) {
332          association.swap(it->second);
333          refs.erase(it);
334          if (refs.size() == 0) {
335              associations.erase(refs_it);
336          }
337      }
338  }
339
340  return association;
341
342  else {
343      auto &refs = refs_it->second;
344      auto it = refs.find(key);
345      if (it != refs.end()) {
346          association.swap(it->second);
347          refs.erase(it);
348          if (refs.size() == 0) {
349              associations.erase(refs_it);
350          }
351      }
352  }
353
354  return association;
355
356  else {
357      auto &refs = refs_it->second;
358      auto it = refs.find(key);
359      if (it != refs.end()) {
360          association.swap(it->second);
361          refs.erase(it);
362          if (refs.size() == 0) {
363              associations.erase(refs_it);
364          }
365      }
366  }
367
368  return association;
369
370  else {
371      auto &refs = refs_it->second;
372      auto it = refs.find(key);
373      if (it != refs.end()) {
374          association.swap(it->second);
375          refs.erase(it);
376          if (refs.size() == 0) {
377              associations.erase(refs_it);
378          }
379      }
380  }
381
382  return association;
383
384  else {
385      auto &refs = refs_it->second;
386      auto it = refs.find(key);
387      if (it != refs.end()) {
388          association.swap(it->second);
389          refs.erase(it);
390          if (refs.size() == 0) {
391              associations.erase(refs_it);
392          }
393      }
394  }
395
396  return association;
397
398  else {
399      auto &refs = refs_it->second;
400      auto it = refs.find(key);
401      if (it != refs.end()) {
402          association.swap(it->second);
403          refs.erase(it);
404          if (refs.size() == 0) {
405              associations.erase(refs_it);
406          }
407      }
408  }
409
410  return association;
411
412  else {
413      auto &refs = refs_it->second;
414      auto it = refs.find(key);
415      if (it != refs.end()) {
416          association.swap(it->second);
417          refs.erase(it);
418          if (refs.size() == 0) {
419              associations.erase(refs_it);
420          }
421      }
422  }
423
424  return association;
425
426  else {
427      auto &refs = refs_it->second;
428      auto it = refs.find(key);
429      if (it != refs.end()) {
430          association.swap(it->second);
431          refs.erase(it);
432          if (refs.size() == 0) {
433              associations.erase(refs_it);
434          }
435      }
436  }
437
438  return association;
439
440  else {
441      auto &refs = refs_it->second;
442      auto it = refs.find(key);
443      if (it != refs.end()) {
444          association.swap(it->second);
445          refs.erase(it);
446          if (refs.size() == 0) {
447              associations.erase(refs_it);
448          }
449      }
450  }
451
452  return association;
453
454  else {
455      auto &refs = refs_it->second;
456      auto it = refs.find(key);
457      if (it != refs.end()) {
458          association.swap(it->second);
459          refs.erase(it);
460          if (refs.size() == 0) {
461              associations.erase(refs_it);
462          }
463      }
464  }
465
466  return association;
467
468  else {
469      auto &refs = refs_it->second;
470      auto it = refs.find(key);
471      if (it != refs.end()) {
472          association.swap(it->second);
473          refs.erase(it);
474          if (refs.size() == 0) {
475              associations.erase(refs_it);
476          }
477      }
478  }
479
480  return association;
481
482  else {
483      auto &refs = refs_it->second;
484      auto it = refs.find(key);
485      if (it != refs.end()) {
486          association.swap(it->second);
487          refs.erase(it);
488          if (refs.size() == 0) {
489              associations.erase(refs_it);
490          }
491      }
492  }
493
494  return association;
495
496  else {
497      auto &refs = refs_it->second;
498      auto it = refs.find(key);
499      if (it != refs.end()) {
500          association.swap(it->second);
501          refs.erase(it);
502          if (refs.size() == 0) {
503              associations.erase(refs_it);
504          }
505      }
506  }
507
508  return association;
509
510  else {
511      auto &refs = refs_it->second;
512      auto it = refs.find(key);
513      if (it != refs.end()) {
514          association.swap(it->second);
515          refs.erase(it);
516          if (refs.size() == 0) {
517              associations.erase(refs_it);
518          }
519      }
520  }
521
522  return association;
523
524  else {
525      auto &refs = refs_it->second;
526      auto it = refs.find(key);
527      if (it != refs.end()) {
528          association.swap(it->second);
529          refs.erase(it);
530          if (refs.size() == 0) {
531              associations.erase(refs_it);
532          }
533      }
534  }
535
536  return association;
537
538  else {
539      auto &refs = refs_it->second;
540      auto it = refs.find(key);
541      if (it != refs.end()) {
542          association.swap(it->second);
543          refs.erase(it);
544          if (refs.size() == 0) {
545              associations.erase(refs_it);
546          }
547      }
548  }
549
550  return association;
551
552  else {
553      auto &refs = refs_it->second;
554      auto it = refs.find(key);
555      if (it != refs.end()) {
556          association.swap(it->second);
557          refs.erase(it);
558          if (refs.size() == 0) {
559              associations.erase(refs_it);
560          }
561      }
562  }
563
564  return association;
565
566  else {
567      auto &refs = refs_it->second;
568      auto it = refs.find(key);
569      if (it != refs.end()) {
570          association.swap(it->second);
571          refs.erase(it);
572          if (refs.size() == 0) {
573              associations.erase(refs_it);
574          }
575      }
576  }
577
578  return association;
579
580  else {
581      auto &refs = refs_it->second;
582      auto it = refs.find(key);
583      if (it != refs.end()) {
584          association.swap(it->second);
585          refs.erase(it);
586          if (refs.size() == 0) {
587              associations.erase(refs_it);
588          }
589      }
590  }
591
592  return association;
593
594  else {
595      auto &refs = refs_it->second;
596      auto it = refs.find(key);
597      if (it != refs.end()) {
598          association.swap(it->second);
599          refs.erase(it);
600          if (refs.size() == 0) {
601              associations.erase(refs_it);
602          }
603      }
604  }
605
606  return association;
607
608  else {
609      auto &refs = refs_it->second;
610      auto it = refs.find(key);
611      if (it != refs.end()) {
612          association.swap(it->second);
613          refs.erase(it);
614          if (refs.size() == 0) {
615              associations.erase(refs_it);
616          }
617      }
618  }
619
620  return association;
621
622  else {
623      auto &refs = refs_it->second;
624      auto it = refs.find(key);
625      if (it != refs.end()) {
626          association.swap(it->second);
627          refs.erase(it);
628          if (refs.size() == 0) {
629              associations.erase(refs_it);
630          }
631      }
632  }
633
634  return association;
635
636  else {
637      auto &refs = refs_it->second;
638      auto it = refs.find(key);
639      if (it != refs.end()) {
640          association.swap(it->second);
641          refs.erase(it);
642          if (refs.size() == 0) {
643              associations.erase(refs_it);
644          }
645      }
646  }
647
648  return association;
649
650  else {
651      auto &refs = refs_it->second;
652      auto it = refs.find(key);
653      if (it != refs.end()) {
654          association.swap(it->second);
655          refs.erase(it);
656          if (refs.size() == 0) {
657              associations.erase(refs_it);
658          }
659      }
660  }
661
662  return association;
663
664  else {
665      auto &refs = refs_it->second;
666      auto it = refs.find(key);
667      if (it != refs.end()) {
668          association.swap(it->second);
669          refs.erase(it);
670          if (refs.size() == 0) {
671              associations.erase(refs_it);
672          }
673      }
674  }
675
676  return association;
677
678  else {
679      auto &refs = refs_it->second;
680      auto it = refs.find(key);
681      if (it != refs.end()) {
682          association.swap(it->second);
683          refs.erase(it);
684          if (refs.size() == 0) {
685              associations.erase(refs_it);
686          }
687      }
688  }
689
690  return association;
691
692  else {
693      auto &refs = refs_it->second;
694      auto it = refs.find(key);
695      if (it != refs.end()) {
696          association.swap(it->second);
697          refs.erase(it);
698          if (refs.size() == 0) {
699              associations.erase(refs_it);
700          }
701      }
702  }
703
704  return association;
705
706  else {
707      auto &refs = refs_it->second;
708      auto it = refs.find(key);
709      if (it != refs.end()) {
710          association.swap(it->second);
711          refs.erase(it);
712          if (refs.size() == 0) {
713              associations.erase(refs_it);
714          }
715      }
716  }
717
718  return association;
719
720  else {
721      auto &refs = refs_it->second;
722      auto it = refs.find(key);
723      if (it != refs.end()) {
724          association.swap(it->second);
725          refs.erase(it);
726          if (refs.size() == 0) {
727              associations.erase(refs_it);
728          }
729      }
730  }
731
732  return association;
733
734  else {
735      auto &refs = refs_it->second;
736      auto it = refs.find(key);
737      if (it != refs.end()) {
738          association.swap(it->second);
739          refs.erase(it);
740          if (refs.size() == 0) {
741              associations.erase(refs_it);
742          }
743      }
744  }
745
746  return association;
747
748  else {
749      auto &refs = refs_it->second;
750      auto it = refs.find(key);
751      if (it != refs.end()) {
752          association.swap(it->second);
753          refs.erase(it);
754          if (refs.size() == 0) {
755              associations.erase(refs_it);
756          }
757      }
758  }
759
760  return association;
761
762  else {
763      auto &refs = refs_it->second;
764      auto it = refs.find(key);
765      if (it != refs.end()) {
766          association.swap(it->second);
767          refs.erase(it);
768          if (refs.size() == 0) {
769              associations.erase(refs_it);
770          }
771      }
772  }
773
774  return association;
775
776  else {
777      auto &refs = refs_it->second;
778      auto it = refs.find(key);
779      if (it != refs.end()) {
780          association.swap(it->second);
781          refs.erase(it);
782          if (refs.size() == 0) {
783              associations.erase(refs_it);
784          }
785      }
786  }
787
788  return association;
789
790  else {
791      auto &refs = refs_it->second;
792      auto it = refs.find(key);
793      if (it != refs.end()) {
794          association.swap(it->second);
795          refs.erase(it);
796          if (refs.size() == 0) {
797              associations.erase(refs_it);
798          }
799      }
800  }
801
802  return association;
803
804  else {
805      auto &refs = refs_it->second;
806      auto it = refs.find(key);
807      if (it != refs.end()) {
808          association.swap(it->second);
809          refs.erase(it);
810          if (refs.size() == 0) {
811              associations.erase(refs_it);
812          }
813      }
814  }
815
816  return association;
817
818  else {
819      auto &refs = refs_it->second;
820      auto it = refs.find(key);
821      if (it != refs.end()) {
822          association.swap(it->second);
823          refs.erase(it);
824          if (refs.size() == 0) {
825              associations.erase(refs_it);
826          }
827      }
828  }
829
830  return association;
831
832  else {
833      auto &refs = refs_it->second;
834      auto it = refs.find(key);
835      if (it != refs.end()) {
836          association.swap(it->second);
837          refs.erase(it);
838          if (refs.size() == 0) {
839              associations.erase(refs_it);
840          }
841      }
842  }
843
844  return association;
845
846  else {
847      auto &refs = refs_it->second;
848      auto it = refs.find(key);
849      if (it != refs.end()) {
850          association.swap(it->second);
851          refs.erase(it);
852          if (refs.size() == 0) {
853              associations.erase(refs_it);
854          }
855      }
856  }
857
858  return
```

```

49
50             }
51         }
52     }
53 }
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