Class的本质

我们知道不管是类对象还是元类对象，类型都是Class，class和mete-class的底层都是objc_class结构体的指针，内存中就是结构体，本章来探寻Class的本质。

```
Class objectClass = [NSObject class];
Class objectMetaClass = object_getClass([NSObject class]);
```

点击Class来到内部，我们可以发现

```
typedef struct objc_class *Class;
```

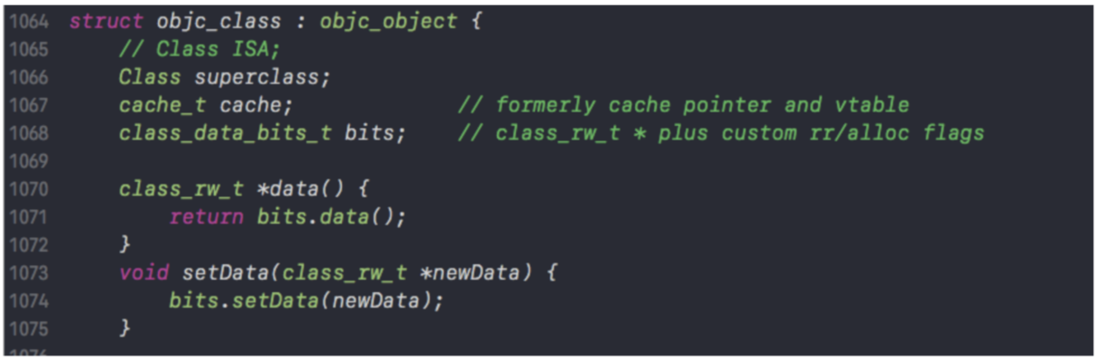
Class对象其实是一个指向objc_class结构体的指针。因此我们可以说类对象或元类对象在内存中其实就是objc_class结构体。

我们来到objc_class内部，可以看到这段在底层原理中经常出现的代码。

```
struct objc_class {
    Class _Nonnull isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
    Class _Nullable super_class
OBJC2_UNAVAILABLE;
    const char * _Nonnull name
OBJC2_UNAVAILABLE;
    long version
OBJC2_UNAVAILABLE;
    long info
OBJC2_UNAVAILABLE;
    long instance_size
OBJC2_UNAVAILABLE;
    struct objc_ivar_list * _Nullable ivars
OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable * _Nullable methodLists
OBJC2_UNAVAILABLE;
    struct objc_cache * _Nonnull cache
OBJC2_UNAVAILABLE;
```

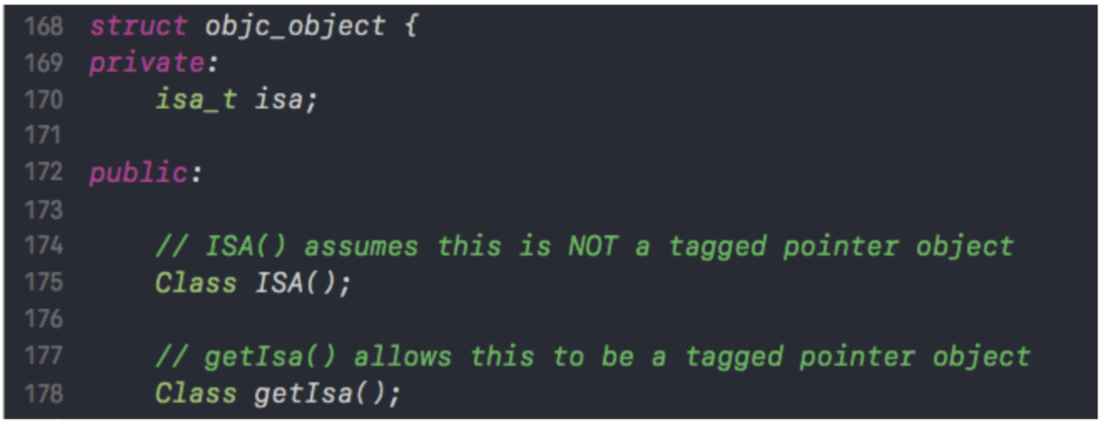
```
    struct objc_protocol_list * _Nullable protocols
OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;
/* Use `Class` instead of `struct objc_class *` */
```

这部分代码相信在文章中很常见，但是`OBJC2_UNAVAILABLE;`说明这些代码已经不在使用了。那么目前objc_class的结构是什么样的呢？我们通过objc源码中去查找objc_class结构体的内容。



```
struct objc_class : objc_object {
    // Class ISA;
    Class superclass;
    cache_t cache;             // formerly cache pointer and vtable
    class_data_bits_t bits;    // class_rw_t * plus custom rr/alloc flags

    class_rw_t *data() {
        return bits.data();
    }
    void setData(class_rw_t *newData) {
        bits.setData(newData);
    }
```

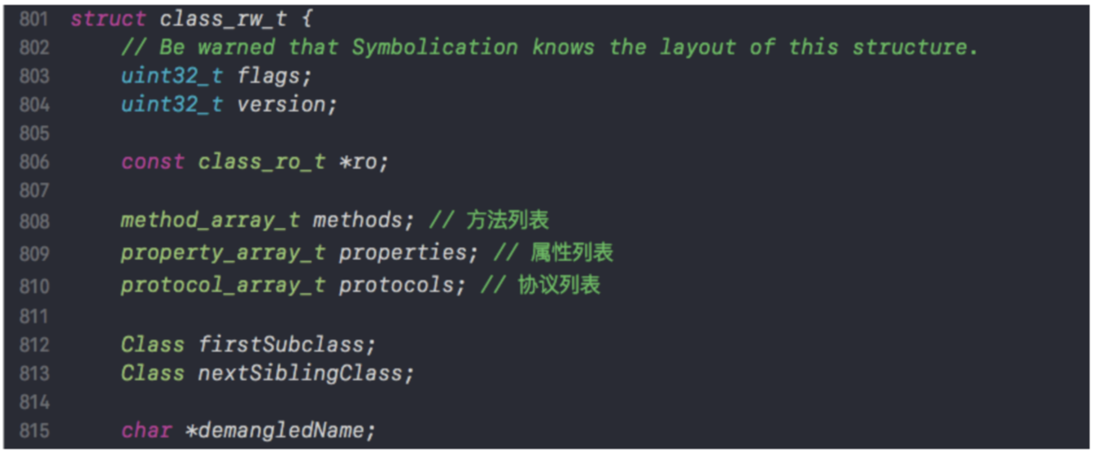
我们发现这个结构体继承 objc_object 并且结构体内有一些函数，因为这是c++结构体，在c上做了扩展，因此结构体中可以包含函数。我们来到objc_object内，截取部分代码



```
struct objc_object {
private:
    isa_t isa;

public:

    // ISA() assumes this is NOT a tagged pointer object
    Class ISA();

    // getIsa() allows this to be a tagged pointer object
    Class getIsa();
```

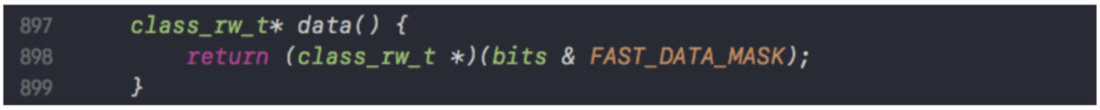
我们发现objc_object中有一个isa指针，那么objc_class继承objc_object，也就同样拥有一个isa指针

那么我们之前了解到的，类中存储的类的成员变量信息，实例方法，属性名等这些信息在哪里呢。我们来到class_rw_t中，截取部分代码，我们发现class_rw_t中存储着方法列表，属性列表，协议列表等内容。



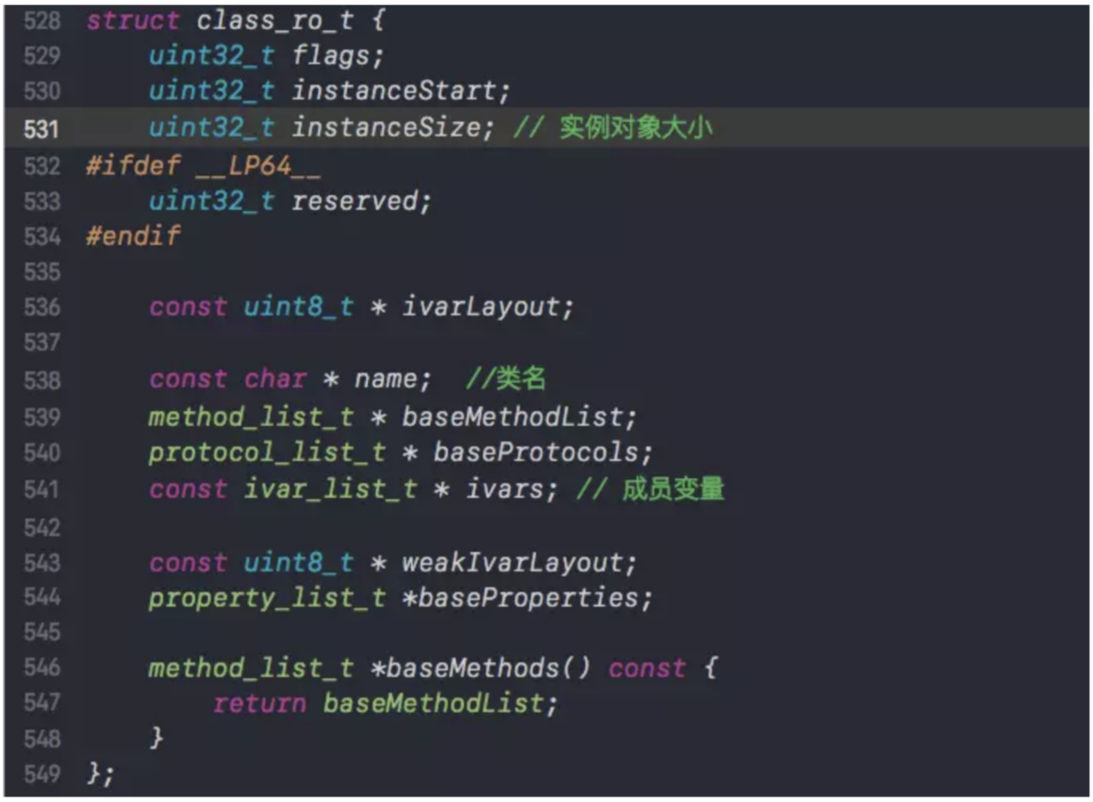
```
struct class_rw_t {
    // Be warned that Symbolication knows the layout of this structure.
    uint32_t flags;
    uint32_t version;

    const class_ro_t *ro;

    method_array_t methods; // 方法列表
    property_array_t properties; // 属性列表
    protocol_array_t protocols; // 协议列表

    Class firstSubclass;
    Class nextSiblingClass;

    char *demangledName;
```

而class_rw_t是通过bits调用data方法得来的，我们来到data方法内部实现。我们可以看到，data函数内部仅仅对bits进行&FAST_DATA_MASK操作



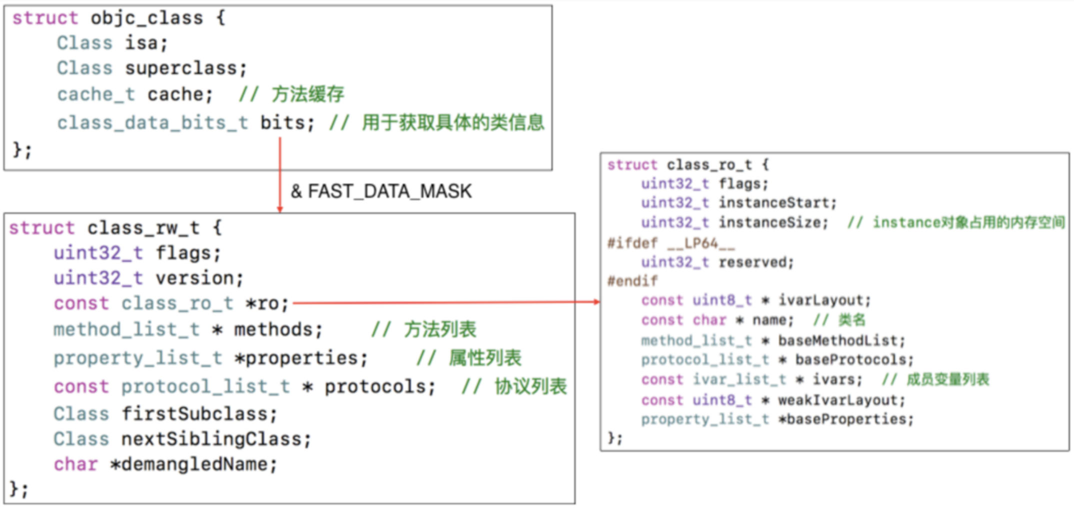
```
    class_rw_t* data() {
        return (class_rw_t *)(bits & FAST_DATA_MASK);
    }
```

而成员变量信息则是存储在class_ro_t内部中的，我们来到class_ro_t内查看。



```
struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize; // 实例对象大小
#ifdef __LP64__
    uint32_t reserved;
#endif

    const uint8_t * ivarLayout;

    const char * name;  //类名
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
    const ivar_list_t * ivars; // 成员变量

    const uint8_t * weakIvarLayout;
    property_list_t *baseProperties;

    method_list_t *baseMethods() const {
        return baseMethodList;
    }
};
```

最后总结通过一张图进行总结



```
struct objc_class {
    Class isa;
    Class superclass;
    cache_t cache;   // 方法缓存
    class_data_bits_t bits;  // 用于获取具体的类信息
};
```

```
struct class_rw_t {
    uint32_t flags;
    uint32_t version;
    const class_ro_t *ro;
    method_list_t * methods;     // 方法列表
    property_list_t *properties;     // 属性列表
    const protocol_list_t * protocols;   // 协议列表
    Class firstSubclass;
    Class nextSiblingClass;
    char *demangledName;
};
```

```
struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;  // instance对象占用的内存空间
#ifdef __LP64__
    uint32_t reserved;
#endif
    const uint8_t * ivarLayout;
    const char * name;  // 类名
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
    const ivar_list_t * ivars;  // 成员变量列表
    const uint8_t * weakIvarLayout;
    property_list_t *baseProperties;
};
```

如何证明上述内容是正确的。

我们可以自定义一个结构体，如果我们自己写的结构和objc_class真实结构是一样的，那么当我们强制转化的时候，就会一一对应的赋值。此时我们就可以拿到结构

体内部的信息。

下列代码是我们仿照objc_class结构体，提取其中需要使用到的信息，自定义的一个结构体。

```
#import <Foundation/Foundation.h>

#ifndef XXClassInfo_h
#define XXClassInfo_h

# if __arm64__
#   define ISA_MASK        0x0000000ffffffff8ULL
# elif __x86_64__
#   define ISA_MASK        0x00007ffffffffff8ULL
# endif

#if __LP64__
typedef uint32_t mask_t;
#else
typedef uint16_t mask_t;
#endif
typedef uintptr_t cache_key_t;

struct bucket_t {
    cache_key_t _key;
    IMP _imp;
};

struct cache_t {
    bucket_t *_buckets;
    mask_t _mask;
    mask_t _occupied;
};

struct entsize_list_tt {
    uint32_t entsizeAndFlags;
    uint32_t count;
};

struct method_t {
    SEL name;
    const char *types;
    IMP imp;
};

struct method_list_t : entsize_list_tt {
    method_t first;
```

```
};

struct ivar_t {
    int32_t *offset;
    const char *name;
    const char *type;
    uint32_t alignment_raw;
    uint32_t size;
};

struct ivar_list_t : entsize_list_tt {
    ivar_t first;
};

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

struct property_list_t : entsize_list_tt {
    property_t first;
};

struct chained_property_list {
    chained_property_list *next;
    uint32_t count;
    property_t list[0];
};

typedef uintptr_t protocol_ref_t;
struct protocol_list_t {
    uintptr_t count;
    protocol_ref_t list[0];
};

struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;  // instance对象占用的内存空间
#ifdef __LP64__
    uint32_t reserved;
#endif
    const uint8_t * ivarLayout;
    const char * name;  // 类名
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
    const ivar_list_t * ivars;  // 成员变量列表
    const uint8_t * weakIvarLayout;
    property_list_t *baseProperties;
};

struct class_rw_t {
    uint32_t flags;
    uint32_t version;
    const class_ro_t *ro;
    method_list_t * methods;    // 方法列表
    property_list_t *properties;    // 属性列表
    const protocol_list_t * protocols;  // 协议列表
    Class firstSubclass;
    Class nextSiblingClass;
    char *demangledName;
};

#define FAST_DATA_MASK          0x00007ffffffffff8UL
struct class_data_bits_t {
    uintptr_t bits;
public:
    class_rw_t* data() { // 提供data()方法进行 & FAST_DATA_MASK 操作
        return (class_rw_t *)(bits & FAST_DATA_MASK);
    }
};

/* OC对象 */
struct xx_objc_object {
    void *isa;
};

/* 类对象 */
struct xx_objc_class : xx_objc_object {
    Class superclass;
    cache_t cache;
    class_data_bits_t bits;
public:
    class_rw_t* data() {
        return bits.data();
    }

    xx_objc_class* metaClass() { // 提供metaClass函数，获取元类对象
// 上一篇我们讲解过，isa指针需要经过一次 & ISA_MASK操作之后才得到真正的地址
        return (xx_objc_class *)((long long)isa & ISA_MASK);
    }
};

#endif /* XXClassInfo_h */
```

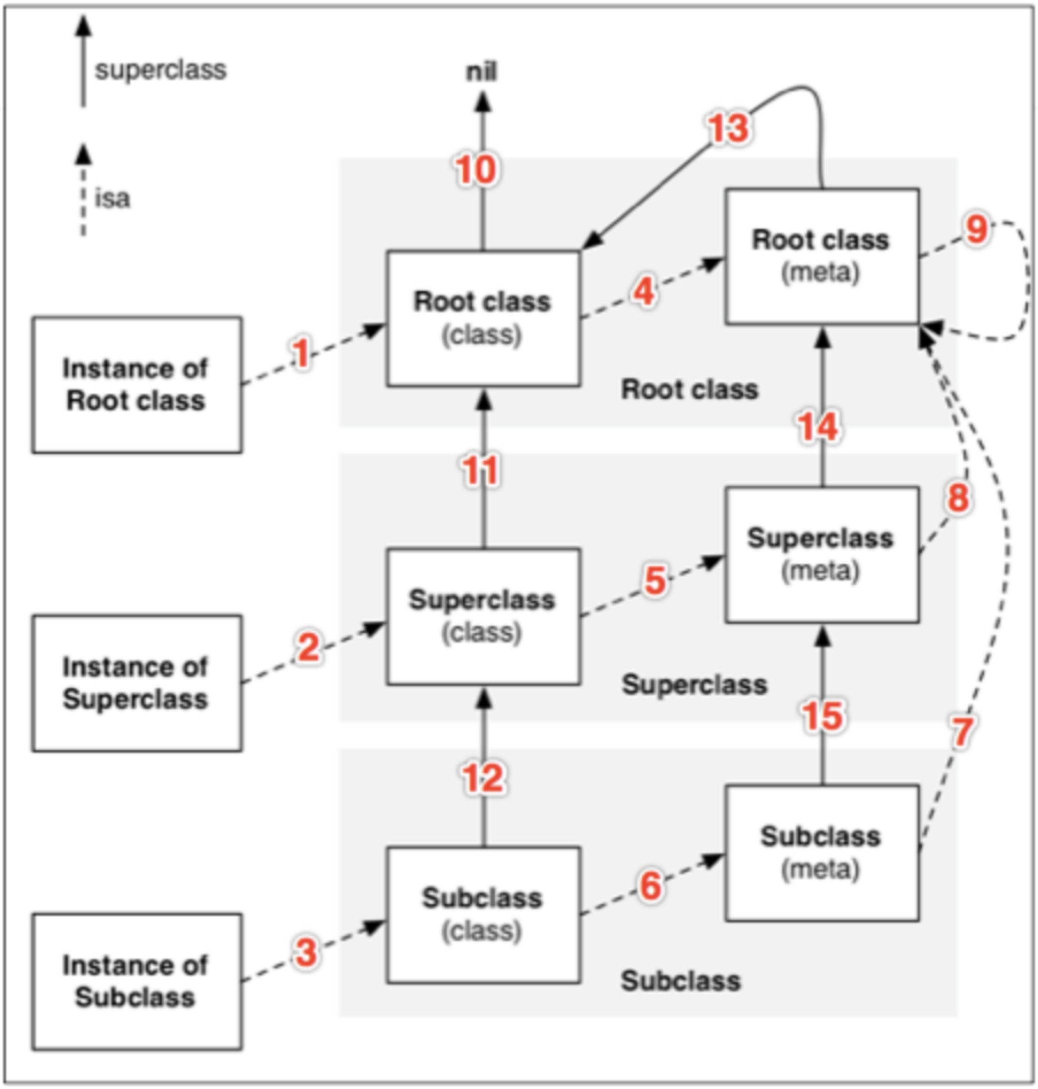
接下来我们将自己定义的类强制转化为我们自定义的精简的class结构体类型。

```
#import <Foundation/Foundation.h>
#import <objc/runtime.h>
#import "XXClassInfo.h"

/* Person */
@interface Person : NSObject <NSCopying>
{
    @public
    int _age;
}
@property (nonatomic, assign) int height;
- (void)personMethod;
+ (void)personClassMethod;
@end

@implementation Person
- (void)personMethod {}
+ (void)personClassMethod {}
@end

/* Student */
@interface Student : Person <NSCoding>
{
    @public
    int _no;
}

@property (nonatomic, assign) int score;
- (void)studentMethod;
+ (void)studentClassMethod;
@end

@implementation Student
- (void)studentMethod {}
+ (void)studentClassMethod {}
@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSObject *object = [[NSObject alloc] init];
        Person *person = [[Person alloc] init];
        Student *student = [[Student alloc] init];

        xx_objc_class *objectClass = (__bridge xx_objc_class *)
[object class];
```

```
        xx_objc_class *personClass = (__bridge xx_objc_class *)
[person class];
        xx_objc_class *studentClass = (__bridge xx_objc_class *)
[student class];

        xx_objc_class *objectMetaClass = objectClass->metaClass();
        xx_objc_class *personMetaClass = personClass->metaClass();
        xx_objc_class *studentMetaClass = studentClass-
>metaClass();

        class_rw_t *objectClassData = objectClass->data();
        class_rw_t *personClassData = personClass->data();
        class_rw_t *studentClassData = studentClass->data();

        class_rw_t *objectMetaClassData = objectMetaClass->data();
        class_rw_t *personMetaClassData = personMetaClass->data();
        class_rw_t *studentMetaClassData = studentMetaClass-
>data();

        // 0x00007ffffffffff8
        NSLog(@"%p %p %p %p %p %p",  objectClassData,
personClassData, studentClassData,
              objectMetaClassData, personMetaClassData,
studentMetaClassData);

    return 0;
}
```

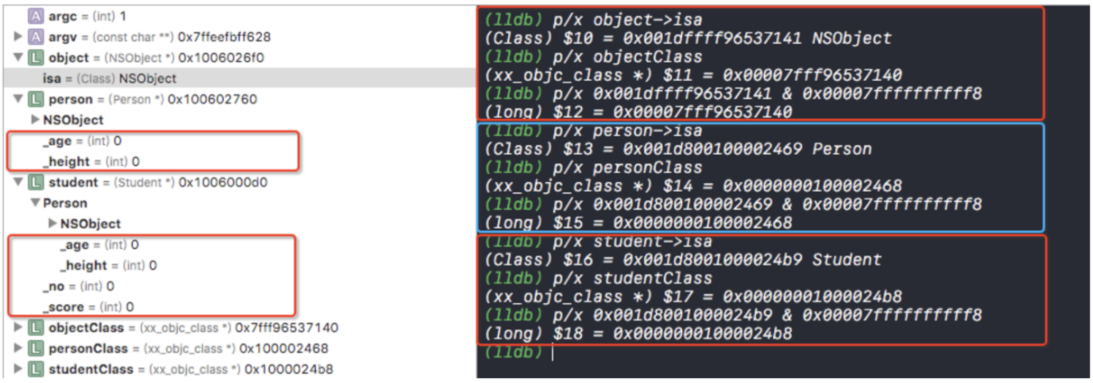
通过打断点，我们可以看到class内部信息。

至此，我们再次拿出那张经典的图，挨个分析图中isa指针和superclass指针的指向



instance对象

首先我们来看instance对象，我们通过上一篇文章知道，instance对象中存储着isa指针和其他成员变量，并且instance对象的isa指针是指向其**类对象**地址的。我们首先分析上述代码中我们创建的object，person，student三个instance对象与其相对应的类对象objectClass，personClass，studentClass。



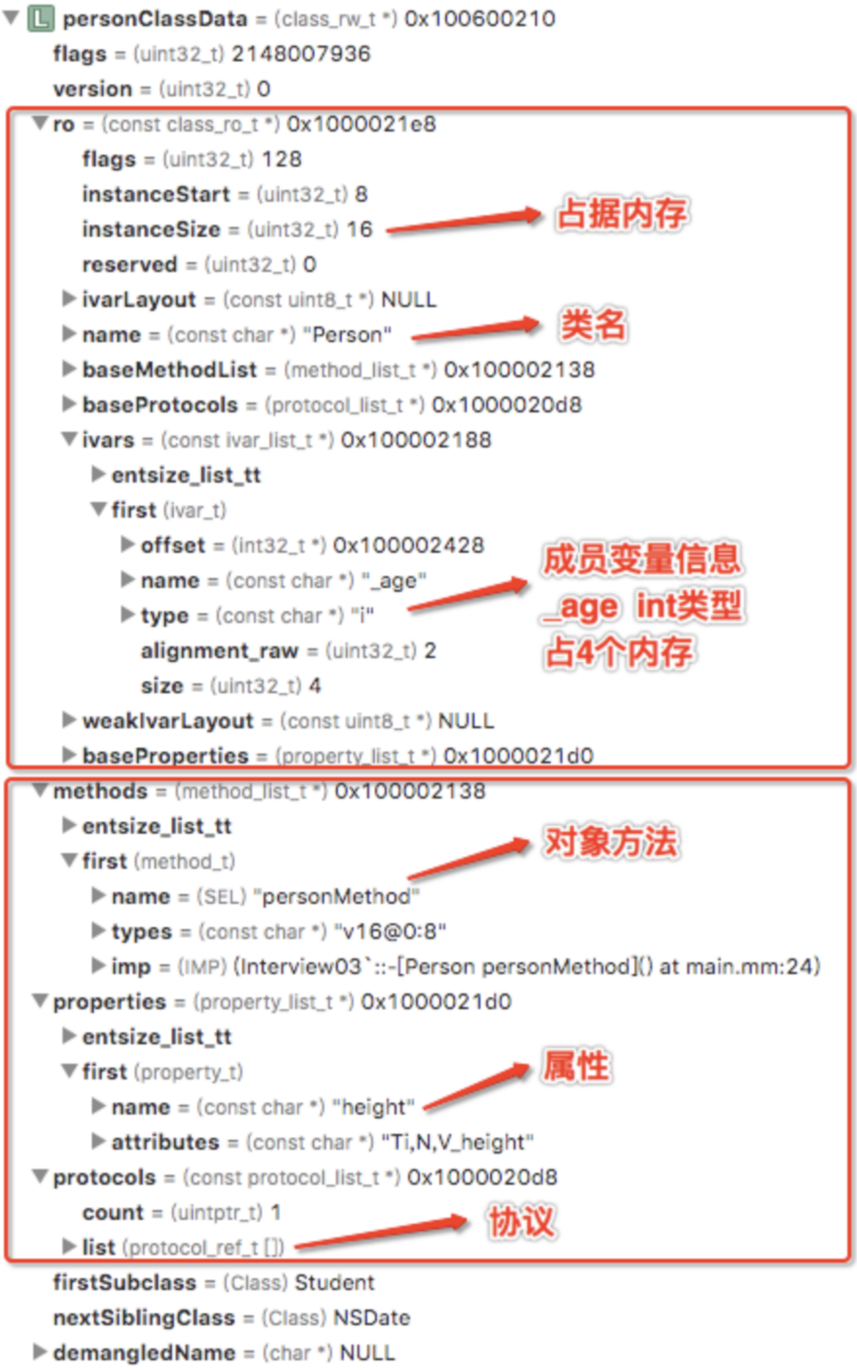
```
argc = (int) 1
▶ argv = (const char **) 0x7ffeefbff628
▼ object = (NSObject *) 0x1006026f0
    isa = (Class) NSObject
▼ person = (Person *) 0x100602760
  ▶ NSObject
    _age = (int) 0
    _height = (int) 0
▼ student = (Student *) 0x1006000d0
  ▼ Person
    ▶ NSObject
      _age = (int) 0
      _height = (int) 0
    _no = (int) 0
    _score = (int) 0
▶ objectClass = (xx_objc_class *) 0x7fff96537140
▶ personClass = (xx_objc_class *) 0x100002468
▶ studentClass = (xx_objc_class *) 0x1000024b8
```

```
(lldb) p/x object->isa
(Class) $10 = 0x001dffff96537141 NSObject
(lldb) p/x objectClass
(xx_objc_class *) $11 = 0x00007fff96537140
(lldb) p/x 0x001dffff96537141 & 0x00007ffffffffff8
(long) $12 = 0x00007fff96537140
(lldb) p/x person->isa
(Class) $13 = 0x001d800100002469 Person
(lldb) p/x personClass
(xx_objc_class *) $14 = 0x0000000100002468
(lldb) p/x 0x001d800100002469 & 0x00007ffffffffff8
(long) $15 = 0x0000000100002468
(lldb) p/x student->isa
(Class) $16 = 0x001d8001000024b9 Student
(lldb) p/x studentClass
(xx_objc_class *) $17 = 0x00000001000024b8
(lldb) p/x 0x001d8001000024b9 & 0x00007ffffffffff8
(long) $18 = 0x00000001000024b8
(lldb) |
```

从上图中我们可以发现instance对象中确实存储了isa指针和其成员变量，同时将instance对象的isa指针经过&运算之后计算出的地址确实是其相应类对象的内存地址。由此我们证明isa，superclass指向图中的1，2，3号线。

class对象

接着我们来看class对象，同样通过上一篇文章，我们明确class对象中存储着isa指针，superclass指针，以及类的属性信息，类的成员变量信息，类的对象方法，和类的协议信息，而通过上面对object源码的分析，我们知道这些信息存储在class对象的class_rw_t中，我们通过强制转化来窥探其中的内容。如下图



```
▼ L personClassData = (class_rw_t *) 0x100600210
    flags = (uint32_t) 2148007936
    version = (uint32_t) 0
  ▼ ro = (const class_ro_t *) 0x1000021e8
      flags = (uint32_t) 128
      instanceStart = (uint32_t) 8
      instanceSize = (uint32_t) 16
      reserved = (uint32_t) 0
    ▶ ivarLayout = (const uint8_t *) NULL
    ▶ name = (const char *) "Person"
    ▶ baseMethodList = (method_list_t *) 0x100002138
    ▶ baseProtocols = (protocol_list_t *) 0x1000020d8
    ▼ ivars = (const ivar_list_t *) 0x100002188
      ▶ entsize_list_tt
      ▼ first (ivar_t)
        ▶ offset = (int32_t *) 0x100002428
        ▶ name = (const char *) "_age"
        ▶ type = (const char *) "i"
          alignment_raw = (uint32_t) 2
          size = (uint32_t) 4
    ▶ weakIvarLayout = (const uint8_t *) NULL
    ▶ baseProperties = (property_list_t *) 0x1000021d0
  ▼ methods = (method_list_t *) 0x100002138
    ▶ entsize_list_tt
    ▼ first (method_t)
      ▶ name = (SEL) "personMethod"
      ▶ types = (const char *) "v16@0:8"
      ▶ imp = (IMP) (Interview03`::-[Person personMethod]() at main.mm:24)
  ▼ properties = (property_list_t *) 0x1000021d0
    ▶ entsize_list_tt
    ▼ first (property_t)
      ▶ name = (const char *) "height"
      ▶ attributes = (const char *) "Ti,N,V_height"
  ▼ protocols = (const protocol_list_t *) 0x1000020d8
      count = (uintptr_t) 1
    ▶ list (protocol_ref_t [])
    firstSubclass = (Class) Student
    nextSiblingClass = (Class) NSDate
  ▶ demangledName = (char *) NULL
```

占据内存

类名

成员变量信息
_age int类型
占4个内存

对象方法

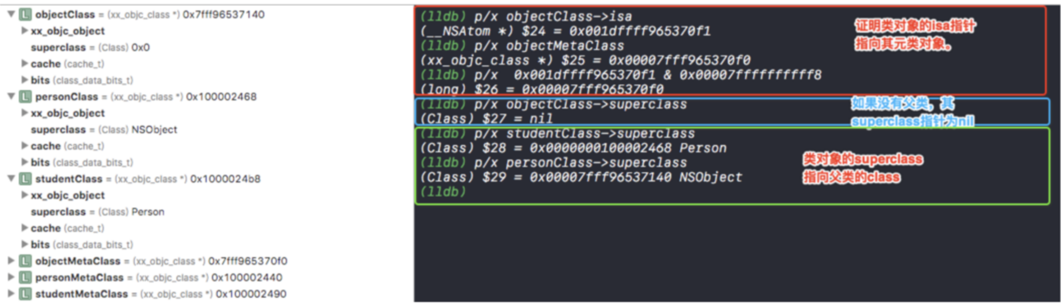
属性

协议

上图中我们通过模拟对person类对象调用.data函数，即对bits进行&FAST_DATA_MASK(0x00007ffffffffff8UL)运算，并转化为class_rw_t。即上图中的personClassData。其中我们发现成员变量信息，对象方法，属性等信息只显示first第一个，如果想要拿到更多的需要通过代码将指针后移获取。而上图中的instaceSize = 16也同person对象中isa指针8个字节+_age4个字节+_height4个字节

相对应起来。这里不在展开对objectClassData及studentClassData进行分析，基本内容同personClassData相同。

那么类对象中的isa指针和superclass指针的指向是否如那张经典的图示呢？我们来验证一下。



```
▼ objectClass = (xx_objc_class *) 0x7fff96537140
  ▶ xx_objc_object
    superclass = (Class) 0x0
  ▶ cache (cache_t)
  ▶ bits (class_data_bits_t)
▼ personClass = (xx_objc_class *) 0x100002468
  ▶ xx_objc_object
    superclass = (Class) NSObject
  ▶ cache (cache_t)
  ▶ bits (class_data_bits_t)
▼ studentClass = (xx_objc_class *) 0x1000024b8
  ▶ xx_objc_object
    superclass = (Class) Person
  ▶ cache (cache_t)
  ▶ bits (class_data_bits_t)
▶ objectMetaClass = (xx_objc_class *) 0x7fff965370f0
▶ personMetaClass = (xx_objc_class *) 0x100002440
▶ studentMetaClass = (xx_objc_class *) 0x100002490
```

```
(lldb) p/x objectClass->isa
(__NSAtom *) $24 = 0x001dffff965370f1
(lldb) p/x objectMetaClass
(xx_objc_class *) $25 = 0x00007fff965370f0
(lldb) p/x  0x001dffff965370f1 & 0x00007ffffffffff8
(long) $26 = 0x00007fff965370f0
(lldb) p/x objectClass->superclass
(Class) $27 = nil
(lldb) p/x studentClass->superclass
(Class) $28 = 0x0000000100002468 Person
(lldb) p/x personClass->superclass
(Class) $29 = 0x00007fff96537140 NSObject
(lldb)
```

证明类对象的isa指针指向其元类对象。

如果没有父类，其superclass指针为nil

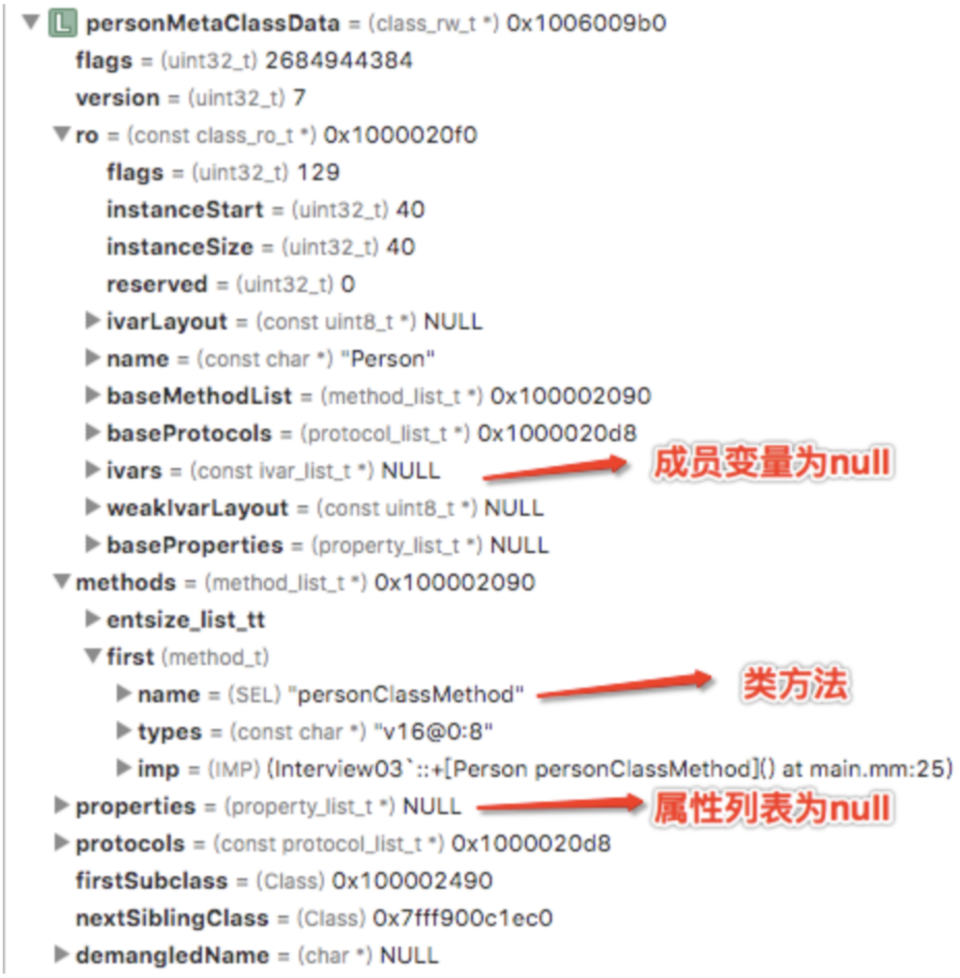
类对象的superclass指向父类的class

通过上图中的内存地址的分析，由此我们证明isa，superclass指向图中，isa指针的4，5，6号线，以及superclass指针的10，11，12号线。

meta-class对象

最后我们来看meta-class元类对象，上文提到meta-class中存储着isa指针，superclass指针，以及类的类方法信息。同时我们知道meta-class元类对象与class类对象，具有相同的结构，只不过存储的信息不同，并且元类对象的isa指针指向基类的元类对象，基类的元类对象的isa指针指向自己。元类对象的superclass指针指向其父类的元类对象，基类的元类对象的superclass指针指向其类对象。

与class对象相同，我们同样通过模拟对person元类对象调用.data函数，即对bits进行&FAST_DATA_MASK(0x00007ffffffffff8UL)运算，并转化为class_rw_t。



```
▼ L personMetaClassData = (class_rw_t *) 0x1006009b0
      flags = (uint32_t) 2684944384
      version = (uint32_t) 7
    ▼ ro = (const class_ro_t *) 0x1000020f0
        flags = (uint32_t) 129
        instanceStart = (uint32_t) 40
        instanceSize = (uint32_t) 40
        reserved = (uint32_t) 0
      ▶ ivarLayout = (const uint8_t *) NULL
      ▶ name = (const char *) "Person"
      ▶ baseMethodList = (method_list_t *) 0x100002090
      ▶ baseProtocols = (protocol_list_t *) 0x1000020d8
      ▶ ivars = (const ivar_list_t *) NULL
      ▶ weakIvarLayout = (const uint8_t *) NULL
      ▶ baseProperties = (property_list_t *) NULL
    ▼ methods = (method_list_t *) 0x100002090
      ▶ entsize_list_tt
      ▼ first (method_t)
        ▶ name = (SEL) "personClassMethod"
        ▶ types = (const char *) "v16@0:8"
        ▶ imp = (IMP) (Interview03`::+[Person personClassMethod]() at main.mm:25)
    ▶ properties = (property_list_t *) NULL
    ▶ protocols = (const protocol_list_t *) 0x1000020d8
      firstSubclass = (Class) 0x100002490
      nextSiblingClass = (Class) 0x7fff900c1ec0
    ▶ demangledName = (char *) NULL
```

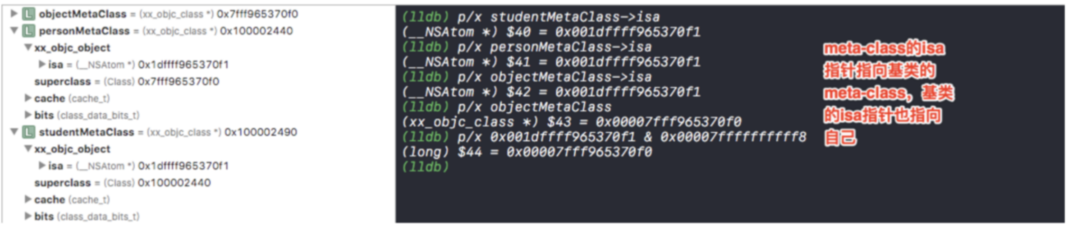
成员变量为null

类方法

属性列表为null

首先我们可以看到结构同personClassData相同，并且成员变量及属性列表等信息为空，而methods中存储着类方法personClassMethod。

接着来验证isa及superclass指针的指向是否同上图序号标注一样。

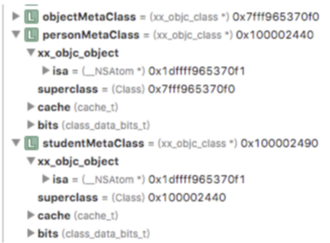


```
▶ L objectMetaClass = (xx_objc_class *) 0x7fff965370f0
▼ L personMetaClass = (xx_objc_class *) 0x100002440
    ▼ xx_objc_object
      ▶ isa = (__NSAtom *) 0x1dffff965370f1
      superclass = (Class) 0x7fff965370f0
    ▶ cache (cache_t)
    ▶ bits (class_data_bits_t)
▼ L studentMetaClass = (xx_objc_class *) 0x100002490
    ▼ xx_objc_object
      ▶ isa = (__NSAtom *) 0x1dffff965370f1
      superclass = (Class) 0x100002440
    ▶ cache (cache_t)
    ▶ bits (class_data_bits_t)
```

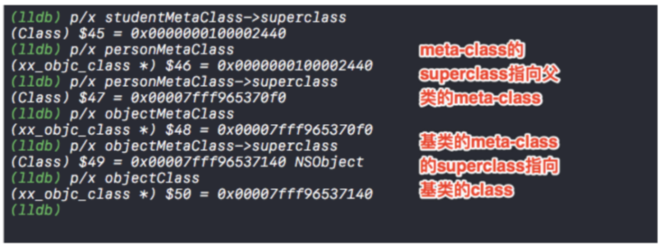
```
(lldb) p/x studentMetaClass->isa
(__NSAtom *) $40 = 0x001dffff965370f1
(lldb) p/x personMetaClass->isa
(__NSAtom *) $41 = 0x001dffff965370f1
(lldb) p/x objectMetaClass->isa
(__NSAtom *) $42 = 0x001dffff965370f1
(lldb) p/x objectMetaClass
(xx_objc_class *) $43 = 0x00007fff965370f0
(lldb) p/x 0x001dffff965370f1 & 0x00007ffffffffff8
(long) $44 = 0x00007fff965370f0
(lldb)
```

meta-class的isa指针指向基类的meta-class，基类的isa指针也指向自己

上图中通过地址证明meta-class的isa指向基类的meta-class，基类的isa指针也指向自己。



```
▶ objectMetaClass = (xx_objc_class *) 0x7fff965370f0
▼ personMetaClass = (xx_objc_class *) 0x100002440
  ▼ xx_objc_object
    ▶ isa = (__NSAtom *) 0x1dffff965370f1
    superclass = (Class) 0x7fff965370f0
  ▶ cache (cache_t)
  ▶ bits (class_data_bits_t)
▼ studentMetaClass = (xx_objc_class *) 0x100002490
  ▼ xx_objc_object
    ▶ isa = (__NSAtom *) 0x1dffff965370f1
    superclass = (Class) 0x100002440
  ▶ cache (cache_t)
  ▶ bits (class_data_bits_t)
```



```
(lldb) p/x studentMetaClass->superclass
(Class) $45 = 0x0000000100002440
(lldb) p/x personMetaClass
(xx_objc_class *) $46 = 0x0000000100002440
(lldb) p/x personMetaClass->superclass
(Class) $47 = 0x00007fff965370f0
(lldb) p/x objectMetaClass
(xx_objc_class *) $48 = 0x00007fff965370f0
(lldb) p/x objectMetaClass->superclass
(Class) $49 = 0x00007fff96537140 NSObject
(lldb) p/x objectClass
(xx_objc_class *) $50 = 0x00007fff96537140
(lldb)
```

meta-class的superclass指向父类的meta-class

基类的meta-class的superclass指向基类的class

上图中通过地址证明meta-class的superclass指向父类的meta-class，基类的meta-class的superclass指向基类的class类。