isa的本质

在学习Runtime之前首先需要对isa的本质有一定的了解，这样之后学习Runtime会更便于理解。 回顾OC对象的本质，**每个OC对象都含有一个isa指针，__arm64__之前，isa仅仅是一个指针，保存着对象或类对象内存地址，在__arm64__架构之后，apple对isa进行了优化，变成了一个共用体（union）结构，同时使用位域来存储更多的信息。** 我们知道OC对象的isa指针并不是直接指向类对象或者元类对象，而是需要**&ISA_MASK通过位运算才能获取到类对象或者元类对象的地址**。今天来探寻一下为什么需要&ISA_MASK才能获取到类对象或者元类对象的地址，以及这样的好处。 首先在源码中找到isa指针，看一下isa指针的本质。

```
// 截取objc_object内部分代码
struct objc_object {
private:
    isa_t isa;
}
isa指针其实是一个isa_t类型的共用体，来到isa_t内部查看其结构
// 精简过的isa_t共用体
union isa_t
{
    isa_t() { }
    isa_t(uintptr_t value) : bits(value) { }

    Class cls;
    uintptr_t bits;

#if SUPPORT_PACKED_ISA
# if __arm64__
#   define ISA_MASK        0x0000000ffffffff8ULL
#   define ISA_MAGIC_MASK  0x000003f000000001ULL
#   define ISA_MAGIC_VALUE 0x000001a000000001ULL
    struct {
        uintptr_t nonpointer        : 1;
        uintptr_t has_assoc         : 1;
        uintptr_t has_cxx_dtor      : 1;
        uintptr_t shiftcls          : 33; // MACH_VM_MAX_ADDRESS
0x1000000000
        uintptr_t magic             : 6;
        uintptr_t weakly_referenced : 1;
        uintptr_t deallocating      : 1;
        uintptr_t has_sidetable_rc  : 1;
        uintptr_t extra_rc          : 19;
```

```
    #       define RC_ONE   (1ULL<<45)
    #       define RC_HALF  (1ULL<<18)
    };

# elif __x86_64__
#   define ISA_MASK        0x00007ffffffffff8ULL
#   define ISA_MAGIC_MASK  0x001f800000000001ULL
#   define ISA_MAGIC_VALUE 0x001d800000000001ULL
    struct {
        uintptr_t nonpointer        : 1;
        uintptr_t has_assoc         : 1;
        uintptr_t has_cxx_dtor      : 1;
        uintptr_t shiftcls          : 44; // MACH_VM_MAX_ADDRESS
0x7ffffe00000
        uintptr_t magic             : 6;
        uintptr_t weakly_referenced : 1;
        uintptr_t deallocating      : 1;
        uintptr_t has_sidetable_rc  : 1;
        uintptr_t extra_rc          : 8;
#       define RC_ONE   (1ULL<<56)
#       define RC_HALF  (1ULL<<7)
    };

# else
#   error unknown architecture for packed isa
# endif
#endif
```

上述源码中isa_t是union类型，union表示共用体。可以看到共用体中有一个结构体，结构体内部分别定义了一些变量，变量后面的值代表的是该变量占用多少个字节，也就是位域技术。 共用体：在进行某些算法的C语言编程的时候，需要使几种不同类型的变量存放到同一段内存单元中。也就是使用覆盖技术，几个变量互相覆盖。这种几个不同的变量共同占用一段内存的结构，在C语言中，被称作“共用体”类型结构，简称共用体。 接下来使用共用体的方式来深入的了解apple为什么要使用共用体，以及使用共用体的好处。 探寻过程 接下来使用代码来模仿底层的做法，创建一个person类并含有三个BOOL类型的成员变量。

```
@interface Person : NSObject
@property (nonatomic, assign, getter = isTall) BOOL tall;
@property (nonatomic, assign, getter = isRich) BOOL rich;
@property (nonatomic, assign, getter = isHansome) BOOL handsome;
@end
int main(int argc, const char * argv[]) {
    @autoreleasepool {
```

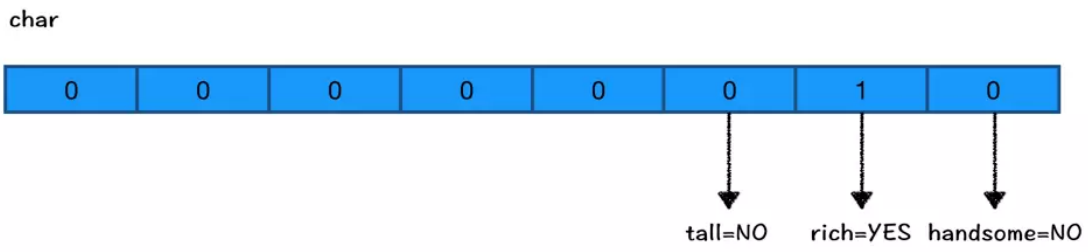
```
        NSLog(@"%zd", class_getInstanceSize([Person class]));
    }
    return 0;
}
```

```
// 打印内容 // Runtime - union探寻[52235:3160607] 16
```

上述代码中Person含有3个BOOL类型的属性，打印Person类对象占据内存空间为16，也就是(isa指针 = 8) + (BOOL tall = 1) + (BOOL rich = 1) + (BOOL handsome = 1) = 13。因为内存对齐原则所以Person类对象占据内存空间为16。 上面提到过共用体中变量可以相互覆盖，可以使几个不同的变量存放到同一段内存单元中，可以很大程度上节省内存空间。 那么我们知道BOOL值只有两种情况 0 或者 1，但是却占据了一个字节的内存空间，而一个内存空间中有8个二进制位，并且二进制只有 0 或者 1 。那么是否可以使用1个二进制位来表示一个BOOL值，也就是说3个BOOL值最终只使用3个二进制位，也就是一个内存空间即可呢？如何实现这种方式？ 首先如果使用这种方式需要自己写方法声明与实现，不可以写属性，因为一旦写属性，系统会自动帮我们添加成员变量。 另外想要将三个BOOL值存放在一个字节中，我们可以添加一个char类型的成员变量，char类型占据一个字节内存空间，也就是8个二进制位。可以使用其中最后三个二进制位来存储3个BOOL值。

```
@interface Person()
{
    char _tallRichHandsome;
}
```

例如_tallRichHansome的值为 0b 0000 0010 ，那么只使用8个二进制位中的最后3个，分别为其赋值0或者1来代表tall、rich、handsome的值。如下图所示



那么现在面临的问题就是如何取出8个二进制位中的某一位的值，或者为某一位赋值呢?

取值

首先来看一下取值，假如char类型的成员变量中存储的二进制为0b 0000 0010如果想将倒数第2位的值也就是rich的值取出来，可以使用&进行按位与运算进而去除相应位置的值。

&：按位与，同真为真，其他都为假。

```
// 示例
// 取出倒数第三位 tall
  0000 0010
& 0000 0100
------------
  0000 0000  // 取出倒数第三位的值为0，其他位都置为0

// 取出倒数第二位 rich
  0000 0010
& 0000 0010
------------
  0000 0010 // 取出倒数第二位的值为1，其他位都置为0
```

按位与可以用来取出特定的位，想取出哪一位就将那一位置为1，其他为都置为0，然后同原数据进行按位与计算，即可取出特定的位。

那么此时可以将get方法写成如下方式

```
#define TallMask 0b00000100 // 4
#define RichMask 0b00000010 // 2
#define HandsomeMask 0b00000001 // 1

- (BOOL)tall
{
    return !!(_tallRichHandsome & TallMask);
}
- (BOOL)rich
{
    return !!(_tallRichHandsome & RichMask);
}
- (BOOL)handsome
{
    return !!(_tallRichHandsome & HandsomeMask);
}
```

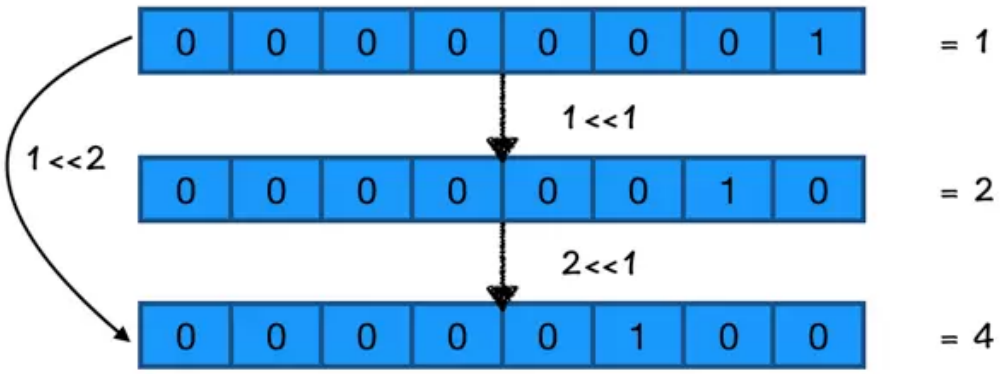
上述代码中使用两个!!（非）来将值改为bool类型。同样使用上面的例子

```
// 取出倒数第二位 rich
  0000 0010  // _tallRichHandsome
& 0000 0010 // RichMask
------------
  0000 0010 // 取出rich的值为1，其他位都置为0
```

上述代码中(_tallRichHandsome & TallMask)的值为0000 0010也就是2，但是我们需要的是一个BOOL类型的值 0 或者 1 ，那么!!2就将 2 先转化为 0 ，之后又转化为 1。相反如果按位与取得的值为 0 时，!!0将 0 先转化为 1 之后又转化为 0。 因此使用!!两个非操作将值转化为 0 或者 1 来表示相应的值。

掩码 : 上述代码中定义了三个宏，用来分别进行按位与运算而取出相应的值，一般用来按位与（&）运算的值称之为掩码。 为了能更清晰的表明掩码是为了取出哪一位的值，上述三个宏的定义可以使用<<（左移）来优化

<<：表示左移一位，下图为例。



那么上述宏定义可以使用<<（左移）优化成如下代码

```
#define TallMask (1<<2) // 0b00000100 4
#define RichMask (1<<1) // 0b00000010 2
#define HandsomeMask (1<<0) // 0b00000001 1
```

设值

设值即是将某一位设值为0或者1，可以使用|（按位或）操作符。 | : 按位或，只要

有一个1即为1，否则为0。

如果想将某一位置为1的话，那么将原本的值与掩码进行按位或的操作即可，例如我们想将tall置为1

```
// 将倒数第三位 tall置为1
  0000 0010  // _tallRichHandsome
| 0000 0100  // TallMask
------------
  0000 0110 // 将tall置为1，其他位值都不变
```

如果想将某一位置为0的话，需要将掩码按位取反（~：按位取反符），之后在与原本的值进行按位与操作即可。

```
// 将倒数第二位 rich置为0
  0000 0010  // _tallRichHandsome
& 1111 1101  // RichMask按位取反
------------
  0000 0000 // 将rich置为0，其他位值都不变
```

此时set方法内部实现如下

```
- (void)setTall:(BOOL)tall
{
    if (tall) { // 如果需要将值置为1  // 按位或掩码
        _tallRichHandsome |= TallMask;
    }else{ // 如果需要将值置为0 // 按位与（按位取反的掩码）
        _tallRichHandsome &= ~TallMask;
    }
}
- (void)setRich:(BOOL)rich
{
    if (rich) {
        _tallRichHandsome |= RichMask;
    }else{
        _tallRichHandsome &= ~RichMask;
    }
}
- (void)setHandsome:(BOOL)handsome
{
    if (handsome) {
        _tallRichHandsome |= HandsomeMask;
```

```
    }else{
        _tallRichHandsome &= ~HandsomeMask;
    }
}
```

写完set、get方法之后通过代码来查看一下是否可以设值、取值成功。

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person  = [[Person alloc] init];
        person.tall = YES;
        person.rich = NO;
        person.handsome = YES;
        NSLog(@"tall : %d, rich : %d, handsome : %d",
person.tall,person.rich,person.handsome);
    }
    return 0;
}
```

> 打印内容 : Runtime - union探寻[58212:3857728] tall : 1, rich : 0, handsome : 1

可以看出上述代码可以正常赋值和取值。但是代码还是有一定的局限性，当需要添加新属性的时候，需要重复上述工作，并且代码可读性比较差。接下来使用结构体的位域特性来优化上述代码。

位域

将上述代码进行优化，使用结构体位域，可以使代码可读性更高。 位域声明 位域名 : 位域长度; 使用位域需要注意以下3点： 1. 如果一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。

2. 位域的长度不能大于数据类型本身的长度，比如int类型就不能超过32位二进位。 3. 位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。 上述代码使用结构体位域优化之后。

```
@interface Person()
{
    struct {
```

```
        char handsome : 1; // 位域，代表占用一位空间
        char rich : 1;  // 按照顺序只占一位空间
        char tall : 1;
    }_tallRichHandsome;
}
```

set、get方法中可以直接通过结构体赋值和取值

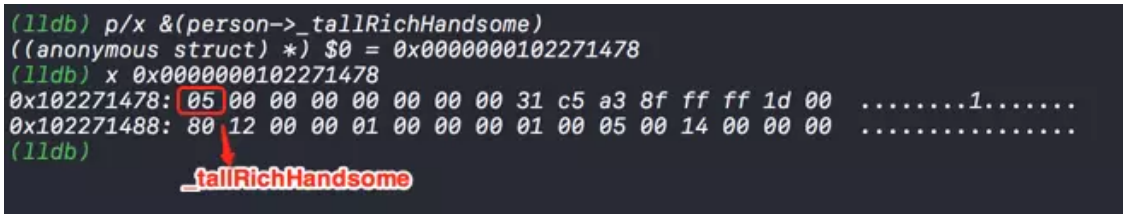
```
- (void)setTall:(BOOL)tall
{
    _tallRichHandsome.tall = tall;
}
- (void)setRich:(BOOL)rich
{
    _tallRichHandsome.rich = rich;
}
- (void)setHandsome:(BOOL)handsome
{
    _tallRichHandsome.handsome = handsome;
}
- (BOOL)tall
{
    return _tallRichHandsome.tall;
}
- (BOOL)rich
{
    return _tallRichHandsome.rich;
}
- (BOOL)handsome
{
    return _tallRichHandsome.handsome;
}
```

通过代码验证一下是否可以赋值或取值正确

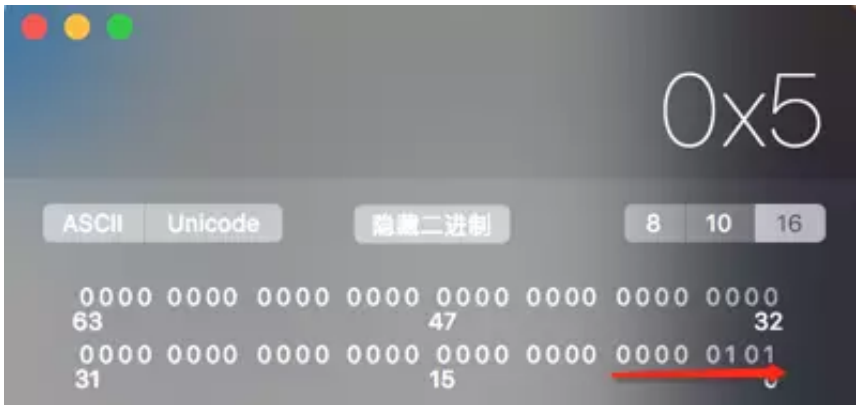
```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *person  = [[Person alloc] init];
        person.tall = YES;
        person.rich = NO;
        person.handsome = YES;
        NSLog(@"tall : %d, rich : %d, handsome : %d",
person.tall,person.rich,person.handsome);
```

```
    }
    return 0;
}
```

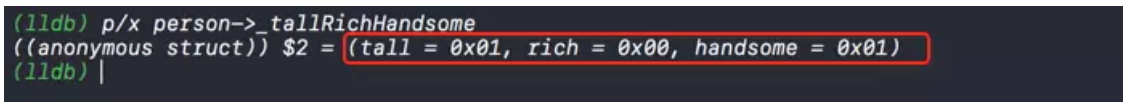
首先在log处打个断点，查看_tallRichHandsome内存储的值



因为_tallRichHandsome占据一个内存空间，也就是8个二进制位，我们将05十六进制转化为二进制查看



上图中可以发现，倒数第三位也就是tall值为1，倒数第二位也就是rich值为0，倒数一位也就是handsome值为1，如此看来和上述代码中我们设置的值一样。可以成功赋值。 接着继续打印内容： Runtime - union探寻[59366:4053478] tall : -1, rich : 0, handsome : -1 此时可以发现问题，tall与handsome我们设值为YES，讲道理应该输出的值为1为何上面输出为-1呢？ 并且上面通过打印_tallRichHandsome中存储的值，也确认tall和handsome的值都为1。我们再次打印_tallRichHandsome结构体内变量的值。



可以看到值确实为1的，为什么打印出来值为-1呢？此时应该可以想到应该是get方法内部有问题。我们来到get方法内部通过打印断点查看获取到的值。

```
- (BOOL)handsome
{
    BOOL ret = _tallRichHandsome.handsome;
```

```
    return ret;
}
```

打印ret的值

```
(lldb) po ret
255

(lldb) p/x ret
(BOOL) $1 = 0xff 255
(lldb)
```

通过打印ret的值发现其值为255，也就是1111 1111，此时也就能解释为什么打印出来值为 -1了，首先此时通过结构体获取到的handsome的值为0b1只占一个内存空间中的1位，但是BOOL值占据一个内存空间，也就是8位。当仅有1位的值扩展成8位的话，其余空位就会根据前面一位的值全部补位成1，因此此时ret的值就被映射成了0b 11111 1111。 **11111111 在一个字节时，有符号数则为-1，无符号数则为255。因此我们在打印时候打印出的值为-1** 为了验证当1位的值扩展成8位时，会全部补位，我们将tall、rich、handsome值设置为占据两位

```
@interface Person()
{
    struct {
        char tall : 2;
        char rich : 2;
        char handsome : 2;
    }_tallRichHandsome;
}
```

此时在打印就发现值可以正常打印出来。 Runtime - union探寻[60827:4259630] tall : 1, rich : 0, handsome : 1

这是因为，在get方法内部获取到的_tallRichHandsome.handsome为两位的也就是0b 01，此时在赋值给8位的BOOL类型的值时，前面的空值就会自动根据前面一位补全为0，因此返回的值为0b 0000 0001，因此打印出的值也就为1了。

因此上述问题同样可以使用!!双感叹号来解决问题。!!的原理上面已经讲解过，这里不再赘述了。 使用结构体位域优化之后的代码

```
@interface Person()
{
    struct {
        char tall : 1;
        char rich : 1;
        char handsome : 1;
    }_tallRichHandsome;
}
@end

@implementation Person

- (void)setTall:(BOOL)tall
{
    _tallRichHandsome.tall = tall;
}
- (void)setRich:(BOOL)rich
{
    _tallRichHandsome.rich = rich;
}
- (void)setHandsome:(BOOL)handsome
{
    _tallRichHandsome.handsome = handsome;
}
- (BOOL)tall
{
    return !!_tallRichHandsome.tall;
}
- (BOOL)rich
{
    return !!_tallRichHandsome.rich;
}
- (BOOL)handsome
{
    return !!_tallRichHandsome.handsome;
}
```

上述代码中使用结构体的位域则不在需要使用掩码，使代码可读性增强了很多，但是效率相比直接使用位运算的方式来说差很多，如果想要高效率的进行数据的读取与存储同时又有较强的可读性就需要使用到共用体了。

共用体

为了使代码存储数据高效率的同时，有较强的可读性，可以使用共用体来增强代码可读性，同时使用位运算来提高数据存取的效率。

使用共用体优化的代码

```
#define TallMask (1<<2) // 0b00000100 4
#define RichMask (1<<1) // 0b00000010 2
#define HandsomeMask (1<<0) // 0b00000001 1

@interface Person()
{
    union {
        char bits;
       // 结构体仅仅是为了增强代码可读性，无实质用处
        struct {
            char tall : 1;
            char rich : 1;
            char handsome : 1;
        };
    }_tallRichHandsome;
}
@end

@implementation Person

- (void)setTall:(BOOL)tall
{
    if (tall) {
        _tallRichHandsome.bits |= TallMask;
    }else{
        _tallRichHandsome.bits &= ~TallMask;
    }
}
- (void)setRich:(BOOL)rich
{
    if (rich) {
        _tallRichHandsome.bits |= RichMask;
    }else{
        _tallRichHandsome.bits &= ~RichMask;
    }
}
- (void)setHandsome:(BOOL)handsome
{
    if (handsome) {
        _tallRichHandsome.bits |= HandsomeMask;
    }else{
        _tallRichHandsome.bits &= ~HandsomeMask;
    }
}
- (BOOL)tall
```

```
{
    return !!(_tallRichHandsome.bits & TallMask);
}
- (BOOL)rich
{
    return !!(_tallRichHandsome.bits & RichMask);
}
- (BOOL)handsome
{
    return !!(_tallRichHandsome.bits & HandsomeMask);
}
```

上述代码中使用位运算这种比较高效的方式存取值，使用union共用体来对数据进行存储。增加读取效率的同时增强代码可读性。

其中_tallRichHandsome共用体只占用一个字节，因为结构体中tall、rich、handsome都只占一位二进制空间，所以结构体只占一个字节，而char类型的bits也只占一个字节，他们都在共用体中，因此共用一个字节的内存即可。

并且在get、set方法中并没有使用到结构体，结构体仅仅为了增加代码可读性，指明共用体中存储了哪些值，以及这些值各占多少位空间。同时存值取值还使用位运算来增加效率，存储使用共用体，存放的位置依然通过与掩码进行位运算来控制。

此时代码已经算是优化完成了，高效的同时可读性高，那么此时在回头看isa_t共用体的源码

isa_t源码

此时我们在回头查看isa_t源码

```
// 精简过的isa_t共用体
union isa_t
{
    isa_t() { }
    isa_t(uintptr_t value) : bits(value) { }

    Class cls;
    uintptr_t bits;

# if __arm64__
#   define ISA_MASK        0x0000000ffffffff8ULL
#   define ISA_MAGIC_MASK  0x000003f000000001ULL
#   define ISA_MAGIC_VALUE 0x000001a000000001ULL
```

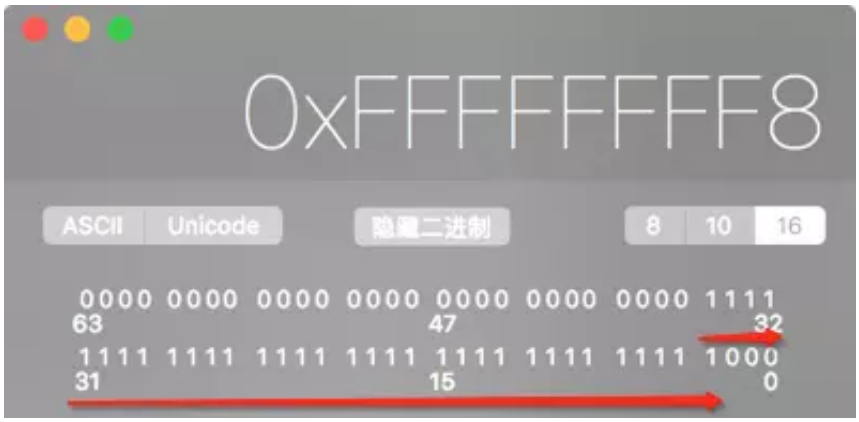
```
    struct {
        uintptr_t nonpointer        : 1;
        uintptr_t has_assoc         : 1;
        uintptr_t has_cxx_dtor      : 1;
        uintptr_t shiftcls          : 33; // MACH_VM_MAX_ADDRESS
0x1000000000
        uintptr_t magic             : 6;
        uintptr_t weakly_referenced : 1;
        uintptr_t deallocating      : 1;
        uintptr_t has_sidetable_rc  : 1;
        uintptr_t extra_rc          : 19;
#       define RC_ONE   (1ULL<<45)
#       define RC_HALF  (1ULL<<18)
    };
#endif
};
```

经过上面对位运算、位域以及共用体的分析，现在再来看源码已经可以很清晰的理解其中的内容。源码中通过共用体的形式存储了64位的值，这些值在结构体中被展示出来，通过对bits进行位运算而取出相应位置的值。

这里主要关注一下shiftcls，shiftcls中存储着Class、Meta-Class对象的内存地址信息，我们之前在OC对象的本质中提到过，对象的isa指针需要同ISA_MASK经过一次&（按位与）运算才能得出真正的Class对象地址。

```
(lldb) p/x object->isa
(Class) $2 = 0x001dffff96537141 NSObject
(lldb) p/x objectClass
(Class) $3 = 0x00007fff96537140 NSObject
(lldb) p/x 0x00007ffffffffff8 & 0x001dffff96537141
(long) $4 = 0x00007fff96537140
(lldb)
```

那么此时我们重新来看ISA_MASK的值0x0000000ffffffff8ULL，我们将其转化为二进制数



上图中可以看出ISA_MASK的值转化为二进制中有33位都为1，上面提到过按位与的作用是可以取出这33位中的值。那么此时很明显了，同ISA_MASK进行按位与运算即可以取出Class或Meta-Class的值。

同时可以看出ISA_MASK最后三位的值为0，那么任何数同ISA_MASK按位与运算之后，得到的最后三位必定都为0，因此任何类对象或元类对象的内存地址最后三位必定为0，转化为十六进制末位必定为8或者0。

isa中存储的信息及作用

将结构体取出来标记一下这些信息的作用。

```
struct {
    // 0代表普通的指针，存储着Class，Meta-Class对象的内存地址。
    // 1代表优化后的使用位域存储更多的信息。
    uintptr_t nonpointer        : 1;

   // 是否有设置过关联对象，如果没有，释放时会更快
    uintptr_t has_assoc         : 1;

    // 是否有C++析构函数，如果没有，释放时会更快
    uintptr_t has_cxx_dtor      : 1;

    // 存储着Class、Meta-Class对象的内存地址信息
    uintptr_t shiftcls          : 33;

    // 用于在调试时分辨对象是否未完成初始化
    uintptr_t magic             : 6;

    // 是否有被弱引用指向过。
    uintptr_t weakly_referenced : 1;
```

```
    // 对象是否正在释放
    uintptr_t deallocating      : 1;

    // 引用计数器是否过大无法存储在isa中
    // 如果为1，那么引用计数会存储在一个叫SideTable的类的属性中
    uintptr_t has_sidetable_rc  : 1;

    // 里面存储的值是引用计数器减1
    uintptr_t extra_rc          : 19;
};
```

验证

通过下面一段代码验证上述信息存储的位置及作用

```
// 以下代码需要在真机中运行，因为真机中才是__arm64__ 位架构
- (void)viewDidLoad {
    [super viewDidLoad];
    Person *person = [[Person alloc] init];
    NSLog(@"%p",[person class]);
    NSLog(@"%@",person);
}
```

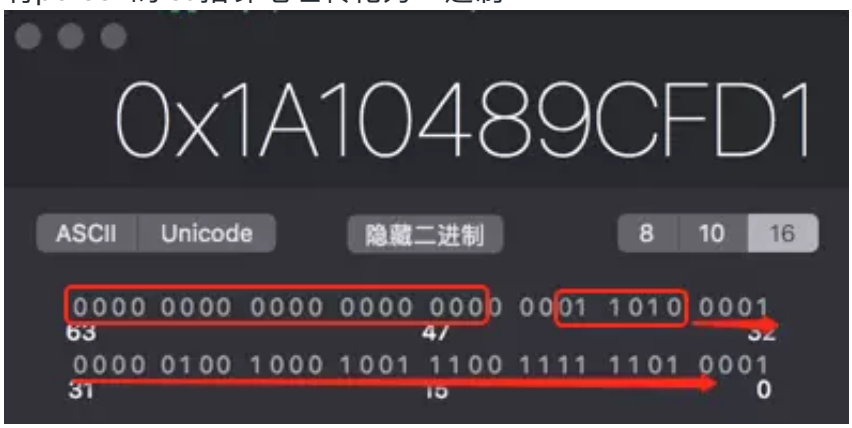
首先打印person类对象的地址，之后通过断点打印一下person对象的isa指针地址。

首先来看一下打印的内容

```
2018-06-15 14:17:30.663482+0800 Runtime-test[338:33939] 0x10489cfd0
(lldb) p/x person->isa
(Class) $0 = 0x000001a10489cfd1 Person
(lldb)
```

将类对象地址转化为二进制

将person的isa指针地址转化为二进制



shiftcls : shiftcls中存储类对象地址，通过上面两张图对比可以发现存储类对象地址的33位二进制内容完全相同。 extra_rc : extra_rc的19位中存储着的值为引用计数减一，因为此时person的引用计数为1，因此此时extra_rc的19位二进制中存储的是0。 magic : magic的6位用于在调试时分辨对象是否未完成初始化，上述代码中person已经完成初始化，那么此时这6位二进制中存储的值011010即为共用体中定义的宏# define ISA_MAGIC_VALUE 0x000001a000000001ULL的值。 nonpointer : 这里肯定是使用的优化后的isa，因此nonpointer的值肯定为1 因为此时person对象没有关联对象并且没有弱指针引用过，可以看出has_assoc和weakly_referenced值都为0，接着我们为person对象添加弱引用和关联对象，来观察一下has_assoc和weakly_referenced的变化。

```
- (void)viewDidLoad {
    [super viewDidLoad];
    Person *person = [[Person alloc] init];
    NSLog(@"%p",[person class]);
    // 为person添加弱引用
    __weak Person *weakPerson = person;
    // 为person添加关联对象
    objc_setAssociatedObject(person, @"name", @"xx_cc",
```

```
OBJC_ASSOCIATION_RETAIN_NONATOMIC);
    NSLog(@"%@",person);
}
```

重新打印person的isa指针地址将其转化为二进制可以看到has_assoc和weakly_referenced的值都变成了1

注意：只要设置过关联对象或者弱引用引用过对象has_assoc和weakly_referenced的值就会变成1，不论之后是否将关联对象置为nil或断开弱引用。

```
void *objc_destructInstance(id obj)
{
    if (obj) {
        Class isa = obj->getIsa();
        // 是否有c++析构函数
        if (isa->hasCxxDtor()) {
            object_cxxDestruct(obj);
        }
        // 是否有关联对象，如果有则移除
        if (isa->instancesHaveAssociatedObjects()) {
            _object_remove_assocations(obj);
        }
        objc_clear_deallocating(obj);
    }
    return obj;
}
```

相信至此我们已经对isa指针有了新的认识，__arm64__架构之后，isa指针不单单只存储了Class或Meta-Class的地址，而是使用共用体的方式存储了更多信息，其中shiftcls存储了Class或Meta-Class的地址，需要同ISA_MASK进行按位&运算才可以

取出其内存地址值。