Class的结构

通过上一章中对isa本质结构有了新的认识，今天来回顾Class的结构，重新认识Class内部结构。

首先来看一下Class的内部结构代码，对探寻Class的本质做简单回顾。

```
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);
    }
}
```

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

class_rw_t

上述源码中我们知道`bits & FAST_DATA_MASK`位运算之后，可以得到`class_rw_t`，而`class_rw_t`中存储着方法列表、属性列表以及协议列表，来看一下`class_rw_t`部分代码

```
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;
};
```

上述源码中，`method_array_t、property_array_t、protocol_array_t`其实都是二维数组，来到`method_array_t、property_array_t、protocol_array_t`内部看一下。这里以`method_array_t`为例，`method_array_t`本身就是一个数组，数组里面存放的是数组`method_list_t`，`method_list_t`里面最终存放的是`method_t`

```
class method_array_t :
    public list_array_tt<method_t, method_list_t>
{
    typedef list_array_tt<method_t, method_list_t> Super;

 public:
    method_list_t **beginCategoryMethodLists() {
        return beginLists();
    }

    method_list_t **endCategoryMethodLists(Class cls);

    method_array_t duplicate() {
        return Super::duplicate<method_array_t>();
    }
};

class property_array_t :
    public list_array_tt<property_t, property_list_t>
{
    typedef list_array_tt<property_t, property_list_t> Super;

 public:
    property_array_t duplicate() {
```

```
        return Super::duplicate<property_array_t>();
    }
};

class protocol_array_t :
    public list_array_tt<protocol_ref_t, protocol_list_t>
{
    typedef list_array_tt<protocol_ref_t, protocol_list_t> Super;

 public:
    protocol_array_t duplicate() {
        return Super::duplicate<protocol_array_t>();
    }
};
```

`class_rw_t`里面的`methods、properties、protocols`是二维数组，是可读可写的，其中包含了类的初始内容以及分类的内容。

这里以`method_array_t`为例，图示其中的结构。

[图片上传中...(image-c6d8a6-1529890407880-10)]

class_ro_t

我们之前提到过`class_ro_t`中也有存储方法、属性、协议列表，另外还有成员变量列表。

接着来看一下`class_ro_t`部分代码

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

    const uint8_t * ivarLayout;

    const char * name;
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
    const ivar_list_t * ivars;
```

```
    const uint8_t * weakIvarLayout;
    property_list_t *baseProperties;

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

上述源码中可以看到 `class_ro_t *ro` 是只读的，内部直接存储的直接就是 `method_list_t`、`protocol_list_t` 、`property_list_t` 类型的一维数组，数组里面分别存放的是类的初始信息，以 `method_list_t` 为例， `method_list_t` 中直接存放的就是 `method_t` ，但是是只读的，不允许增加删除修改。

总结

以方法列表为例， `class_rw_t` 中的methods是二维数组的结构，并且可读可写，因此可以动态的添加方法，并且更加便于分类方法的添加。因为我们在Category的本质里面提到过， `attachList` 函数内通过 `memmove 和 memcpy` 两个操作将分类的方法列表合并在本类的方法列表中。那么此时就将分类的方法和本类的方法统一整合到一起了。

其实一开始类的方法，属性，成员变量属性协议等等都是存放在 `class_ro_t` 中的，当程序运行的时候，需要将分类中的列表跟类初始的列表合并在一起的时，就会将 `class_ro_t` 中的列表和分类中的列表合并起来存放在 `class_rw_t` 中，也就是说 `class_rw_t` 中有部分列表是从 `class_ro_t` 里面拿出来的。并且最终和分类的方法合并。可以通过源码提现这里一点。

realizeClass部分源码

```
static Class realizeClass(Class cls)
{
    runtimeLock.assertWriting();

    const class_ro_t *ro;
    class_rw_t *rw;
    Class supercls;
    Class metacls;
    bool isMeta;

    if (!cls) return nil;
```

```
    if (cls->isRealized()) return cls;
    assert(cls == remapClass(cls));

    // 最开始cls->data是指向ro的
    ro = (const class_ro_t *)cls->data();

    if (ro->flags & RO_FUTURE) {
        // rw已经初始化并且分配内存空间
        rw = cls->data();  // cls->data指向rw
        ro = cls->data()->ro;  // cls->data()->ro指向ro
        cls->changeInfo(RW_REALIZED|RW_REALIZING, RW_FUTURE);
    } else {
        // 如果rw并不存在，则为rw分配空间
        rw = (class_rw_t *)calloc(sizeof(class_rw_t), 1); // 分配空间
        rw->ro = ro;  // rw->ro重新指向ro
        rw->flags = RW_REALIZED|RW_REALIZING;
        // 将rw传入setData函数，等于cls->data()重新指向rw
        cls->setData(rw);
    }
}
```

那么从上述源码中就可以发现，类的初始信息本来其实是存储在 `class_ro_t` 中的，并且 `ro` 本来是指向 `cls->data()` 的，也就是说 `bits.data()` 得到的是 `ro`，但是在运行过程中创建了 `class_rw_t`，并将 `cls->data` 指向 `rw`，同时将初始信息 `ro` 赋值给 `rw` 中的 `ro`。最后在通过setData(rw)设置data。那么此时 `bits.data()` 得到的就是 `rw`，之后再去检查是否有分类，同时将分类的方法，属性，协议列表整合存储在 `class_rw_t` 的方法，属性及协议列表中。

通过上述对源码的分析，我们对 `class_rw_t` 内存储方法、属性、协议列表的过程有了更清晰的认识，那么接下来探寻 `class_rw_t` 中是如何存储方法的。

class_rw_t中是如何存储方法的

method_t

我们知道 `method_array_t、property_array_t、protocol_array_t` 中以 `method_array_t` 为例， `method_array_t` 中最终存储的是 `method_t`， `method_t` 是对方法、函数的封装，每一个方法对象就是一个 `method_t`。通过源码看一下 `method_t` 的结构体

```
struct method_t {
    SEL name;  // 函数名
    const char *types;  // 编码（返回值类型，参数类型）
    IMP imp; // 指向函数的指针（函数地址）
};
```

method_t结构体中可以看到三个成员变量，我们依次来看三个成员变量分别代表什么。

SEL

SEL代表方法\函数名，一般叫做选择器，底层结构跟`char *`类似`typedef struct objc_selector *SEL;`，可以把SEL看做是方法名字符串。

SEL可以通过`@selector()`和`sel_registerName()`获得

```
SEL sel1 = @selector(test);
SEL sel2 = sel_registerName("test");
```

也可以通过`sel_getName()`和`NSStringFromSelector()`将SEL转成字符串

```
char *string = sel_getName(sel1);
NSString *string2 = NSStringFromSelector(sel2);
```

不同类中相同名字的方法，所对应的方法选择器是相同的。

```
NSLog(@"%p,%p", sel1,sel2);
Runtime-test[23738:8888825] 0x1017718a3,0x1017718a3
```

SEL仅仅代表方法的名字，并且不同类中相同的方法名的SEL是全局唯一的。

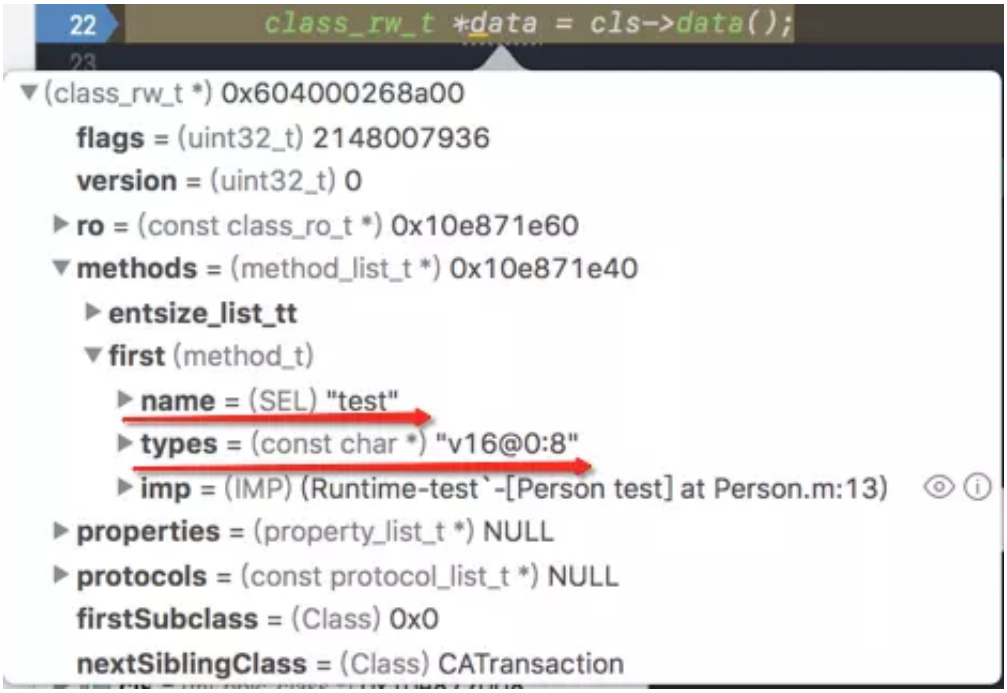
types

`types`包含了函数返回值，参数编码的字符串。通过字符串拼接的方式将返回值和参数拼接成一个字符串，来代表函数返回值及参数。

我们通过代码查看一下types是如何代表函数返回值及参数的，首先通过自己模拟Class的内部实现，通过强制转化来探寻内部数据，相关代码在探寻Class的本质中提到过，这里不在赘述。

```
Person *person = [[Person alloc] init];
xx_objc_class *cls = (__bridge xx_objc_class *)[Person class];
class_rw_t *data = cls->data();
```

通过断点可以在data中找到types的值



上图中可以看出`types`的值为`v16@0:8`，那么这个值代表什么呢？apple为了能够清晰的使用字符串表示方法及其返回值，制定了一系列对应规则，通过下表可以看到一一对应关系

Table 6-1 Objective-C type encodings

Code	Meaning
`c`	A `char`
`i`	An `int`
`s`	A `short`
`l`	A `long` `l` is treated as a 32-bit quantity on 64-bit programs.
`q`	A `long long`
`C`	An `unsigned char`
`I`	An `unsigned int`
`S`	An `unsigned short`
`L`	An `unsigned long`
`Q`	An `unsigned long long`
`f`	A `float`
`d`	A `double`
`B`	A C++ `bool` or a C99 `_Bool`
`v`	A `void`
`*`	A character string (`char *`)
`@`	An object (whether statically typed or typed `id`)
`#`	A class object (`Class`)
`:`	A method selector (`SEL`)
[*array type*]	An array
{*name=type...*}	A structure
(*name=type...*)	A union
`b`num	A bit field of *num* bits
^type	A pointer to *type*
`?`	An unknown type (among other things, this code is used for function pointers)

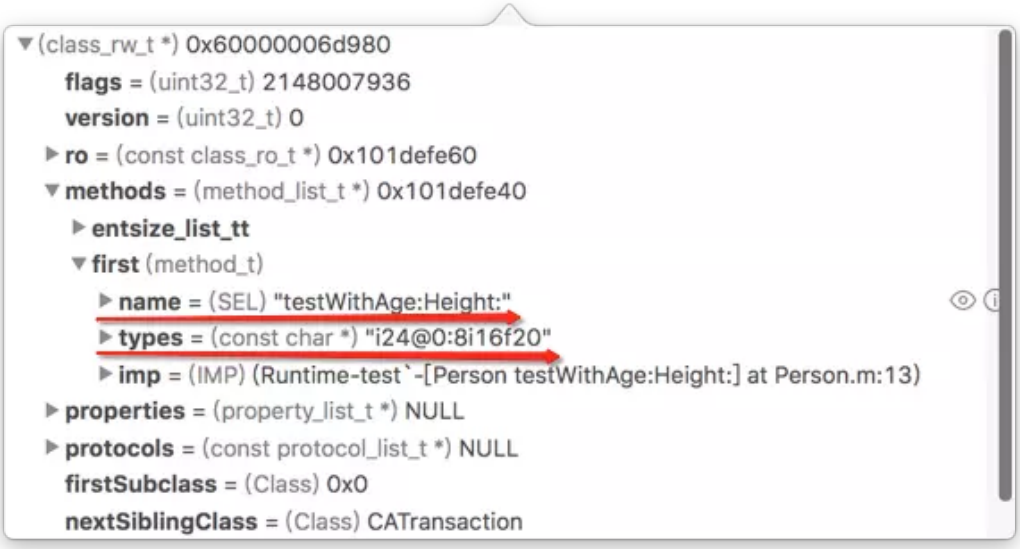
将types的值同表中的一一对照查看 `types` 的值 `v16@0:8` 代表什么

```
- (void) test;
```

```
 v    16      @     0     :     8
void         id          SEL
// 16表示参数的占用空间大小，id后面跟的0表示从0位开始存储，id占8位空间。
// SEL后面的8表示从第8位开始存储，SEL同样占8位空间
```

我们知道任何方法都默认有两个参数的，`id`类型的`self`，和`SEL`类型的`_cmd`，而上述通过对`types`的分析同时也验证了这个说法。

为了能够看的更加清晰，我们为test添加返回值及参数之后重新查看types的值。



同样通过上表找出一一对应的值，查看types的值代表的方法

```
- (int)testWithAge:(int)age Height:(float)height
{
    return 0;
}
  i    24    @    0    :    8    i    16    f    20
int         id        SEL       int       float
// 参数的总占用空间为 8 + 8 + 4 + 4 = 24
// id 从第0位开始占据8位空间
// SEL 从第8位开始占据8位空间
// int 从第16位开始占据4位空间
// float 从第20位开始占据4位空间
```

iOS提供了`@encode`的指令，可以将具体的类型转化成字符串编码。

```
NSLog(@"%s",@encode(int));
NSLog(@"%s",@encode(float));
NSLog(@"%s",@encode(id));
NSLog(@"%s",@encode(SEL));

// 打印内容
Runtime-test[25275:9144176] i
Runtime-test[25275:9144176] f
Runtime-test[25275:9144176] @
Runtime-test[25275:9144176] :
```

上述代码中可以看到，对应关系确实如上表所示。

IMP

`IMP`代表函数的具体实现，存储的内容是函数地址。也就是说当找到`imp`的时候就可以找到函数实现，进而对函数进行调用。

在上述代码中打印`IMP`的值

```
Printing description of data->methods->first.imp:
(IMP) imp = 0x000000010c66a4a0 (Runtime-test`-[Person
testWithAge:Height:] at Person.m:13)
```

之后在`test`方法内部打印断点，并来到其方法内部可以看出`imp`中的存储的地址也就是方法实现的地址。

通过上面的学习我们知道了方法列表是如何存储在`Class类对象`中的，但是当多次继承的子类想要调用基类方法时，就需要通过`superclass`指针一层一层找到基类，在从基类方法列表中找到对应的方法进行调用。如果多次调用基类方法，那么就需要多次遍历每一层父类的方法列表，这对性能来说无疑是伤害巨大的。

apple通过方法缓存的形式解决了这一问题，接下来我们来探寻`Class类对象`是如何进行方法缓存的

方法缓存 cache_t

回到类对象结构体，成员变量`cache`就是用来对方法进行缓存的。

```
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);
    }
}
```

`cache_t cache;`用来缓存曾经调用过的方法，可以提高方法的查找速度。

回顾方法调用过程：调用方法的时候，需要去方法列表里面进行遍历查找。如果方法不在列表里面，就会通过`superclass`找到父类的类对象，在去父类类对象方法列表里面遍历查找。

如果方法需要调用很多次的话，那就相当于每次调用都需要去遍历多次方法列表，为了能够快速查找方法，`apple`设计了`cache_t`来进行方法缓存。

每当调用方法的时候，会先去`cache`中查找是否有缓存的方法，如果没有缓存，在去类对象方法列表中查找，以此类推直到找到方法之后，就会将方法直接存储在`cache`中，下一次在调用这个方法的时候，就会在类对象的`cache`里面找到这个方法，直接调用了。

cache_t 如何进行缓存

那么`cache_t`是如何对方法进行缓存的呢？首先来看一下`cache_t`的内部结构。

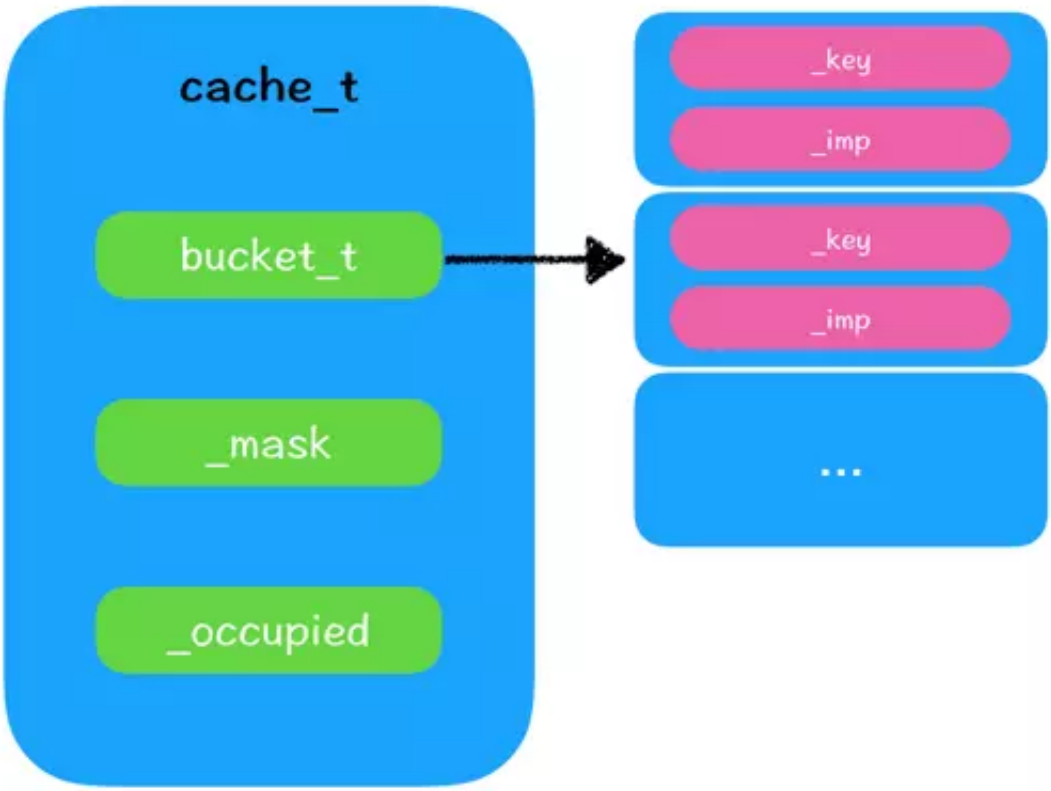
```
struct cache_t {
    struct bucket_t *_buckets; // 散列表 数组
    mask_t _mask; // 散列表的长度 -1
    mask_t _occupied; // 已经缓存的方法数量
};
```

`bucket_t`是以数组的方式存储方法列表的，看一下`bucket_t`内部结构

```
struct bucket_t {
private:
    cache_key_t _key; // SEL作为Key
    IMP _imp; // 函数的内存地址
};
```

从源码中可以看出`bucket_t`中存储着`SEL`和`_imp`，通过`key->value`的形式，以`SEL`为`key`，`函数实现的内存地址 _imp`为`value`来存储方法。

通过一张图来展示一下`cache_t`的结构。



上述`bucket_t`列表我们称之为散列表（哈希表） **散列表（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。**

那么apple如何在散列表中快速并且准确的找到对应的key以及函数实现呢？这就需要我们通过源码来看一下apple的散列函数是如何设计的。

散列函数及散列表原理

首先来看一下存储的源码，主要查看几个函数，关键代码都有注释，不在赘述。

cache_fill 及 cache_fill_nolock 函数

```
void cache_fill(Class cls, SEL sel, IMP imp, id receiver)
{
#if !DEBUG_TASK_THREADS
    mutex_locker_t lock(cacheUpdateLock);
    cache_fill_nolock(cls, sel, imp, receiver);
```

```
#else
    _collecting_in_critical();
    return;
#endif
}

static void cache_fill_nolock(Class cls, SEL sel, IMP imp, id
receiver)
{
    cacheUpdateLock.assertLocked();
    // 如果没有initialize直接return
    if (!cls->isInitialized()) return;
    // 确保线程安全，没有其他线程添加缓存
    if (cache_getImp(cls, sel)) return;
    // 通过类对象获取到cache
    cache_t *cache = getCache(cls);
    // 将SEL包装成Key
    cache_key_t key = getKey(sel);
   // 占用空间+1
    mask_t newOccupied = cache->occupied() + 1;
   // 获取缓存列表的缓存能力，能存储多少个键值对
    mask_t capacity = cache->capacity();
    if (cache->isConstantEmptyCache()) {
        // 如果为空的，则创建空间，这里创建的空间为4个。
        cache->reallocate(capacity, capacity ?: INIT_CACHE_SIZE);
    }
    else if (newOccupied <= capacity / 4 * 3) {
        // 如果所占用的空间占总数的3/4一下，则继续使用现在的空间
    }
    else {
       // 如果占用空间超过3/4则扩展空间
        cache->expand();
    }
    // 通过key查找合适的存储空间。
    bucket_t *bucket = cache->find(key, receiver);
    // 如果key==0则说明之前未存储过这个key，占用空间+1
    if (bucket->key() == 0) cache->incrementOccupied();
    // 存储key，imp
    bucket->set(key, imp);
}
```

reallocate 函数

通过上述源码看到`reallocate`函数负责分配散列表空间，来到`reallocate`函数内

部。

```
void cache_t::reallocate(mask_t oldCapacity, mask_t newCapacity)
{
    // 旧的散列表能否被释放
    bool freeOld = canBeFreed();
    // 获取旧的散列表
    bucket_t *oldBuckets = buckets();
    // 通过新的空间需求量创建新的散列表
    bucket_t *newBuckets = allocateBuckets(newCapacity);

    assert(newCapacity > 0);
    assert((uintptr_t)(mask_t)(newCapacity-1) == newCapacity-1);
    // 设置Buckets和Mash，Mask的值为散列表长度-1
    setBucketsAndMask(newBuckets, newCapacity - 1);
    // 释放旧的散列表
    if (freeOld) {
        cache_collect_free(oldBuckets, oldCapacity);
        cache_collect(false);
    }
}
```

上述源码中首次传入`reallocate`函数的`newCapacity`为`INIT_CACHE_SIZE`，`INIT_CACHE_SIZE`是个枚举值，也就是4。因此散列表最初创建的空间就是4个。

```
enum {
    INIT_CACHE_SIZE_LOG2 = 2,
    INIT_CACHE_SIZE      = (1 << INIT_CACHE_SIZE_LOG2)
};
```

expand ()函数

当散列表的空间被占用超过3/4的时候，散列表会调用`expand ()`函数进行扩展，我们来看一下`expand ()`函数内散列表如何进行扩展的。

```
void cache_t::expand()
{
    cacheUpdateLock.assertLocked();
```

```
    // 获取旧的散列表的存储空间
    uint32_t oldCapacity = capacity();
    // 将旧的散列表存储空间扩容至两倍
    uint32_t newCapacity = oldCapacity ? oldCapacity*2 :
INIT_CACHE_SIZE;
    // 为新的存储空间赋值
    if ((uint32_t)(mask_t)newCapacity != newCapacity) {
        newCapacity = oldCapacity;
    }
    // 调用reallocate函数，重新创建存储空间
    reallocate(oldCapacity, newCapacity);
}
```

上述源码中可以发现散列表进行扩容时会将容量增至之前的2倍。

find 函数

最后来看一下散列表中如何快速的通过 `key` 找到相应的 `bucket` 呢？我们来到 `find` 函数内部

```
bucket_t * cache_t::find(cache_key_t k, id receiver)
{
    assert(k != 0);
    // 获取散列表
    bucket_t *b = buckets();
    // 获取mask
    mask_t m = mask();
    // 通过key找到key在散列表中存储的下标
    mask_t begin = cache_hash(k, m);
    // 将下标赋值给i
    mask_t i = begin;
    // 如果下标i中存储的bucket的key==0说明当前没有存储相应的key，将b[i]返回
出去进行存储
    // 如果下标i中存储的bucket的key==k，说明当前空间内已经存储了相应key，将
b[i]返回出去进行存储
    do {
        if (b[i].key() == 0  ||  b[i].key() == k) {
            // 如果满足条件则直接reutrn出去
            return &b[i];
        }
    // 如果走到这里说明上面不满足，那么会往前移动一个空间重新进行判定，知道可以
成功return为止
    } while ((i = cache_next(i, m)) != begin);
```

```
    // hack
    Class cls = (Class)((uintptr_t)this - offsetof(objc_class,
cache));
    cache_t::bad_cache(receiver, (SEL)k, cls);
}
```

函数`cache_hash (k, m)`用来通过`key`找到方法在散列表中存储的下标，来到`cache_hash (k, m)`函数内部

```
static inline mask_t cache_hash(cache_key_t key, mask_t mask)
{
    return (mask_t)(key & mask);
}
```

可以发现`cache_hash (k, m)`函数内部仅仅是进行了`key & mask`的按位与运算，得到下标即存储在相应的位置上。按位与运算在上文中已详细讲解过，这里不在赘述。

_mask

通过上面的分析我们知道`_mask`的值是散列表的长度减一，那么任何数通过与`_mask`进行按位与运算之后获得的值都会小于等于`_mask`，因此不会出现数组溢出的情况。

举个例子，假设散列表的长度为8，那么mask的值为7

```
  0101 1011  // 任意值
& 0000 0111  // mask = 7
------------
  0000 0011 //获取的值始终等于或小于mask的值
```

总结

当第一次使用方法时，消息机制通过isa找到方法之后，会对方法以`SEL为keyIMP为value`的方式缓存在`cache`的`_buckets`中，当第一次存储的时候，会创建具有4个

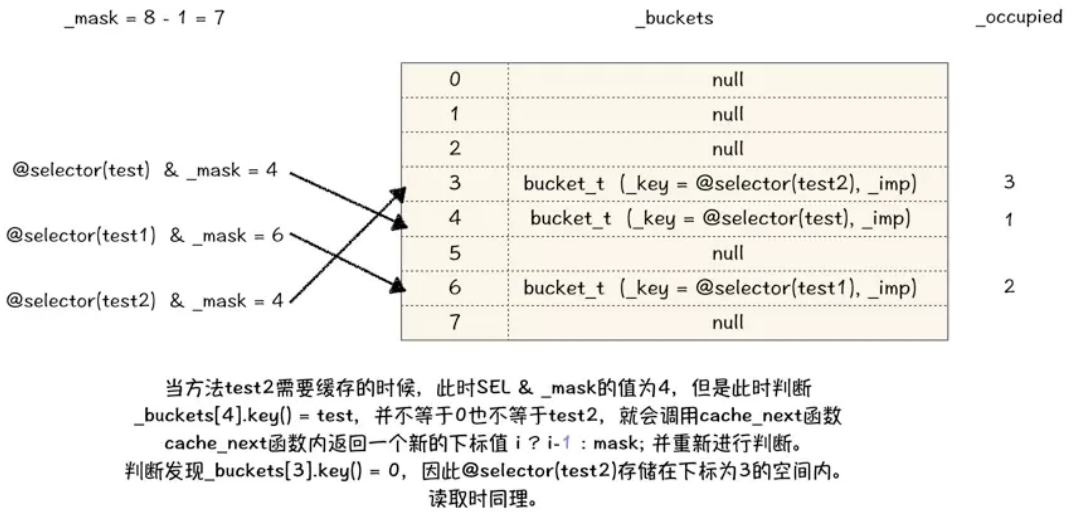
空间的散列表，并将`_mask`的值置为散列表的长度减一，之后通过`SEL & mask`计算出方法存储的下标值，并将方法存储在散列表中。举个例子，如果计算出下标值为3，那么就将方法直接存储在下标为3的空间中，前面的空间会留空。

当散列表中存储的方法占据散列表长度超过3/4的时候，散列表会进行扩容操作，将创建一个新的散列表并且空间扩容至原来空间的两倍，并重置`_mask`的值，最后释放旧的散列表，此时再有方法要进行缓存的话，就需要重新通过`SEL & mask`计算出下标值之后在按照下标进行存储了。

如果一个类中方法很多，其中很可能会出现多个方法的`SEL & mask`得到的值为同一个下标值，那么会调用`cache_next`函数往下标值-1位去进行存储，如果下标值-1位空间中有存储方法，并且key不与要存储的key相同，那么再到前面一位进行比较，直到找到一位空间没有存储方法或者`key`与要存储的`key`相同为止，如果到下标0的话就会到下标为`_mask`的空间也就是最大空间处进行比较。

当要查找方法时，并不需要遍历散列表，同样通过`SEL & mask`计算出下标值，直接去下标值的空间取值即可，同上，如果下标值中存储的key与要查找的key不相同，就去前面一位查找。这样虽然占用了少量控件，但是大大节省了时间，也就是说其实apple是使用空间换取了存取的时间。

通过一张图更清晰的看一下其中的流程。



验证上述流程

通过一段代码演示一下 。同样使用仿照`objc_class结构体`自定义一个结构体，并进

行强制转化来查看其内部数据，自定义结构体在之前的文章中使用过多次这里不在赘述。

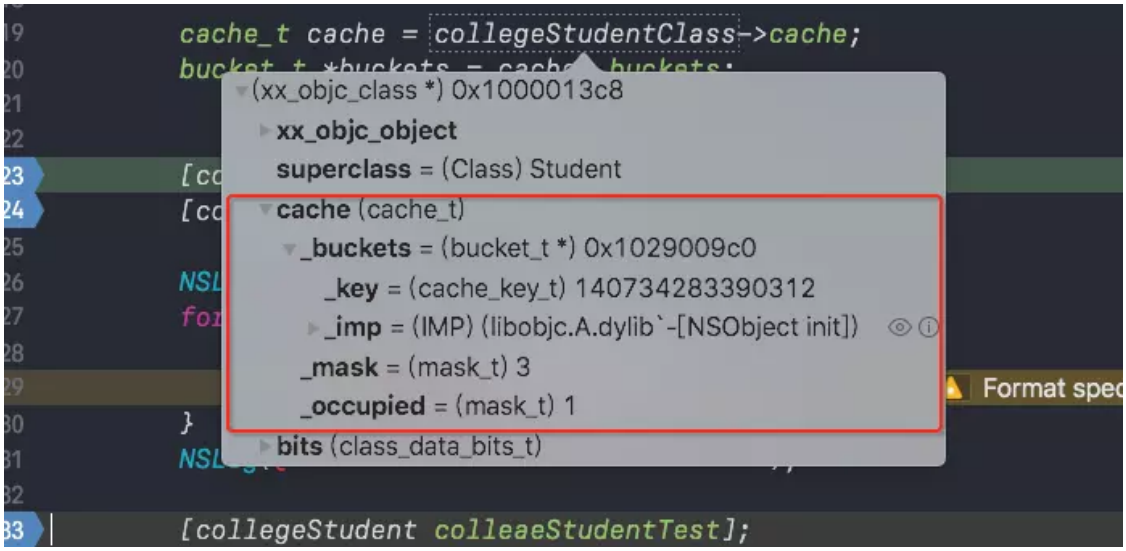
我们创建`Person`类继承`NSObject`， `Student`类继承`Person`， `CollegeStudent`继承`Student`。三个类分别有`personTest`, `studentTest`, `colleaeStudentTest`方法

通过打印断点来看一下方法缓存的过程

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {

        CollegeStudent *collegeStudent = [[CollegeStudent alloc]
init];
        xx_objc_class *collegeStudentClass = (__bridge
xx_objc_class *)[CollegeStudent class];

        cache_t cache = collegeStudentClass->cache;
        bucket_t *buckets = cache._buckets;

        [collegeStudent personTest];
        [collegeStudent studentTest];

        NSLog(@"----------------------------");
        for (int i = 0; i <= cache._mask; i++) {
            bucket_t bucket = buckets[i];
            NSLog(@"%s %p", bucket._key, bucket._imp);
        }
        NSLog(@"----------------------------");

        [collegeStudent colleaeStudentTest];

        cache = collegeStudentClass->cache;
        buckets = cache._buckets;
        NSLog(@"----------------------------");
        for (int i = 0; i <= cache._mask; i++) {
            bucket_t bucket = buckets[i];
            NSLog(@"%s %p", bucket._key, bucket._imp);
        }
        NSLog(@"----------------------------");

        NSLog(@"%p",@selector(colleaeStudentTest));
        NSLog(@"----------------------------");
    }
    return 0;
}
```

我们分别在 `collegeStudent` 实例对象调用 `personTest`，`studentTest`，`colleaeStudentTest` 方法处打断点查看 `cache` 的变化。

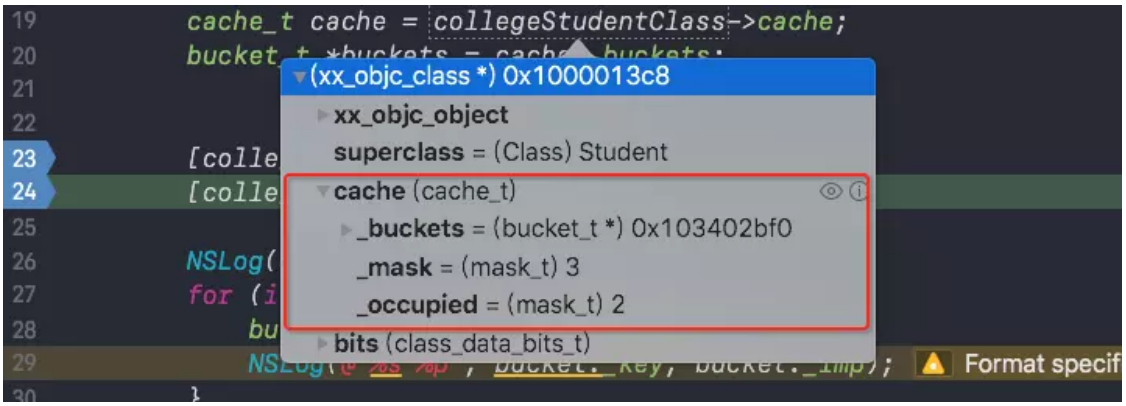
`personTest` 方法调用之前



从上图中可以发现，`personTest` 方法调用之前，`cache` 中仅仅存储了 `init方法`，上图中可以看出 `init方法` 恰好存储在下标为0的位置因此我们可以看到，`_mask` 的值为3验证我们上述源码中提到的散列表第一次存储时会分配4个内存空间，`_occupied` 的值为1证明此时 `_buckets` 中仅仅存储了一个方法。

当 `collegeStudent` 在调用 `personTest` 的时候，首先发现 `collegeStudent类对象` 的 `cache` 中没有 `personTest方法`，就会去 `collegeStudent类对象` 的方法列表中查找，方法列表中也没有，那么就通过 `superclass指针` 找到 `Student类对象`，`Studeng类对象` 中 `cache` 和方法列表同样没有，再通过 `superclass指针` 找到 `Person类对象`，最终在 `Person类对象` 方法列表中找到之后进行调用，并缓存在 `collegeStudent类对象` 的 `cache` 中。

执行 `personTest` 方法之后查看 `cache` 方法的变化



上图中可以发现 `_occupied` 值为2，说明此时 `personTest` 方法已经被缓存在 `collegeStudent类对象` 的 `cache` 中。

同理执行过 `studentTest` 方法之后，我们通过打印查看一下此时 `cache` 内存储的信息

[图片上传中...(image-db43fb-1529890407878-1)]

上图中可以看到 `cache` 中确实存储了 `init` 、`personTest` 、`studentTest` 三个方法。

那么执行过 `colleaeStudentTest方法` 之后此时 `cache` 中应该对 `colleaeStudentTest方法` 进行缓存。上面源码提到过，当存储的方法数超过散列表长度的3/4时，系统会重新创建一个容量为原来两倍的新的散列表替代原来的散列表。过掉 `colleaeStudentTest方法` ，重新打印 `cache` 内存储的方法查看。

```
Runtime[57861:6054889] personTest
Runtime[57861:6054889] studentTest
Runtime[57861:6054889] ----------------------------
Runtime[57861:6054889] init 0x7fff6edc4ffd
Runtime[57861:6054889] personTest 0x100000b00
Runtime[57861:6054889] (null) 0x0
Runtime[57861:6054889] studentTest 0x100000e10
Runtime[57861:6054889] ----------------------------
```

可以看出上图中 `_bucket` 散列表扩容之后仅仅存储了 `colleaeStudentTest方法` ，并且上图中打印 `SEL & _mask` 位运算得出下标的值确实是 `_bucket` 列表中 `colleaeStudentTest方法` 存储的位置。

至此已经对Class的结构及方法缓存的过程有了新的认知，apple通过散列表的形式对方法进行缓存，以少量的空间节省了大量查找方法的时间。