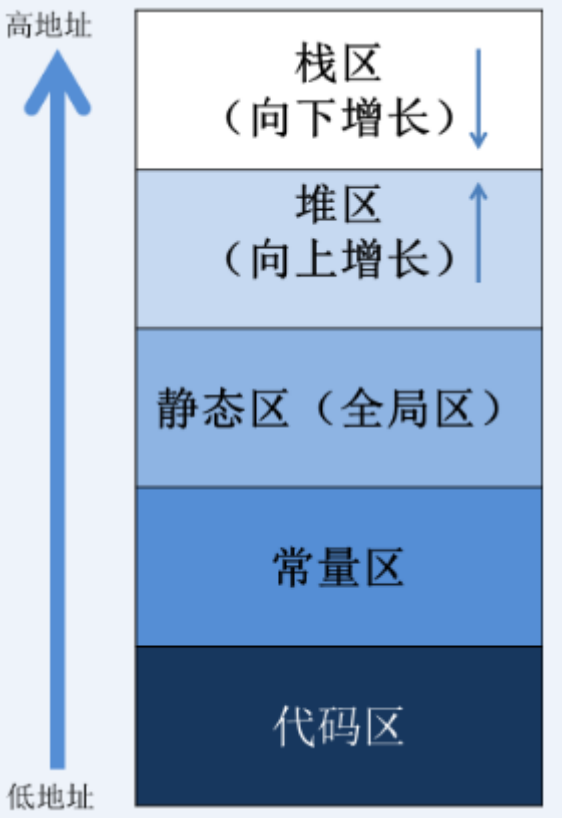
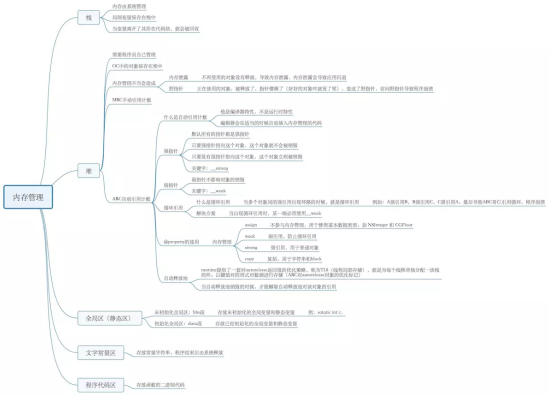
iOS

堆栈





- 栈（stack）区：
  - 栈是由编译器自动分配并释放，存放程序临时创建的局部变量，参数等。
  - 先进后出，一旦出了作用域就会被销毁
  - 栈区地址从高到低分配；
  - 自动管理内存；

- 堆（heap）区：堆是由程序员自己管理分配和释放的，它大小并不固定。是从低地址到高地址扩张与压缩。
- 全局（静态）区：
  - 它包含1.数据区（存放程序已初使化变量）。2.BSS区（程序中未初始化全局变量）；
  - 存放全局变量和静态变量（未初始化过 、初始化过）；
  - 初始化的全局变量和静态变量存放在一块区域，未初始化的全局变量和静态变量在相邻的另一块区域；
  - 程序结束后由系统释放；
- 常量区：
  - 存放常量字符串；
  - 程序结束后由系统释放；
- 代码区：代码段是用来存放可执行文件的操作指令（存放函数的二进制代码），它只能读取不能修改

- 当App启动后，代码区，常量区，全局区大小已固定，而堆区和栈区是时时刻刻变化的

Foundation与CFoundation对象

- foundation对象是objective-c对象，CFoundation对象是C对象。
- 在ARC(Automatic Reference Counting)下，oc对象不需要手动管理内存。
- 两者可通过bridge进行转换，并决定是否由ARC系统自动管理。
- _bridge不改变管理权。

```
1 /* ARC管理的Foundation对象 */
2 NSString *s1 = @"string"; /* 转换后依然由ARC管理释放 */
3 CFStringRef cfstring = (__bridge CFStringRef)s1;
4 /* 开发者手动管理的Core Foundation对象 */
5 CFStringRef s2 = CFStringCreateWithCString(NULL, "string",
  kCFStringEncodingASCII);/* 转换后仍然需要开发者手动管理释放 */
6 NSString *fstring = (__bridge NSString*)s2;
```

- _bridge_transfer：给予ARC所有权，CF -> Foundation。
- _bridge_retained：解除ARC所有权, Foundation ->CF。

ARC

- 自动引用计数（Automatic Reference Count 简称 ARC）。
- 引用计数：创建、新指针 +1；不在指向 -1；为0 回收。

- ARC工作原理：编译时，编译器分析每个对象的生命周期，添加对应的引用计数操作代码。
- 好处：相对于垃圾回收机制，没有运行时的额外开销。并且深度分析对象的生命周期，比人工效率更高，例如先 +1后 -1，会直接优化掉。
- 野指针：指向对象被回收的指针。
- 空指针：未指向一块有意义的内存区的指针。
- nil、NULL、Nil、NSNull
  - nil：用于oc对象。oc空指针调用方法不会crash，是因为oc方法调用是通过**objc_msgSend**，如果self为空则直接返回。
  - NULL：一般用于C指针。
  - Nil：用于oc类指针。

```
1 Class class = Nil;
```

  - NSNull：[NSNull null]返回一个空对象，一般用于集合类，用来表示一个空的对象。因为nil在集合内表示结尾。

copy和mutableCopy

- 遵守协议NSCopying、NSMutableCopying，分别实现copyWithZone、mutableCopyWithZone。
- 因为自定义的类，copy和mutableCopy都是要看你怎么实现，所以我们只需要研究NSString和系统容器类。
- NSString和不可变容器类
  - copy浅拷贝；
  - mutable深拷贝。
- NSMutableString和可变容器类
  - 都是深拷贝
  - copy后的是不可变对象，mutableCopy后的是可变对象。

Property修饰符

- copy、strong、weak、assign、；
  - copy：对象可变，跟strong一样；对象不可变，深拷贝。无论拷贝的对象是否可变，本身是否可变，最后的对象都不可变。
    - 属性是可变对象，可变对象和不可变对象赋值最后都是不可变对象。
  - strong：对象引用计数+1，浅拷贝。
  - weak：不持有所指对象所有权，引用计数不变，对象回收后引用本身为nil，避免野指针。

    -
    -
- readwrite、readonly;
- nonatomic/atomic;

- strong、retain都是浅拷贝；
- copy对不可变对象是浅拷贝，可变对象是深拷贝。
- copy修饰的属性， = 相当于 copy
- 对于集合类，不管可不可变，copy得到的是不可变对象
- 当我们声明一个属性str时，编译时，编译器会自动添加一个实例变量_str和get、set方法。添加实例变量的前提是没有同x名变量，否则不添加。
- @synthesize，可以修改默认的实例变量名

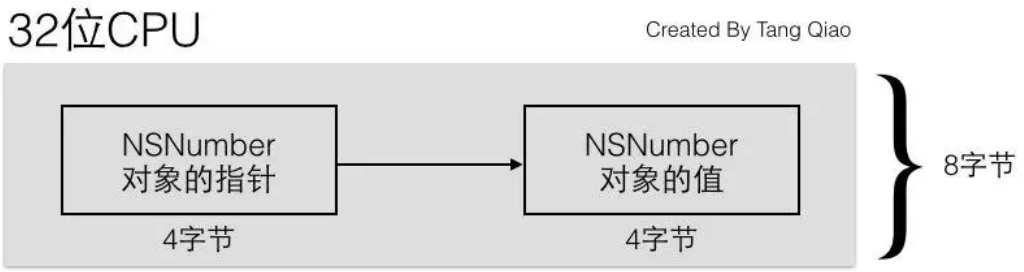
```
1 @synthesize str = mystr;
```

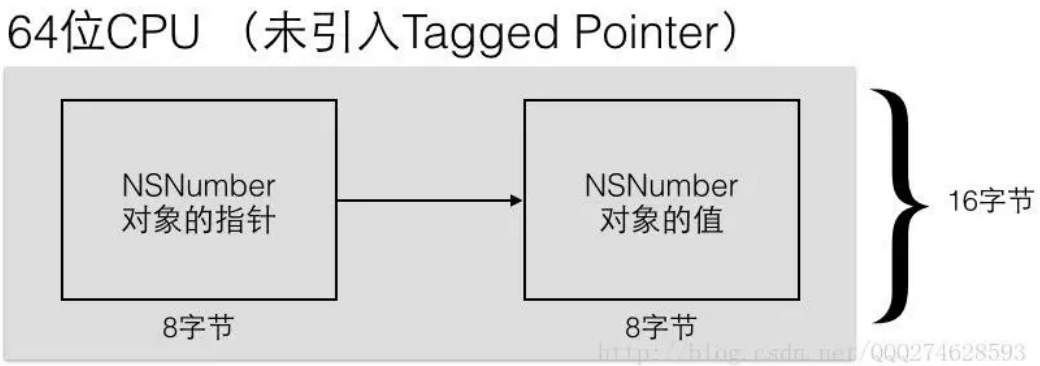
    - 在MRC模式下`@synthesize str`表示自动生成存取方法，ARC不管加不加都会默认生成
- `@dynamic`表示不默认生成存取方法

iOS引用计数管理方

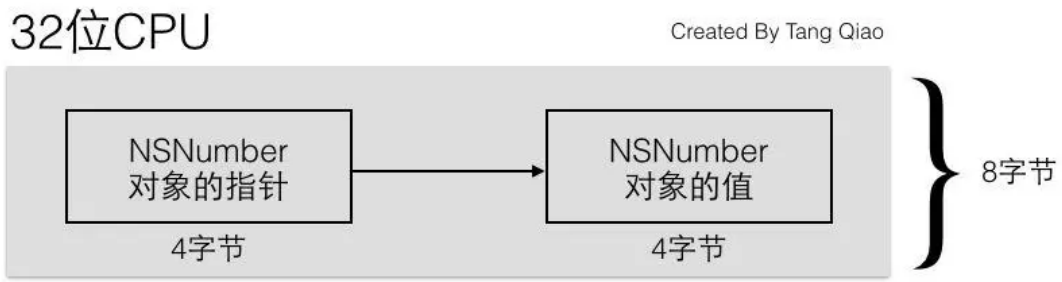
Tagged Pointer

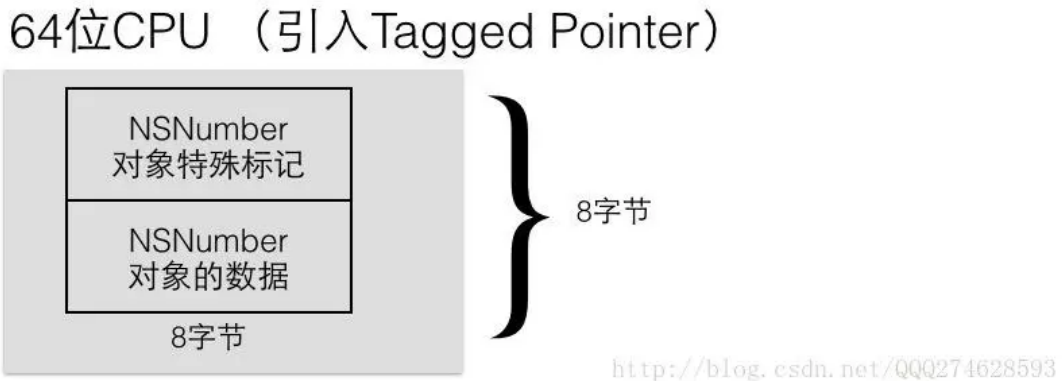
- 在iPhone5S采用了64位架构的双核处理器，指针和对象大小都和CPU位数有关，32位占4字节，64位占8字节。





- 为了存储和访问一个NSNumber对象，我们需要在堆上为其分配内存，另外还要维护引用计数、管理生命周期。
- 为了改进，苹果提出了**Tagged Pointer**。对于NSNumber，一般需要其本身值需要占用的内存通常不需要8个字节。所以我们可以把一个对象的指针拆成两部分，一部分直接保存数据，另一部分作为特殊标记，表示这是一个特殊指针，不指向任何一个地址。





- 从引用指针来看，这是一个释放不掉的单例常量对象，运行时根据情况创建。
- 如何判断一个指针是否为Tagged Pointer?
  - iOS平台，最高有效位是1（第64bit）

- Mac平台，最低有效位是1

nonpointer isa

isa占用64位，实际上存储类地址只要32位，为了提高利用率会存储一些其他消息，包括小型的引用计数。

```
/** isa_t 结构体 */union isa_t {
    Class cls;
    uintptr_t bits;
    struct {
        //0代表普通指针，存储Class/Meta-Class内存地址;
        //1代表优化过，存了更多的信息
        uintptr_t nonpointer        : 1;
        //是否有关联对象，如果没有，释放时更快
        uintptr_t has_assoc         : 1;
        //是否有C++的析构函数（对象销毁调用的函数），如果没有，释放时更快
        uintptr_t has_cxx_dtor      : 1;
        //Class、Meta-Class的内存地址
        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，如果没有释放会更快
        uintptr_t weakly_referenced : 1;
        //该对象是否正在释放
        uintptr_t deallocating      : 1;
        //对象引用计数，是实际计数-1。如果对象引用计数为10，extra_rc为9。如果引用计数大于10，则需要用到下面的has_sidetable_rc
        uintptr_t extra_rc          : 19;
        //当引用计数大于10，has_sidetable_rc为1，引用计数会存储在一个叫SideTable的类的属性中，这是一个散列表
        uintptr_t has_sidetable_rc  : 1;
    };};

```

散列表（引用计数表、弱引用表）

SideTables

- 一个长度为64的全局哈希表。里面存储了SideTable，key->对象内存地址，value->SideTable。
- 一个obj对应一个SideTable，一个SideTable对应多个obj。

SideTable

```
spinlock_t slock;           // 自旋锁，防止多线程访问冲突
RefcountMap refcnts;        // 对象引用计数map
weak_table_t weak_table;    // 对象弱引用map
```

- 自旋锁：在修改引用计数时上锁，因为操作非常快，所以使用自旋锁。
- 引用计数器RefcountMap refcnts：key->DisguisedPtr<objc_object>（下面会有解释），value->size_t保存引用计数。
    - `RefcountMap refcnts` 中通过一个`size_t`（64位系统中占用64位）来保存引用计数，其中1位用来存储固定标志位，在溢出的时候使用，一位表示正在释放中，一位表示是否有弱引用，其余位表示实际的引用计数。
- 弱引用表：weak功能实现的数据核心。

```
struct weak_table_t {
    weak_entry_t *weak_entries;         // hash数组，用来存储弱引用对象的相关信息weak_entry_t
    size_t    num_entries;              // hash数组中的元素个数
    uintptr_t mask;                     // hash数组长度-1，会参与hash计算。（注意，这里是hash数组的长度，而不是元素个数。比如，数组长度可能是64，而元素个数仅存了2个）
    uintptr_t max_hash_displacement;    // 可能会发生的hash冲突的最大次数，用于判断是否出现了逻辑错误（hash表中的冲突次数绝不会超过改值）
};
```

    - weak_entry_t:
        - `DisguisedPtr<objc_object> referent`：弱引用对象指针摘要。其实可以理解为弱引用对象的指针，只不过这里使用了摘要的形式存储。（所谓摘要，其实是把地址取负）。也可以理解为被弱引用的对象
        - union：接下来是一个联合，union有两种形式：定长数组weak_referrer_t inline_referrers[WEAK_INLINE_COUNT] 和 动态数组 weak_referrer_t *referrers。这两个数组是用来存储弱引用该对象的指针的指针的，同样也使用了指针摘要的形式存储。当弱引用该对象的指针数目小于等于WEAK_INLINE_COUNT时，使用定长数组。当超过WEAK_INLINE_COUNT时，会将定长数组中的元素转移到动态数组中，并之后都是用动态数组存储。

- bool out_of_line()： 该方法用来判断当前的weak_entry_t是使用的定长数组还是动态数组。当返回true，此时使用的动态数组，当返回false，使用静态数组。
- weak_entry_t& operator=(const weak_entry_t& other)： 赋值方法
- weak_entry_t(objc_object *newReferent, objc_object **newReferrer)： 构造方法。

Retain原理

- 如果是tagged pointer，直接返回指针值；
- 如果是普通指针，给sideTable里的refcnts对应value+1。
- 如果是nonpointer isa，增加isa的extra_rc。增加后如果溢出就把extra_rc一半引用计数存在sideTable的refcnts里，extra_rc减半。

Release原理

- 如果是tagged pointer，返回false；
- 如果是普通指针，找到对应的sidetable里的refcnts，计数-1，如果等于0就发送销毁消息；
- 如果是nonpointer isa，首先让extra_rc --，如果下溢出等于-1，从sideTable取回另一半计数，如果能借到就赋值给extra_rc，然后-1。如果借不到就发送销毁消息。（extra_rc比实际引用计数小1）

Block

原理

- 格式

```
返回值 (^block名字)(参数) = ^返回值(参数){
};
```

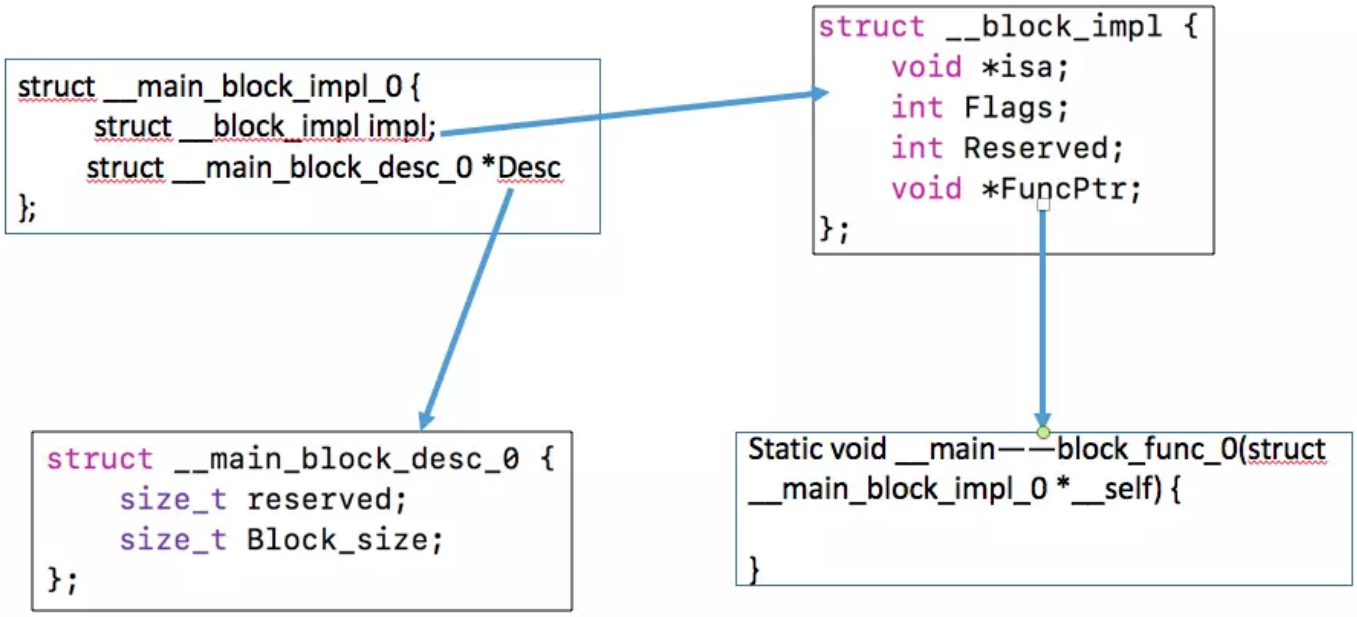
- 外部变量引用
  - 局部变量：值
  - 局部静态变量：指针
- Block结构体

```
struct __main_block_impl_0 {//Block的包装
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  //Block的构造函数
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int flags=0) {
    impl.isa = &_NSConcreteStackBlock;
```

```
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};

//真正的block
struct __block_impl {
  void *isa;//指向Block的类对象
  int Flags;
  int Reserved;//今后版本升级所需的区域大小
  void *FuncPtr;//block函数指针
};

//描述block块的size大小和今后版本升级所需的区域大小信息
static struct __main_block_desc_0 {
  size_t reserved; //今后版本升级所需的区域大小
  size_t Block_size;//block结构体大小
}

//block里的函数
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {

        (printf("hello world!"));
}
//__main_block_desc_0 变量
__main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
```

```
int main() {//代码
    void (^blk)(void) = ^{
        (printf("hello world!"));
    };
    blk();
    return 0;
```

```
}

//转化后
//(void (*)()
int main() {
    void (*blk)(void) = ((void (*)())&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA));
    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__block_impl *)blk);
    return 0;
}

//个人理解的转化
int main() {
    __main_block_impl_0 blk = &__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA)
    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__block_impl *)blk);
}
```



截取变量

```
int main() {
    int a = 1;
    void (^blk)(void) = ^{
        printf("%d", a);
    };
    blk();
    return 0;
}
```

- cpp重写后：

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  int a;//增加了一个变量
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int _a, int flags=0) : a(_a) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
    int a = __cself->a; // bound by copy
    printf("%d", a);
}
static struct __main_block_desc_0 {
  size_t reserved;
  size_t Block_size;
}
__main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};

int main() {
```

```
    int a = 1;
    void (*blk)(void) = ((void (*)())&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA, a));
    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__block_impl *)blk);
    return 0;
}
```

- __main_block_impl_0结构体加入使用的外部变量，构造函数增加变量参数；
- _main_block_func_0 函数增加了使用的变量；
- 如果block使用的是静态局部变量，那么加入的变量就是指针类型。
- 如果引用了当前对象的属性，那么就会引用self。

- 加了__block后

```
int main() {
    __block int number = 1;
    void (^blk)(void) = ^{
        number = 3;
        printf("%d", number);
    };
    blk();
    return 0;
}
```

```
struct __Block_byref_number_0 {
  void *__isa;
__Block_byref_number_0 *__forwarding; //number是变量名
 int __flags;
 int __size;
 int number;
};
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  __Block_byref_number_0 *number; // by ref
```

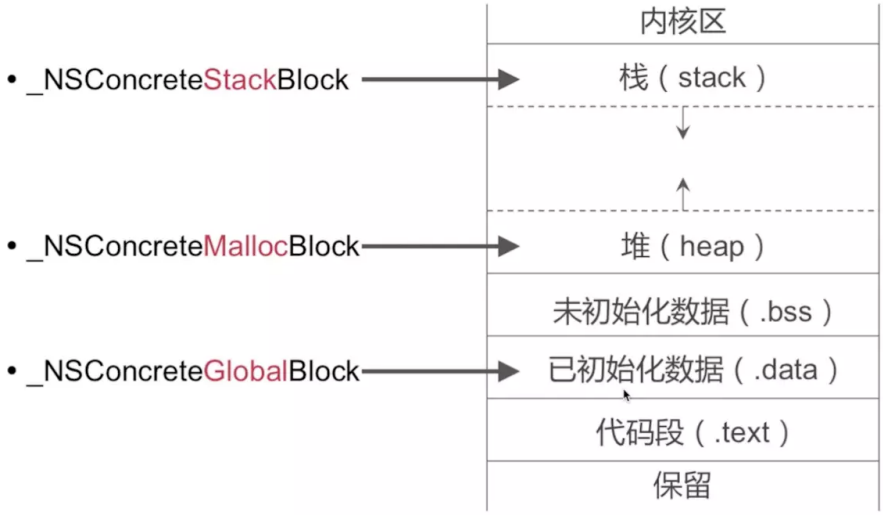
```
12    __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc,
    __Block_byref_number_0 *_number, int flags=0) : number(_number->__forwarding) {
13      impl.isa = &_NSConcreteStackBlock;
14      impl.Flags = flags;
15      impl.FuncPtr = fp;
16      Desc = desc;
17    }
18  };
19  static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
20    __Block_byref_number_0 *number = __cself->number; // bound by ref
21    (number->__forwarding->number) = 3;
22    printf("%d", (number->__forwarding->number));
23      }
24  static void __main_block_copy_0(struct __main_block_impl_0*dst, struct
    __main_block_impl_0*src) {
25      _Block_object_assign((void*)&dst->number,
26      (void*)src->number,
27      8/*BLOCK_FIELD_IS_BYREF*/);
28  }
29
30  static void __main_block_dispose_0(struct __main_block_impl_0*src) {
31      _Block_object_dispose((void*)src->number,
32      8/*BLOCK_FIELD_IS_BYREF*/);
33  }
34
35  static struct __main_block_desc_0 {
36    size_t reserved;
37    size_t Block_size;
38    void (*copy)(struct __main_block_impl_0*, struct __main_block_impl_0*);
39    void (*dispose)(struct __main_block_impl_0*);
40  }
41
42  __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0),
    __main_block_copy_0, __main_block_dispose_0};
43
```

```
int main() {
    __attribute__((__blocks__(byref))) __Block_byref_number_0 number = {(void*)0, (__Block_byref_number_0 *)&number, 0, sizeof(__Block_byref_number_0), 1};

    void (*blk)(void) = ((void (*)())&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA, (__Block_byref_number_0 *)&number, 570425344));

    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__block_impl *)blk);
    return 0;
}
```

- 会为__block变量创建一个结构体，里面有个相同的变量。结构体内的_forwarding指针指向真正的结构体，因为可能会从栈复制到堆上。
- 引用变量会把指针传进来。

Block类型

- _NSConcreteGlobalBlock 全局：在block内部没有引用任何外部变量
  - 当我们声明一个block时，如果这个block没有捕获外部的变量，那么这个block就位于全局区，此时对NSGlobalBlock的retain、copy、release操作都无效。ARC和MRC环境下都是如此。
- _NSConcreteStackBlock 栈类型
  - 引用了外部变量，且用__weak修饰。
- _NSConcreteMallocBlock 堆类型
  - __strong（默认）修饰，引用外部变量。
- 在ARC环境下如果默认修饰符而且捕获了外部变量，实际上是一个_NSConcreteStackBlock->_NSConcreteMallocBlock的类型转换过程，系统帮我们完成了copy操作，将block从栈迁移到堆区。



循环引用

为什么不在block最后置nil?

因为这样的话，如果不调用这个block，循环引用就会一直存在。

为什么__weak要配合__strong使用?

__strong是为了防止在执行block代码过程中，引用的变量被释放。有可能在执行前就被释放。

变量引用问题

```
1     NSObject *a = [NSObject new];
2     NSLog(@"a1 --%p",&a);
3     void (^test)(void) = ^{
4     NSLog(@"a2 --%p",&a);
5     };
6     NSLog(@"a3 --%p",&a);
7
```

- 结果： a1 = a3 != a2
- a1=a3很正常，!=a2的原因要参考源码，a作为方法参数传入，所以指针被复制了。

```
__block NSObject *a = [NSObject new];
  NSLog(@"a1 --%p",&a);
  void (^test)(void) = ^{
  NSLog(@"a2 --%p",&a);
  };
  NSLog(@"a3 --%p",&a);
```

- 结果：a1!=a2=a3;
- 在这里，a实际上是a->forwarding->a。
  - 因为引用外部变量，所以block会从栈复制到堆。而指针a现在其实是一个结构体，位于栈上，所以也要被一起复制到堆上。栈上的结构体a的forwarding一开始指向栈上的自己，后面指向堆上的自己。所以a1!=a2=a3，a1在栈上。
- 如果用__weak修饰block变量，则不会造成从栈到堆的迁移，所以a1=a2=a3。

为什么不能直接修改变量

- 因为block里的指针是复制的，并不会影响外面的指针。就好像修改形参，不会影响实参一样。
- 为什么使用了__block就可以，因为__block实际上创建了一个包裹着变量的结构体。这就好像你不能修改指针，但你可以修改指针所指向的对象的指针一样。
- 对象a。对象A，里面有对象a。你不能直接a=nil;但你可以A.a = nil;

Runtime

Class和Object

- Runtime是一个库，使我们能在运行时创建、检查和修改对象。
- 类其实是结构体的指针。

```
struct objc_class {
    Class isa;                                  // 元类对象
    Class super_class;                          // 父类
    const char * name;                          // 类名
    long version;                               // 类的版本信息，默认为0
    long info;                                  // 类信息，供运行期使用的一些位标识
    long instance_size;                         // 该类的实例变量大小
    struct objc_ivar_list * ivars;              // 该类的成员变量链表
    struct objc_method_list ** methodLists;     // 方法定义的链表
```

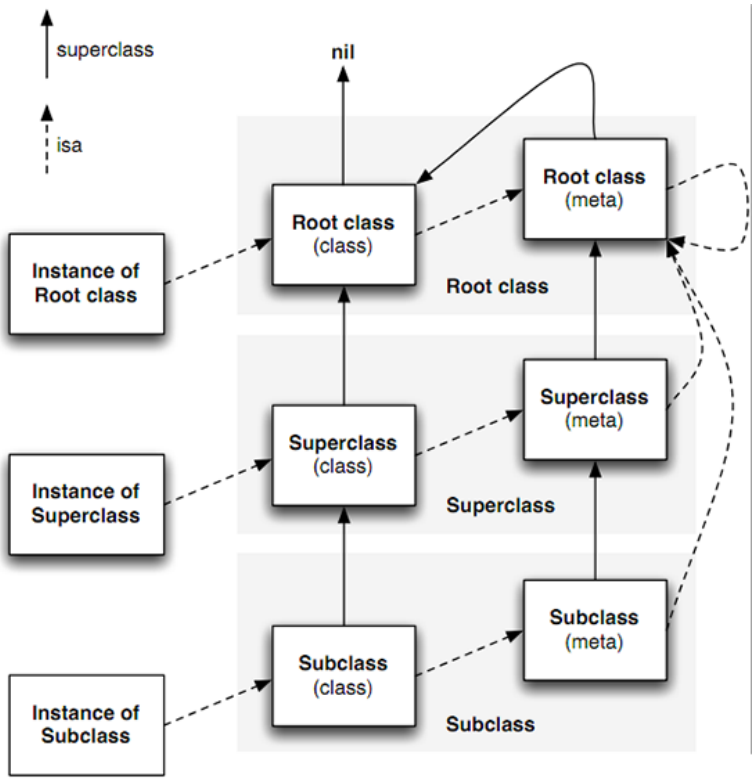
```
    struct objc_cache * cache;                  // 调用过的方法，节约时间
    struct objc_protocol_list * protocols;      // 协议链表
};typedef struct objc_class *Class
```

- id其实就是对象(objc_objc)的指针，里面只有一个类对象。

```
struct objc_object {
    Class isa;
};typedef struct objc_object *id;
```

Meta Class （元类）

- 类也是对象，所以类对象的类就是元类。
- 类中保存了对象的方法/变量，元类保存了类的方法/变量，也就是静态方法和静态变量。
- 对象调用方法/对象通过isa找到类的地址；类调用方法/对象通过isa找到元类的地址
- 元类也是对象，为了防止循环下去，所以元类的isa指向基类（不是父类）的元类，而基类的元类指向自己。



Method

```
typedef struct objc_method *Method;
struct objc_method {
    SEL method_name;
```

```
    char * method_types;
    IMP method_imp;
};
```

- Method是objc_metho，也就是方法的指针。
- SEL可以理解为方法名的字符串，由于一个 Method 只保存了方法的方法名，并最终要根据方法名来查找方法的实现，所以oc无法重载方法。
- IMP是函数指针。参数列表，id是调用的对象，实例方法是对象地址；类方法是类对象地址；SEL是方法名；省略号是参数。

```
// SEL
typedef struct objc_selector *SEL;
// IMP
typedef id (*IMP)(id, SEL, ...);
```

- 方法调用会转化成消息发送，主要使用**objc_msgSend**函数。

```
id objc_msgSend(id self, SEL op, ...);
[self doSomething];//=objc_msgSend(self, @selector(doSomething));
```

- objc_msgSend，传入了默认id和SEL，分别是**self**和** _cmd**，分别代表**对象/类对象地址**和**方法名字符串**。

方法调用流程

- msg_msgSend()返回值是void，但是编译器会对它进行强转。

```
id obj = ((NSObject *(*)(id, SEL))(void *)objc_msgSend)
((id)objc_getClass("NSObject"), sel_registerName("alloc"));
obj = ((id (*)(id, SEL))(void *)objc_msgSend)((id)obj, sel_registerName("init"));
```

- 首先检查这个**selector**是不是要忽略。比如Mac OS X开发，有了垃圾回收后就不理会retain、release函数。
- 检测id是不是nil，是就忽略。
- 在类中的cache查找，找到了就跳到对应的函数去执行。
- cache找不到就去类中的method_list找。
- 还找不到就去父类继续上两步操作。
- 直到NSObject还没找到，就走拦截调用和消息转发流程。

- 首先会调用resolveClassMethod/resolveInstanceMethod，在这里动态添加方法，添加成功返回YES，否则NO。（实际上添加成功，返回什么都没关系）
- forwardingTargetForSelector，快速转发阶段，返回能够响应方法的对象。（类方法不调用）
- 如果 forwardingTargetForSelector 返回nil或者self，则进入第三步常规转发阶段。分两个阶段：
  - methodSignatureForSelector，返回SEL方法签名。
  - 如果返回有方法签名，则进入最后一步。forwardInvocation，在这里通过NSInvocation的invokeWithTarget调用方法。
  - 应用：
    - 为@dynamic实现方法
    - 实现多重代理
    - 间接实现多重继承

Category(分类)

```
struct objc_category {
    char * _Nonnull category_name                          OBJC2_UNAVAILABLE;
    char * _Nonnull class_name                             OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable instance_methods   OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable class_methods      OBJC2_UNAVAILABLE;
    struct objc_protocol_list * _Nullable protocols        OBJC2_UNAVAILABLE;
}                                                          OBJC2_UNAVAILABLE;
```

Category原理

- 编译时会生成一个category_t结构体，将这个结构体保存下来。
- 在运行时，RunTime会拿到我们保存下来的category_t
- 然后将category_t的各种列表都加到主类
- 在这里，因为方法列表是插入到前面，所以虽然没有覆盖原来的方法，但消息发送时会优先调用前面的，所以会造成一种覆盖的假象。

作用

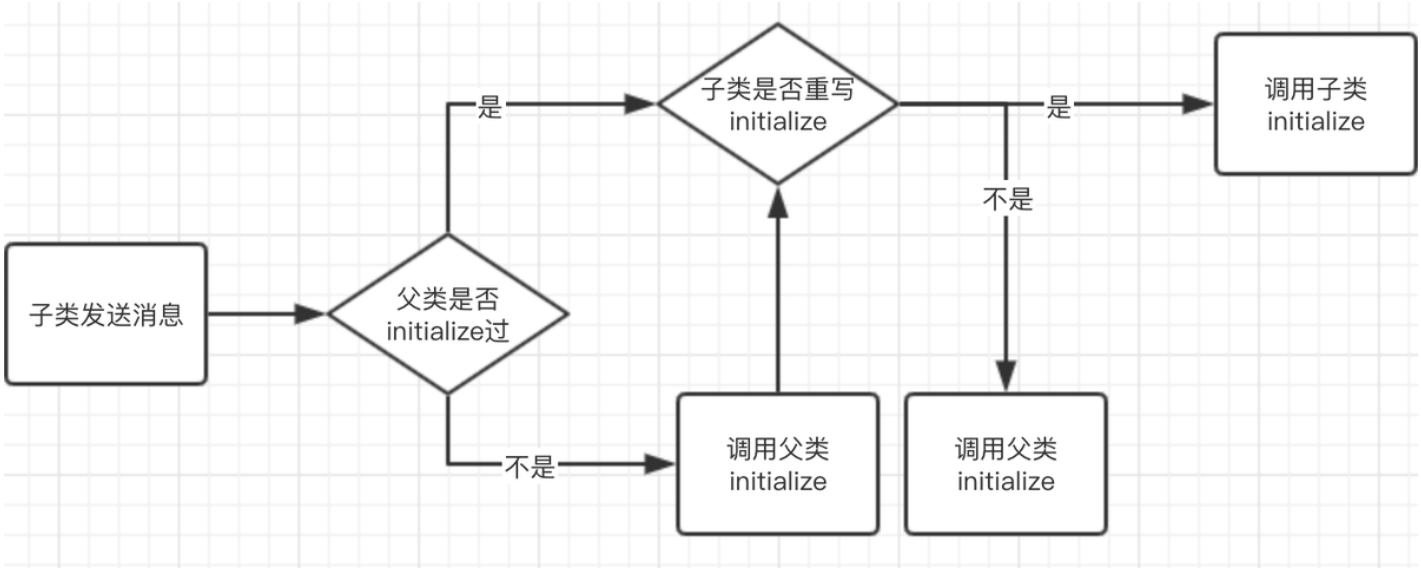
- 声明私有方法；
- 分解庞大的类文件；
- 把Framework私有方法公开；
- 修改系统方法；

Category和Extension

- extension在编译期决定，是类的一部分；
- category在运行期决定，所以无法添加变量。因为运行期，对象的内存布局已经确定，如果添加实例变量就会破坏内存布局，这对编译型语言来说是灾难的。
- 为什么可以添加方法属性不能添加变量?
  - 因为方法和属性不属于类实例，而成员变量属于类实例。类实例指的是一块内存区域，包含了isa指针和所有的成员变量。假如允许修改类实例，那就不符合类的定义，是无效对象。但方法是在objc_class中管理的，不会影响类实例。

load与initialize

- 调用时机
  - load：加载类时。iOS应用在启动时就会加载所有类，所以一启动就会调用load
  - initialize：向该类发送第一个消息时。
- 调用顺序
  - load：先调用父类，再调用子类。父类和子类的分类调用顺序根据编译顺序。
  a. initialize：先调用父类，再调用子类。分类会覆盖本类。
- 作用：
  - `load`方法通常用来进行Method Swizzle
  - `initialize`方法一般用于初始化全局变量或静态变量。
- 子类initialize与父类关系



关联对象

使用

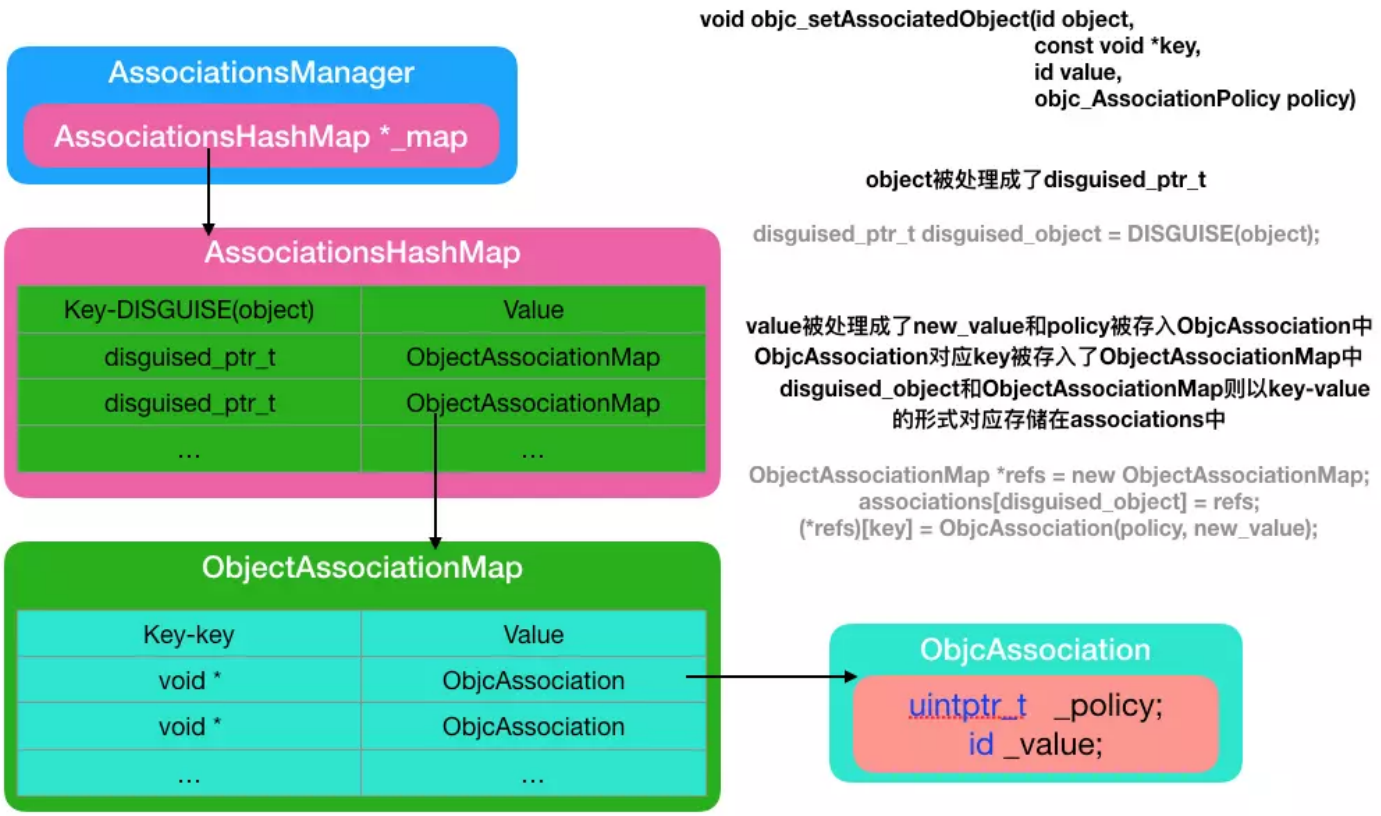
```
-(void)setName:(NSString *)name{
    objc_setAssociatedObject(self, @"name",name, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
-(NSString *)name{
    return objc_getAssociatedObject(self, @"name");
}
```

```
objc_setAssociatedObject(id object, const void *key, id value, objc_AssociationPolicy policy);
typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,  // 指定一个弱引用相关联的对象
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, // 指定相关对象的强引用，非原子性
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,  // 指定相关的对象被复制，非原子性
    OBJC_ASSOCIATION_RETAIN = 01401,  // 指定相关对象的强引用，原子性
    OBJC_ASSOCIATION_COPY = 01403     // 指定相关的对象被复制，原子性    };

```

- id：关联的对象
- key、value：键值对。（void * 代表任意类型的指针）
- policy：关联策略

```
objc_getAssociatedObject(id object, const void *key);
```

原理



- 关联对象并不是存储在被关联对象内，而是存在全局统一的AssociationsManager中。
- AssociationsManager里有一个AssociationsHashMap，key是被关联对象，value是ObjectAssociationMap；ObjectAssociationMap的key是key，value是关联对象和关联策略。

Method Swizzling 方法重排

```
Method originalMethod = class_getInstanceMethod([Test class], @selector(t));
Method newMethod = class_getInstanceMethod([ViewController class], @selector(hook));
//交换方法
method_exchangeImplementations(originalMethod, newMethod);
//设置方法
method_setImplementation(originalMethod, class_getMethodImplementation([self class], @selector(hook)));
```

```
//替换方法
class_replaceMethod([Test class], @selector(t),
class_getMethodImplementation([self class], @selector(hook)), "v@:");
```

```
struct objc_method {
    SEL method_name;
    char * method_types;
    IMP method_imp;
};
```

- 方法交换只是改变了method_imp，如果原来方法依赖_cmd参数，就会出现错误；这种情况是originalImp依赖originalSelector，交换后想要调用originalImp，这时originalImp的_cmd是newSelector。

```
Method originalMethod = class_getInstanceMethod([self class],
@selector(originalMethod));
Method newMethod = class_getInstanceMethod([self class],
@selector(replacementMethod));
__original_Method_Imp = class_getMethodImplementation([self class],
@selector(originalMethod));
method_exchangeImplementations(originalMethod, newMethod);
```

- 首先获得原方法实现，然后交换方法。

```
- (int) replacementMethod
{
    assert([NSStringFromSelector(_cmd) isEqualToString:@"originalMethod"]);
     //code
    int returnValue = ((int(*)(id,SEL))__original_Method_Imp)(self, _cmd);
    return returnValue + 1;
}
```

- 然后在replacementMethod里调用originalMethod，这样最后调用交换后的originalMethod时，调用原实现，selector还是原来的。

获取所有属性和方法

- 成员变量（Ivar），只会获得当前类的公有和私有属性，不会获得父类的。

```objc
- (void)printIvar{
    unsigned int count;
    Ivar *ivars = class_copyIvarList([self class], &count);
    for (int i = 0; i < count; i++) {
        // 取出i位置的成员变量
        Ivar ivar = ivars[i];
        NSLog(@"变量名：%s 变量类型：%s", ivar_getName(ivar), ivar_getTypeEncoding(ivar));
    }
    free(ivars);
}
```

- 属性（Property）

```objc
- (void)printProperty{
    unsigned int count;
    objc_property_t *propertyList = class_copyPropertyList([self class], &count);
    for (int i = 0; i < count; i++) {
        // 取出i位置的成员变量
        objc_property_t property = propertyList[i];
        NSLog(@"属性名：%s 属性类型：%s", property_getName(property), property_getAttributes(property));
    }
    free(propertyList);
}
```

- 方法（Method）

```objc
- (void)printMethod{
    unsigned int count;
      Method *methodList = class_copyMethodList([self class], &count);
      for (int i = 0; i < count; i++) {
          Method method = methodList[i];
          method_getXXX(method)//获得方法信息
      }
      free(methodList);
}
```

- 协议（Protocol）

```objc
- (void)printProtocol{
    unsigned int count;
    Protocol *protocolList = class_copyProtocolList([self class], &count);
    for (int i = 0; i < count; i++) {
        Protocol protocol = protocolList[i];
        protocol_getXX(...)//获得协议信息
    }
    free(protocolList);
}
```

RunTime的应用场景

1.AOP

```objc
Method originalMethod = class_getInstanceMethod(originalClass, originalSelector);
Method newMethod = class_getInstanceMethod(replaceClass, replaceSelector);
class_addMethod(originalClass, originalSelector,
method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod));
originalMethod = class_getInstanceMethod(originalClass, originalSelector);
method_exchangeImplementations(originalMethod, newMethod);

//要替换的方法
- (void)replaceWillAppear:(BOOL)animated{
    [self replaceWillAppear:animated];
    NSLog(@"测试");
}
```

先当前类添加要被替换的方法，因为如果父类有而子类没有，替换结束后

```objc
    [self replaceWillAppear:animated];
```

会在父类调用，这样就等于是父类调用子类的方法，就会报错。

2.字典转模型

KVC的setValuesForKeysWithDictionary可以集中设置设置属性，但是需要一一对应，否则就会报错。
但我们可以利用runtime获得类的所有属性名，从字典中取出值，再利用KVC setValue:forKey来设置值，这样就保证了不会出错。

3.归解档

- 归档：将对象序列化存入沙盒文件的过程，会调用encodeWithCoder:来序列化；
- 解档：将沙盒文件中的数据反序列化读入内存的过程，会调用initWithCoder:来反序列化。

4.逆向工程

5.热修复

KVC（Key-Value Coding）

- kvc指可以直接通过key直接访问对象属性，直接赋值，而不需要明确调用存取方法。这样就可以运行时动态修改，而不用再编译时确定。
- kvc可以用来修改.m文件中声明的属性

KVC设值

- 修改value顺序，如：setValue: forKey:@"name"
  - 首先查看有没有setName方法，有就调用setName修改；
  - 没有就查看+ (BOOL)accessInstanceVariablesDirectly，默认返回YES。如果返回NO，就会直接调用setValue: forUndefinedKey，默认抛出异常；
  - 如果返回YES，按照_key，_iskey，key，iskey的顺序搜索成员并进行赋值操作
- 简单来说就是`如果没有找到Set<Key>方法的话，会按照_key，_iskey，key，iskey的顺序搜索成员并进行赋值操作`。

KVC取值

- 获得value顺序，如：valueForKey:@"name"
  - 首先按照get<Key>、<key>、is<Key>的顺序查找，找到就直接调用。如果是值类型，会包装成NSNumber对象；
  - 没找到就尝试寻找**countOfKey、objectInKeyAtIndex:、keyAtIndexes**，如果找到了第一个和剩下两个中的一个，就返回一个可以响应NSArray所有方法的代理集合。~~给这个集合发消~~

~~息，就会以**countOfKey、objectInKeyAtIndex:、keyAtIndexes**组合的形式调用~~

- 如果还没找到，就尝试寻找**countOfKey、enumeratorOfKey和memberOfKey**。如果三个都找到，就那么就返回一个可以响应NSSet所有的方法的代理集合。~~给这个代理集合发消息也会以countOfKey、enumeratorOfKey和memberOfKey组合的形式调用。~~
- 如果还没找到，判断accessInstanceVariablesDirectly，为NO，调用valueForUndefinedKey，默认抛出异常；
- 为YES，按照_key、_isKey、key、isKey的顺序查找成员变量。
- 还没找到，调用valueForUndefinedKey。

KVO（Key-Value Observing）

定义

是oc对观察者模式的实现，每次当被观察对象的某个属性值发生改变，注册的观察者就能得到通知。

```
+(BOOL)automaticallyNotifiesObserversForKey:(NSString *)key{
//手动触发
return NO;
//自动触发
// return YES;}

```

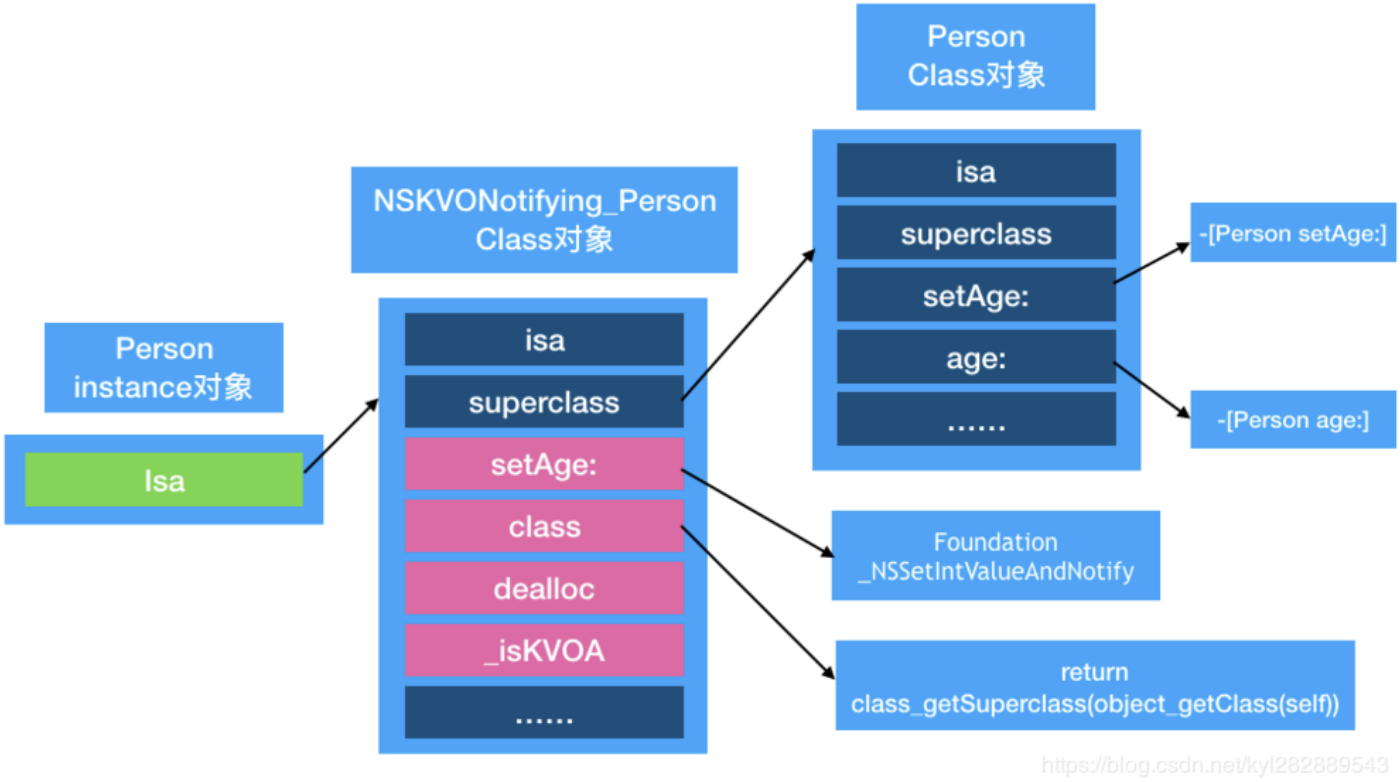
```
static int a;
//手动触发
//即将改变
[_p willChangeValueForKey:@"name"]
_p.name = [NSString stringWithFormat:@"%d",a++];
// 改变完成
[_p didChangeValueForKey:@"name"];
```

属性是对象，对象有很多属性，直接监听该对象，要在持有该对象的类中重写

```objc

+ (NSSet<NSString *> *)keyPathsForValuesAffectingValueForKey:(NSString *)key {
    NSSet *keyPaths = [super keyPathsForValuesAffectingValueForKey:key];
    if ([key isEqualToString:@"cat"]) {
        NSArray *arrKeyPaths = @[@"_cat.age", @"_cat.kind"];
        keyPaths = [keyPaths setByAddingObjectsFromArray:arrKeyPaths];
    }
    return keyPaths;
}
```

原理

对象 A a调用addObserver后，就会创建一个继承自A的类NSKVONotifying_A。这里因为观察的是age，所以会重写setAge:方法。而且还会重写class方法，为了不让外界察觉。



load与initialize

load

- 只要被添加到runtime就一定会调用，而且只会调用一次，子类不会覆盖父类；
- 全部都会调用，分类不会覆盖类
- 调用顺序：父类->类->分类->父分类

initialize

- 在收到第一条消息前被调用，包括实例方法和类方法；
- 会先调用父类，然后调用子类；
- 分类会覆盖类，调用顺序为父分类->父类->分类->类

- 子类没有实现load方法，不会调用父类的load方法；子类没有实现initialize方法，也会自动调用父类的initialize;
- load一般用于method swizzle
- initialize一般用于对一些不方便在编译期初始化的对象赋值，或者对一些静态常量进行初始化操作；

RunLoop

深入理解RunLoop

一个线程只能执行一个任务，执行完毕了就会退出。但是我们的程序显然不能退出，所以我们需要一个机制，让线程能够不断处理事件也不退出。代码逻辑如下：

```
function loop() {
    initialize();
    do {
        var message = get_next_message();
        process_message(message);
    } while (message != quit);
}
```

这种模型被称作Event Loop。实现这种模型的关键点在于：

- 如何管理消息/事件
- 如何让线程在没有消息时休眠以避免资源浪费、在有消息到来时立刻被唤醒

RunLoop实际上就是一个对象，这个对象管理了需要处理的消息和事件，并提供一个入口函数来执行上面Event Loop的逻辑。

iOS中提供了两个这样的对象：**NSRunLoop和CFRunLoopRef。**

CFRunLoopRef是在CoreFoundation框架内的，提供了纯C函数的API，而且是线程安全的。

NSRunLoop是基于CFRunLoopRef的封装，但不是线程安全的。

RunLoop与线程的关系

- 不能直接创建RunLoop，只能通过**CFRunLoopGetMain()**和**CFRunLoopGetCurrent()**自动获取。
- **CFRunLoopGetMain()**和**CFRunLoopGetCurrent()** 内部分别传入主线程和当前线程作为参数。
- 一个静态字典保存，key：thread，value：CFRunLoopRef
- 获得CFRunLoopRef的方法过程
  a. 获得锁，防止并发访问出现错误；
  b. 如果字典为空，创建字典，创建mainLoop，mainThread为key mainLoop为value存入字典；
  c. 通过线程参数获得RunLoopRef；
  d. 如果RunLoopRef为空，创建并存入字典并且注册回调，当线程销毁时也会销毁RunLoopRef；(主线程未添加该回调)
  e. 释放锁，返回线程对应的RunLoopRef。
- 线程与RunLoop是一一对应关系，如果不主动获取就不会主动创建；
- RunLoop在第一次获取时创建，在线程结束时销毁；
- 只能在一个线程的内部获取RunLoop。

```
/// 全局的Dictionary，key 是 pthread_t，  value 是 CFRunLoopRef
static CFMutableDictionaryRef loopsDic;
/// 访问 loopsDic 时的锁
static CFSpinLock_t loopsLock;

/// 获取一个 pthread 对应的 RunLoop。
CFRunLoopRef _CFRunLoopGet(pthread_t thread) {
    OSSpinLockLock(&loopsLock);

if (!loopsDic) {
    // 第一次进入时，初始化全局Dic，并先为主线程创建一个 RunLoop。
    loopsDic = CFDictionaryCreateMutable();
    CFRunLoopRef mainLoop = _CFRunLoopCreate();
    CFDictionarySetValue(loopsDic, pthread_main_thread_np(), mainLoop);
}

/// 直接从 Dictionary 里获取。
CFRunLoopRef loop = CFDictionaryGetValue(loopsDic, thread));

if (!loop) {
    /// 取不到时，创建一个
    loop = _CFRunLoopCreate();
```

```
        CFDictionarySetValue(loopsDic, thread, loop);
        /// 注册一个回调，当线程销毁时，顺便也销毁其对应的 RunLoop。
        _CFSetTSD(..., thread, loop, __CFFinalizeRunLoop);
    }

    OSSpinLockUnLock(&loopsLock);
    return loop;
}

CFRunLoopRef CFRunLoopGetMain() {
    return _CFRunLoopGet(pthread_main_thread_np());
}

CFRunLoopRef CFRunLoopGetCurrent() {
    return _CFRunLoopGet(pthread_self());
}
```

RunLoop的结构

RunLoop种类

- CFRunLoopRef
- CFRunLoopModeRef
- CFRunLoopSourceRef
  - 事件产生的地方，有两种。
    - Source0：只包含一个回调(函数指针)，不能主动触发。使用时需要先调用CFRunLoopSourceSignal(source)将这个source标记为待处理，然后手动调用CFRunLoopWakeUp(runloop)来唤醒RunLoop，让其处理事件。
    - Source1：包含一个mach_port和一个回调(函数指针)，被用于通过内核和其他线程发送消息。能主动唤醒RunLoop。
- CFRunLoopTimerRef：基于时间的触发器，它和 NSTimer 是toll-free bridged 的，可以混用。包含一个时间长度和回调。当其加入到RunLoop，RunLoop会注册对应的时间点，当时间点到，RunLoop会被唤醒以执行那个回调。
- CFRunLoopObserverRef：是观察者，每个观察者都包含一个回调，当RunLoop的状态发生变化时，观察者就能通过回调接受到变化。可以观测的时间点有：

```
typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
    kCFRunLoopEntry         = (1UL << 0), // 即将进入Loop
    kCFRunLoopBeforeTimers  = (1UL << 1), // 即将处理 Timer
    kCFRunLoopBeforeSources = (1UL << 2), // 即将处理 Source
    kCFRunLoopBeforeWaiting = (1UL << 5), // 即将进入休眠
    kCFRunLoopAfterWaiting  = (1UL << 6), // 刚从休眠中唤醒
    kCFRunLoopExit          = (1UL << 7), // 即将退出Loop
};
```

- Source/Timer/Observer被统称为mode item，一个item可以被同时加入多个mode。如果一个mode没有一个item则会直接退出，不进入循环。

CFRunLoop和Mode结构

```
struct __CFRunLoopMode {
    CFStringRef _name;            // Mode Name, 例如 @"kCFRunLoopDefaultMode"
    CFMutableSetRef _sources0;    // Set
    CFMutableSetRef _sources1;    // Set
    CFMutableArrayRef _observers; // Array
    CFMutableArrayRef _timers;    // Array
    ...
};

struct __CFRunLoop {
    CFMutableSetRef _commonModes;     // Set
    CFMutableSetRef _commonModeItems; // Set<Source/Observer/Timer>
    CFRunLoopModeRef _currentMode;    // Current Runloop Mode
    CFMutableSetRef _modes;           // Set
    ...
};
```

- CommonModes：Mode可以将自己标记为**Common**，通过把modeName加入CFRunLoop的_commonModes。没当CFRunLoop的内容发生变化时，就会自动把_commonModeItems里的Source/Timer/Observer同步到具有**Common**标记的Mode里。

- 主线程有两个预置Mode：kCFRunLoopDefaultMode和UITrackingRunLoopMode。
  - kCFRunLoopDefaultMode：App平时状态；
  - UITrackingRunLoopMode：追踪ScrollView滑动时的状态

> 应用场景举例：主线程的 RunLoop 里有两个预置的 Mode：kCFRunLoopDefaultMode 和 UITrackingRunLoopMode。这两个 Mode 都已经被标记为”Common”属性。DefaultMode 是 App 平时所处的状态，TrackingRunLoopMode 是追踪 ScrollView 滑动时的状态。当你创建一个 Timer 并加到 DefaultMode 时，Timer 会得到重复回调，但此时滑动一个TableView时，RunLoop 会将 mode 切换为 TrackingRunLoopMode，这时 Timer 就不会被回调，并且也不会影响到滑动操作。
>
> 有时你需要一个 Timer，在两个 Mode 中都能得到回调，一种办法就是将这个 Timer 分别加入这两个 Mode。还有一种方式，就是将 Timer 加入到顶层的 RunLoop 的 “commonModeItems” 中。”commonModeItems” 被 RunLoop 自动更新到所有具有”Common”属性的 Mode 里去。

- CFRunLoop操作Mode的接口

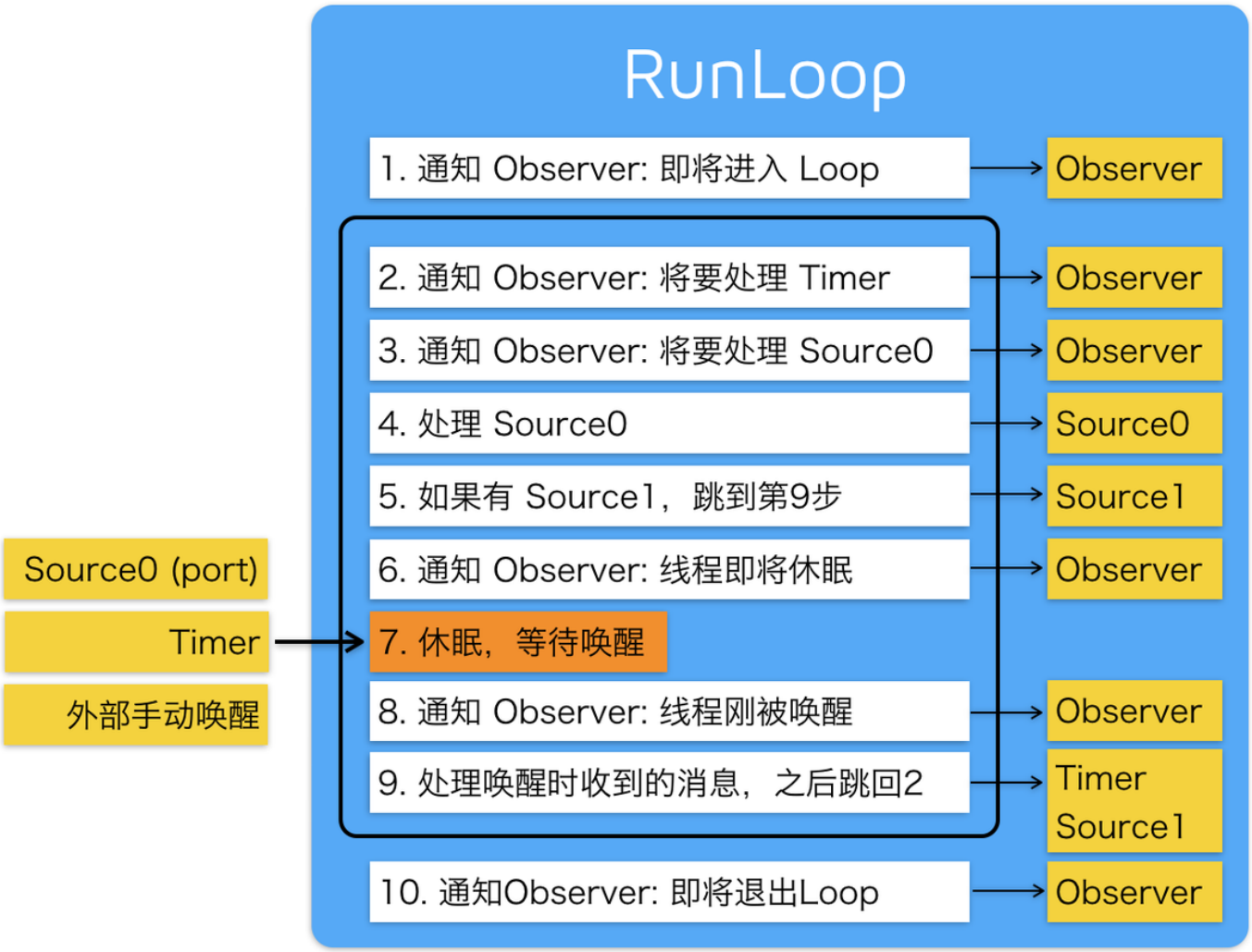
```
1 CFRunLoopAddCommonMode(CFRunLoopRef runloop, CFStringRef modeName);
2 CFRunLoopRunInMode(CFStringRef modeName, ...);
```

- Mode暴露ModeItem的接口

```
1 CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef
  modeName);
2 CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef
  modeName);
3 CFRunLoopAddTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);
4 CFRunLoopRemoveSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef
  modeName);
5 CFRunLoopRemoveObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer,
  CFStringRef modeName);
6 CFRunLoopRemoveTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);
```

- 只能通过modeName操作mode，当往RunLoop传入一个没有的modeName会自动创建CFRunLoopModeRef，只能增加不能删除

RunLoop内部逻辑




```
/// 用DefaultMode启动
void CFRunLoopRun(void) {
    CFRunLoopRunSpecific(CFRunLoopGetCurrent(), kCFRunLoopDefaultMode, 1.0e10, false);
}

/// 用指定的Mode启动，允许设置RunLoop超时时间
int CFRunLoopRunInMode(CFStringRef modeName, CFTimeInterval seconds, Boolean stopAfterHandle) {
    return CFRunLoopRunSpecific(CFRunLoopGetCurrent(), modeName, seconds, returnAfterSourceHandled);
}

/// RunLoop的实现
int CFRunLoopRunSpecific(runloop, modeName, seconds, stopAfterHandle) {
```

```
13
14      /// 首先根据modeName找到对应mode
15      CFRunLoopModeRef currentMode = __CFRunLoopFindMode(runloop, modeName, false);
16      /// 如果mode里没有source/timer/observer, 直接返回。
17      if (__CFRunLoopModeIsEmpty(currentMode)) return;
18
19      /// 1. 通知 Observers: RunLoop 即将进入 loop。
20      __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopEntry);
21
22      /// 内部函数，进入loop
23      __CFRunLoopRun(runloop, currentMode, seconds, returnAfterSourceHandled) {
24
25          Boolean sourceHandledThisLoop = NO;
26          int retVal = 0;
27          do {
28
29              /// 2. 通知 Observers: RunLoop 即将触发 Timer 回调。
30              __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeTimers);
31              /// 3. 通知 Observers: RunLoop 即将触发 Source0 (非port) 回调。
32              __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeSources);
33              /// 执行被加入的block
34              __CFRunLoopDoBlocks(runloop, currentMode);
35
36              /// 4. RunLoop 触发 Source0 (非port) 回调。
37              sourceHandledThisLoop = __CFRunLoopDoSources0(runloop, currentMode,
    stopAfterHandle);
38              /// 执行被加入的block
39              __CFRunLoopDoBlocks(runloop, currentMode);
40
41              /// 5. 如果有 Source1 (基于port) 处于 ready 状态，直接处理这个 Source1 然
    后跳转去处理消息。
42              if (__Source0DidDispatchPortLastTime) {
43                  Boolean hasMsg = __CFRunLoopServiceMachPort(dispatchPort, &msg)
44                  if (hasMsg) goto handle_msg;
45              }
```

```
46
47              /// 通知 Observers: RunLoop 的线程即将进入休眠(sleep)。
48              if (!sourceHandledThisLoop) {
49                  __CFRunLoopDoObservers(runloop, currentMode,
    kCFRunLoopBeforeWaiting);
50              }
51
52              /// 7. 调用 mach_msg 等待接受 mach_port 的消息。线程将进入休眠, 直到被下面某
    一个事件唤醒。
53              /// • 一个基于 port 的Source 的事件。
54              /// • 一个 Timer 到时间了
55              /// • RunLoop 自身的超时时间到了
56              /// • 被其他什么调用者手动唤醒
57              __CFRunLoopServiceMachPort(waitSet, &msg, sizeof(msg_buffer),
    &livePort) {
58                  mach_msg(msg, MACH_RCV_MSG, port); // thread wait for receive msg
59              }
60
61              /// 8. 通知 Observers: RunLoop 的线程刚刚被唤醒了。
62              __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopAfterWaiting);
63
64              /// 收到消息，处理消息。
65              handle_msg:
66
67              /// 9.1 如果一个 Timer 到时间了，触发这个Timer的回调。
68              if (msg_is_timer) {
69                  __CFRunLoopDoTimers(runloop, currentMode, mach_absolute_time())
70              }
71
72              /// 9.2 如果有dispatch到main_queue的block，执行block。
73              else if (msg_is_dispatch) {
74                  __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__(msg);
75              }
76
77              /// 9.3 如果一个 Source1 (基于port) 发出事件了，处理这个事件
```

```
78              else {
79                  CFRunLoopSourceRef source1 = __CFRunLoopModeFindSourceForMachPort(runloop, currentMode, livePort);
80                  sourceHandledThisLoop = __CFRunLoopDoSource1(runloop, currentMode, source1, msg);
81                  if (sourceHandledThisLoop) {
82                      mach_msg(reply, MACH_SEND_MSG, reply);
83                  }
84              }
85
86              /// 执行加入到Loop的block
87              __CFRunLoopDoBlocks(runloop, currentMode);
88
89
90              if (sourceHandledThisLoop && stopAfterHandle) {
91                  /// 进入loop时参数说处理完事件就返回。
92                  retVal = kCFRunLoopRunHandledSource;
93              } else if (timeout) {
94                  /// 超出传入参数标记的超时时间了
95                  retVal = kCFRunLoopRunTimedOut;
96              } else if (__CFRunLoopIsStopped(runloop)) {
97                  /// 被外部调用者强制停止了
98                  retVal = kCFRunLoopRunStopped;
99              } else if (__CFRunLoopModeIsEmpty(runloop, currentMode)) {
100                 /// source/timer/observer一个都没有了
101                 retVal = kCFRunLoopRunFinished;
102             }
103
104             /// 如果没超时，mode里没空，loop也没被停止，那继续loop。
105         } while (retVal == 0);
106     }
107
108     /// 10. 通知 Observers: RunLoop 即将退出。
109     __CFRunLoopDoObservers(rl, currentMode, kCFRunLoopExit);
110 }
```

- RunLoop调用mach_msg接收消息，如果没有人port消息，内核会将线程置为等待状态。
- 这样的机制: 用户态程序切换到内核态, 但是不能控制在内核态中执行的指令。这种机制叫**系统调用**, 在CPU中的实现称之为**陷阱指令**(Trap Instruction)
- 为了实现消息的发送和接收，mach_msg() 函数实际上是调用了一个 Mach 陷阱 (trap)，即函数 mach_msg_trap()，陷阱这个概念在 Mach 中等同于系统调用。当你在用户态调用 mach_msg_trap() 时会触发陷阱机制，切换到内核态；内核态中内核实现的 mach_msg() 函数会完成实际的工作，
- RunLoop 的核心就是一个 mach_msg() (见上面代码的第7步)，RunLoop 调用这个函数去接收消息，如果没有别人发送 port 消息过来，内核会将线程置于等待状态。例如你在模拟器里跑起一个 iOS 的 App，然后在 App 静止时点击暂停，你会看到主线程调用栈是停留在 mach_msg_trap() 这个地方。

- **调用过程：**
  - 首先是通知Observer
    - Entry，然后进入循环
    - BeforeTimer
    - BeforeSources
  - 处理Source0
  - 如果有Souce1跳去处理下两步；如果没有，BeforeWaiting，线程休眠
  - AfterWaiting，线程苏醒；
  - 处理事件，按照先后顺序处理，只处理一件
    - Timer
    - Dispatch
    - source1
  - 判断是否停止runloop，否则重新开始循环，从BeforeTimer开始
    - 参数说处理完事件就退出；
    - 当前mode没有source、timer和observer；
    - 外部强行停止；
    - 时间超时；
- source0：performSelector；source1：系统事件，如触摸屏幕；

多线程

基本概念

- 进程：进程是系统正在运行的程序，是
- 线程：可以理解为轻量级的进程
- 线程和进程的比较：

- 进程是系统资源分配的基本单位
- 线程是CPU调度和分配的基本单位。
- 一个进程至少有一个线程
- 同一个进程的线程共享进程的资源

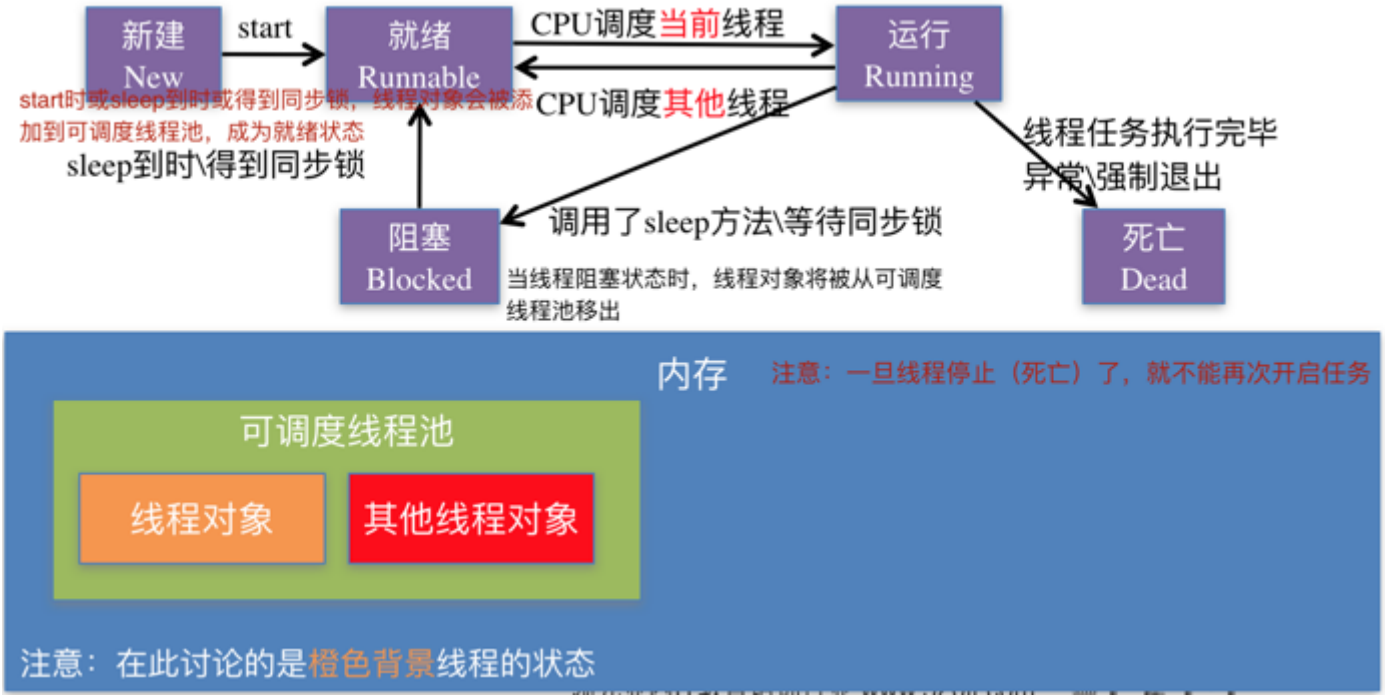
- 同步与异步
  - 同步：方法一旦开始，必须执行完毕才能往下执行；
  - 异步：方法开始，可以接着往下执行；
- 串行与并行
  - 串行：多个任务按照顺序执行，完成一个后再完成下一个；
  - 并行：多个任务同时执行，异步是并行的前提条件。
- 并行和并发：
  - 并行：多个任务真的同一时刻发生；
  - 并发：宏观上多个任务同时执行，实际上是交替执行，因为速度很快，导致看起来是同时执行。
- iOS线程技术

技术方案	简介	语言	线程生命周期	使用频率
pthread	▪ 一套通用的多线程API ▪ 适用于Unix\Linux\Windows等系统 ▪ 跨平台\可移植 ▪ 使用难度大	C	程序员管理	几乎不用
NSThread	▪ 使用更加面向对象 ▪ 简单易用，可直接操作线程对象	OC	程序员管理	偶尔使用
GCD	▪ 旨在替代NSThread等线程技术 ▪ 充分利用设备的多核	C	自动管理	经常使用
NSOperation	▪ 基于GCD（底层是GCD） ▪ 比GCD多了一些更简单实用的功能 ▪ 使用更加面向对象	OC	自动管理	经常使用

NSThread&pthread

```
1     //1.pthread
2 //    pthread_t thead;
3 //    //第二个是线程属性,te是执行的函数，16是传入的参数
4 //    pthread_create(&thead, NULL, te, 16);
```

```
5
6       //2.NSThread
7       //(1)创建线程，需要手动开启
8   //    NSThread *thread = [[NSThread alloc] initWithTarget:self
    selector:@selector(test:) object:@5];
9   //    [thread start];
10      //(2)创建线程，自动开启
11  //    [NSThread detachNewThreadSelector:@selector(test:) toTarget:self
    withObject:@66];
12      //(3)隐式创建线程，自动开启
```



GCD(Grand Central Dispatch)

- 是纯C语言
- 苹果公司为多核的并行运算提出的解决方案
- GCD会自动利用更多的CPU内核
- GCD会自动管理线程的生命周期（创建线程、调度任务、销毁线程）

- GCD有两个核心的概念
  - **任务**：执行什么操作，任务有两种执行方式
    - **同步函数**：阻塞当前函数
    - **异步函数**：不阻塞当前函数；
  - **队列**：用来存放任务，分为串行和并行队列。
    - **串行队列**（Serial Dispatch Queue）：让任务一个接一个执行。

- **并行队**列（Concurrent Dispatch Queue）：任务并发执行，并发功能只有在异步函数下才有用。

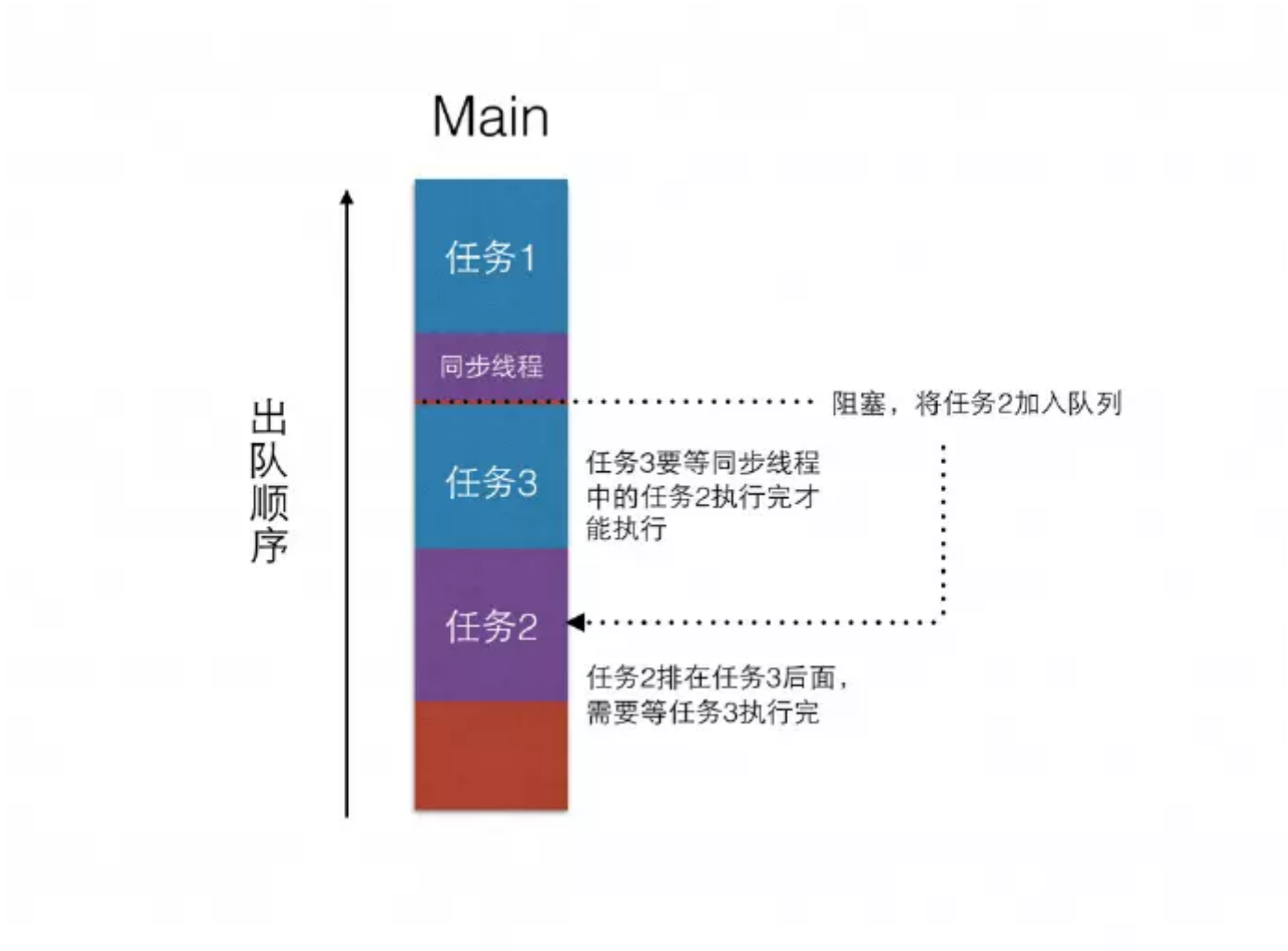
- 创建队列

```
// 第一个参数const char *label : C语言字符串，用来标识
// 第二个参数dispatch_queue_attr_t attr : 队列的类型
// 并发队列:DISPATCH_QUEUE_CONCURRENT// 串行队列:DISPATCH_QUEUE_SERIAL 或者 NULL
dispatch_queue_t queue = dispatch_queue_create(const char *label, dispatch_queue_attr_t attr);

//默认全局并发队列
/**
     第一个参数:优先级 也可直接填后面的数字
     #define DISPATCH_QUEUE_PRIORITY_HIGH 2 // 高
     #define DISPATCH_QUEUE_PRIORITY_DEFAULT 0 // 默认
     #define DISPATCH_QUEUE_PRIORITY_LOW (-2) // 低
     #define DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN // 后台
     第二个参数: 预留参数  0
     */
    dispatch_queue_t ququel = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

//默认主队列
dispatch_queue_t  queue = dispatch_get_main_queue();
```

- 任务的执行，任务在block内

```
//同步函数
dispatch_sync(queue, ^{

    });
//异步函数
dispatch_async(queue, ^{

    });
```

- 函数和队列搭配
  - 同步函数-并行队列：当前线程串行执行；
  - 同步函数-串行队列：当前线程串行执行
  - 同步函数-主队列：死锁，加入任务2，阻塞当前线程，直到任务2完成。任务2无法完成，因为当前线程被阻塞。
  - 异步函数-并行队列：开启多个新的线程，并发执行
  - 异步函数-串行队列：开启一个新线程，串行执行
  - 异步函数-主队列：主线程，串行执行



- 栅栏函数

栅栏函数前面的函数执行完了后执行栅栏函数，然后执行后面的函数。

- 延迟执行
  - dispathc_after
  - performSelector
  - scheduledTimerWithTimeInrerval
- 一次函数

dispatch_once

- 迭代函数

dispatch_apply，不能加入主队列

- 队列组（同栅栏函数）

```
    // 创建队列组
    dispatch_group_t group = dispatch_group_create();
    // 创建并行队列
    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
    // 执行队列组任务
    dispatch_group_async(group, queue, ^{
    });
    //队列组中的任务执行完毕之后，执行该函数
    dispatch_group_notify(group, queue, ^{
    });
```

- 信号量

NSOperation

NSOperation是抽象类，要使用它的子类。NSOperation和NSOperationQueue分别对应GCD的任务和队列

NSOperation的创建

NSInvocationOperaion

```
 /*
    第一个参数:目标对象
    第二个参数:选择器,要调用的方法
    第三个参数:方法要传递的参数
    */
//主线程
NSInvocationOperation *op  = [[NSInvocationOperation alloc]initWithTarget:self selector:@selector(download) object:nil];//启动操作
[op start];
```

NSBlockOperation(最常用)

```
//1.封装操作
NSBlockOperation *op = [NSBlockOperation blockOperationWithBlock:^{
        //要执行的操作，在主线程中执行
        NSLog(@"1------%@",[NSThread currentThread]);
}];
//2.追加操作，追加的操作在子线程中执行，可以追加多条操作
[op addExecutionBlock:^{
         NSLog(@"---download2--%@",[NSThread currentThread]);
    }];
[op start];
```

NSOperationQueue

两种队列

- 主队列：通过mainQueue获得，该队列的任务都将在主线程执行
- 非主队列：直接alloc init出来的队列。非主队列同时具备了并发和串行的功能，通过设置最大并发数属性来控制任务是并发执行还是串行执行。

NSOperationQueue的作用

- NSOperation可以直接start执行任务，不需要queue，但默认是同步的。

将任务添加到队列

```
- (void)addOperation:(NSOperation *)op;
- (void)addOperationWithBlock:(void (^)(void))block;
//把任务添加到队列就会自动开始，内部会自动调用start。
```

重要属性和方法

NSOperation

- 依赖

```
// 操作op1依赖op5，即op1必须等op5执行完毕之后才会执行
```

```
// 添加操作依赖,注意不能循环依赖，如果循环依赖会造成两个任务都不会执行// 也可以夸队列依赖，依赖别的队列的操作
    [op1 addDependency:op5];
```

- 监听

```
// 监听操作的完成
// 当op1线程完成之后，立刻就会执行block块中的代码
// block中的代码与op1不一定在一个线程中执行，但是一定在子线程中执行
op1.completionBlock = ^{
        NSLog(@"op1已经完成了---%@",[NSThread currentThread]);
    };
```

NSOperationQueue

- maxConcurrentOperationCount

```
    //1.创建队列
    NSOperationQueue *queue = [[NSOperationQueue alloc]init];
    /*
     默认是并发队列,如果最大并发数>1,并发
     如果最大并发数==1,串行队列
     系统的默认是最大并发数-1 ,表示不限制
     设置成0则不会执行任何操作
     */
     queue.maxConcurrentOperationCount = 1;
```

AutoReleasePool

含义

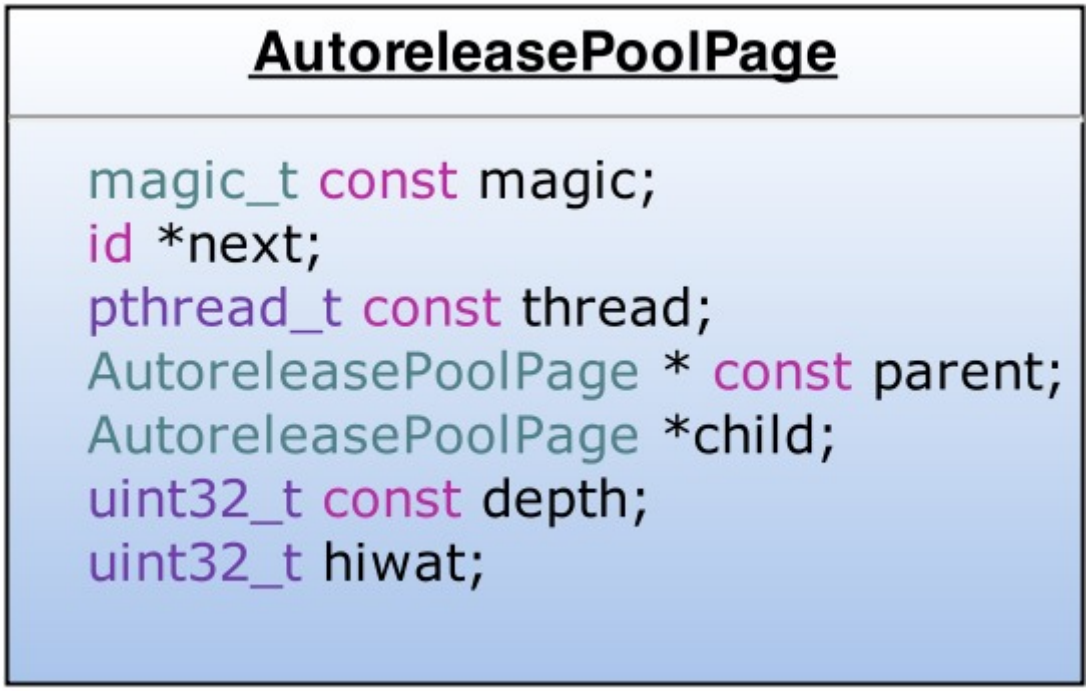
AutoReleasePool是OC中的一种内存自动回收机制，它可以延迟加入AutoReleasePool中的变量的release时机。正常情况下，变量会在超过其作用域时release，但是如果加入AutoReleasePool，release会延迟。

使用场景

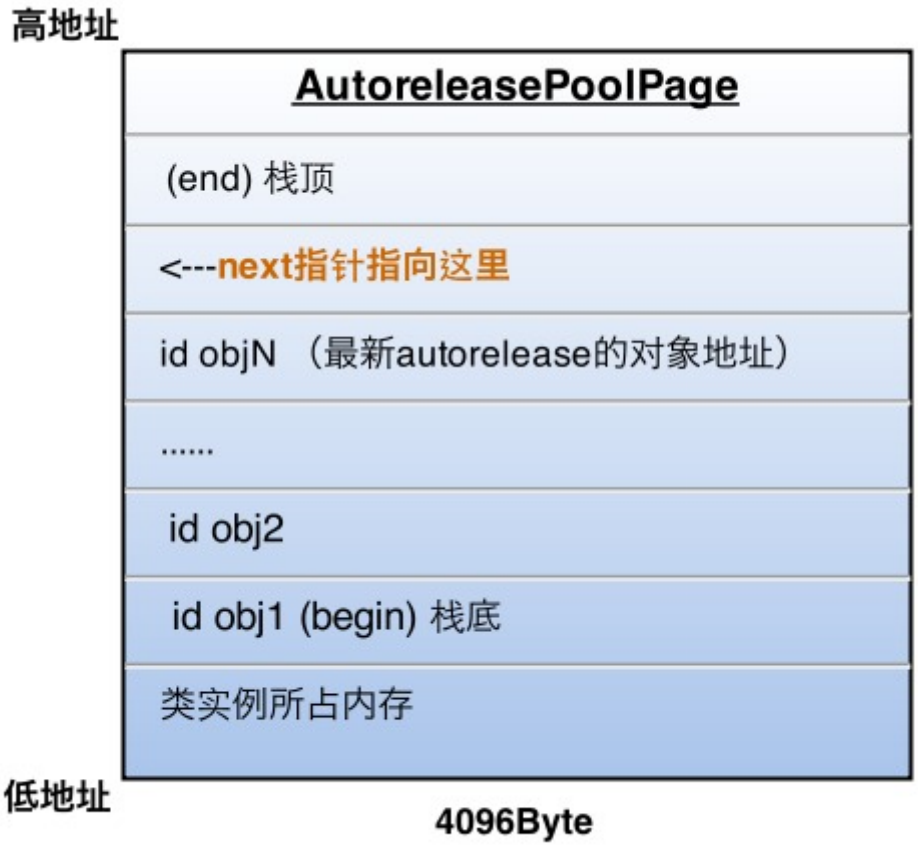
- autorelease机制基于UI FrameWork。因此写非UI FrameWork程序时，需要自己管理对象生成周期。
- autorelease 触发时机发生在下一次runloop的时候。因此如何在一个大的循环里不断创建autorelease对象，那么这些对象在下一次runloop回来之前将没有机会被释放，可能会耗尽内存。这种情况下，可以在循环内部显式使用@autoreleasepool {}将autorelease 对象释放。这种情况我们会比较经常遇到，具体代码如下：

结构

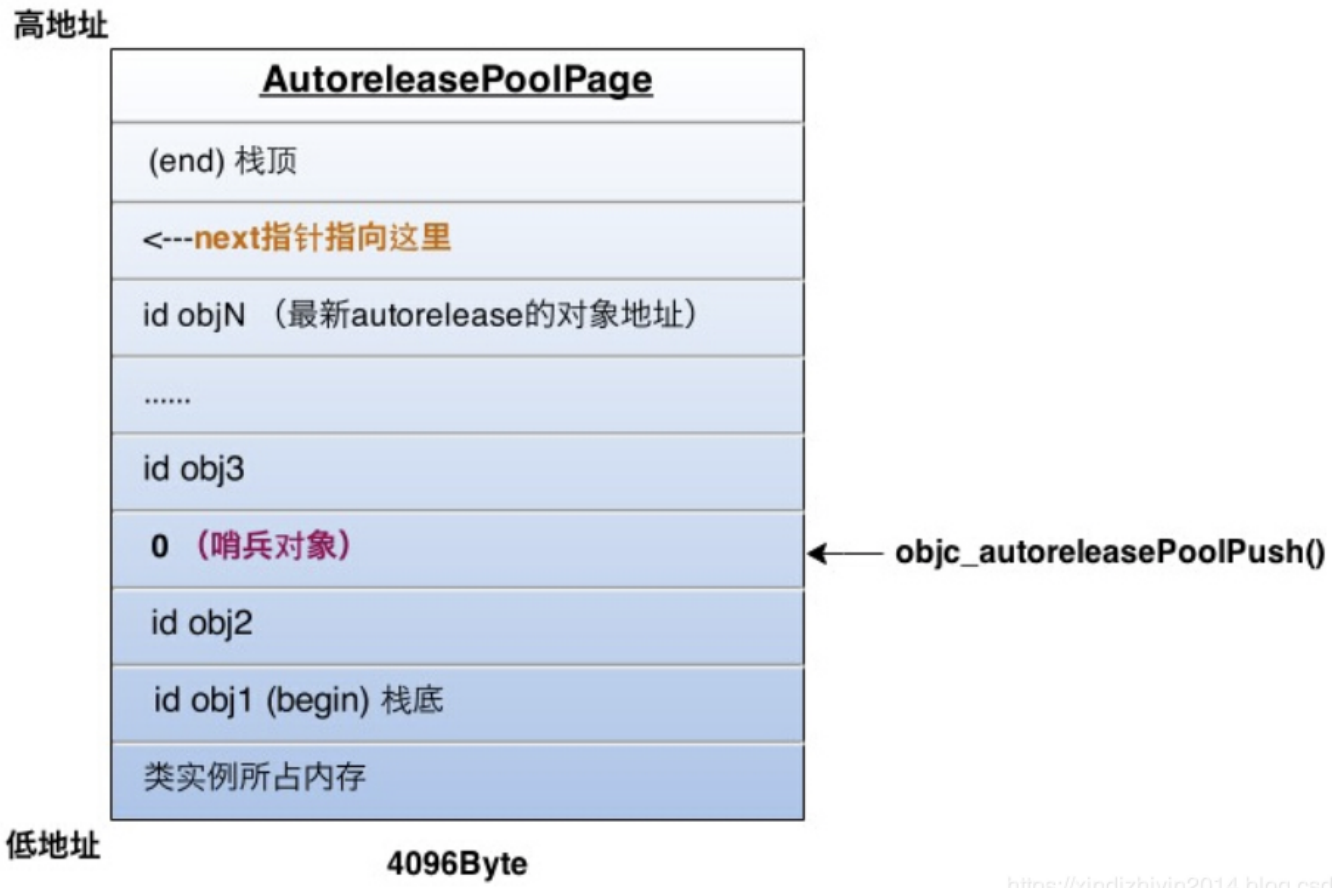
AutoReleasePool没有单独的结构，是由一个个AutoReleasePoolPage以双向链表组合而成。



- `AutoreleasePoolPage`每个对象会开辟4096字节内存（也就是虚拟内存一页的大小），除了上面的实例变量所占空间，剩下的空间全部用来储存`autorelease`对象的地址
- 上面的`id *next`指针作为游标指向栈顶最新add进来的`autorelease`对象的下一个位置
- 一个`AutoreleasePoolPage`的空间被占满时，会新建一个`AutoreleasePoolPage`对象，连接链表，后来的`autorelease`对象在新的page加入
- 下图是加入了autorelease对象，所以，向一个对象发送`- autorelease`消息，就是将这个对象加入到当前`AutoreleasePoolPage`的栈顶`next`指针指向的位置



- `objc_autoreleasePoolPush`返回哨兵对象的地址
- 每进行一次 `objc_autoreleasePoolPush` ，就会向当前AutoReleasePage加入一个哨兵对象，值为0(也就是nil)，一个page就会变成下面：



- `objc_autoreleasePoolPop`把push返回的哨兵对象地址作为参数
  - 根据哨兵对象找到哨兵对象所处的page；
  - 在当前page，往所有晚于哨兵对象加入（上面）的对象都发送release消息，并将next指针移到哨兵对象的位置。

源码解析

```
1 //未编译
2 int main(int argc, const char * argv[]) {
3     @autoreleasepool {
4         NSString *str =[NSString new];
5     }
6     return 0;
7 }
8
9 //编译后。超出作用范围自动调用析构函数，没有看到autoRelease调用，可能是我方法没用对
```

```
int main(int argc, const char * argv[]) {
    /* @autoreleasepool */ { __AtAutoreleasePool __autoreleasepool;
        NSString *str =((NSString *(*)(id, SEL))(void *)objc_msgSend)((id)objc_getClass("NSString"), sel_registerName("new"));
    }
    return 0;
}

//__AtAutoreleasePool结构体
struct __AtAutoreleasePool {
  __AtAutoreleasePool() {atautoreleasepoolobj = objc_autoreleasePoolPush();}//构造函数
  ~__AtAutoreleasePool() {objc_autoreleasePoolPop(atautoreleasepoolobj);}//析构函数
  void * atautoreleasepoolobj;};
```

Push & Pop & autorelease

hotPage()

- hotPage()：获得hotPage，hotPage存在一个字典里，key为常量

```
static inline AutoreleasePoolPage *hotPage()
    {
        AutoreleasePoolPage *result = (AutoreleasePoolPage *)
            tls_get_direct(key);//`hotPage()`个人认为是使用key-value的形式获取到当前的AutoreleasePoolPage。
        if ((id *)result == EMPTY_POOL_PLACEHOLDER) return nil;
        if (result) result->fastcheck();
        return result;
    }
```

autoreleaseFullPage()

- autoreleaseFullPage：hotPage满了时调用的方法

- 往下找到hotPage的子page，直到该子page未满，跳到第三步；
- 如果最后一个子page也满了，创建一个新的子page，跳到第三步；
- 把该page设为hotPage，往该hotPage添加对象（可能是自动释放的对象，也可能是哨兵）。

```
id *autoreleaseFullPage(id obj, AutoreleasePoolPage *page)
    {
        // The hot page is full.
        // Step to the next non-full page, adding a new page if necessary.
        // Then add the object to that page.
        assert(page == hotPage());
        assert(page->full()  ||  DebugPoolAllocation);

        do {
            if (page->child) page = page->child;
            else page = new AutoreleasePoolPage(page);
        } while (page->full());

        setHotPage(page);
        return page->add(obj);
    }
```

autoreleaseNoPage()

- autoreleaseNoPage，hotPage为空时调用，创建一个新的page插入对象。如果haveEmptyPoolPlaceholder()有一个空的page占位，还要先插入一个哨兵对象，然后再插入释放对象。

```
id *autoreleaseNoPage(id obj)
      {
        assert(!hotPage());
        bool pushExtraBoundary = false;
        if (haveEmptyPoolPlaceholder()) {
            pushExtraBoundary = true;
        }
        else if (obj != POOL_BOUNDARY  &&  DebugMissingPools) {
            _objc_inform("MISSING POOLS: (%p) Object %p of class %s "
```

```
                          "autoreleased with no pool in place - "
                          "just leaking - break on "
                          "objc_autoreleaseNoPool() to debug",
                          pthread_self(), (void*)obj, object_getClassName(obj));
            objc_autoreleaseNoPool(obj);
            return nil;
        }
        else if (obj == POOL_BOUNDARY  &&  !DebugPoolAllocation) {
            return setEmptyPoolPlaceholder();
        }
        AutoreleasePoolPage *page = new AutoreleasePoolPage(nil);
        setHotPage(page);
        if (pushExtraBoundary) {
            page->add(POOL_BOUNDARY);
        }
        return page->add(obj);
    }
```

```
#   define EMPTY_POOL_PLACEHOLDER ((id*)1)

static inline bool haveEmptyPoolPlaceholder()
{
    id *tls = (id *)tls_get_direct(key);
    return (tls == EMPTY_POOL_PLACEHOLDER);
}
```

page->add()

- page->add()

```
id *add(id obj)
    {
        assert(!full());//保证page未满
        unprotect();
        id *ret = next;  // 获得栈顶位置地址
```

```
        *next++ = obj;////  往栈顶加入释放对象，next指针往上移动
        protect();
        return ret;//返回之前栈顶位置指针，也就是释放对象的指针。
    }
```

autoreleaseFast()

- 首先获得hotPage；
- hotPage存在且未满就直接插入释放对象；
- hotPage存在但是满了就调用autoreleaseFullPage()；
- hotPage不存在就调用autoreleaseNoPage();

```
static inline id *autoreleaseFast(id obj)
    {
        AutoreleasePoolPage *page = hotPage();
        if (page && !page->full()) {
            return page->add(obj);
        } else if (page) {
            return autoreleaseFullPage(obj, page);
        } else {
            return autoreleaseNoPage(obj);
        }
    }
```

autoReleaseNewPage()

- 调试模式下调用，每个释放对象都会单独创建一个page

```
id *autoreleaseNewPage(id obj)
    {
        AutoreleasePoolPage *page = hotPage();
        if (page)
        return autoreleaseFullPage(obj, page);
        else
        return autoreleaseNoPage(obj);
```

```
    }
```

Push(objc_autoreleasePoolPush)

```
void *objc_autoreleasePoolPush(void){
    return AutoreleasePoolPage::push();
}
```

- `__AtAutoreleasePool`的`objc_autoreleasePoolPush`本质上还是调用了`AutoreleasePoolPage`的`push`方法。

```
static inline void *push()
    {
        id *dest;
        if (DebugPoolAllocation) {
            // Each autorelease pool starts on a new pool page.
            /**  序号1  **/
            dest = autoreleaseNewPage(POOL_BOUNDARY);
        } else {
            /**  序号2  **/
            dest = autoreleaseFast(POOL_BOUNDARY);
        }
        assert(dest == EMPTY_POOL_PLACEHOLDER || *dest == POOL_BOUNDARY);
        return dest;
    }

```

- 调试模式就调用`autoreleaseNewPage`，为每一个释放对象单独创建一个page，否则就正常插入。

autorelease

autorelease()

```
// Replaced by ObjectAlloc
- (id)autorelease {
```

```
    return ((id)self)->rootAutorelease();
}

// Base autorelease implementation, ignoring overrides.
inline id
objc_object::rootAutorelease()
{
    if (isTaggedPointer()) return (id)this; // 如果是 tagged pointer，直接返回 this
    if (prepareOptimizedReturn(ReturnAtPlus1)) return (id)this; // 如果 prepareOptimizedReturn(ReturnAtPlus1) 返回 true，直接返回 this

    return rootAutorelease2(); // 调用 rootAutorelease2
}
```

- 如果一个指针是tagged pointer，就直接返回指针；
- 如果能够优化，就直接返回指针；
- 否则调用rootAutorelease2();

rootAutorelease2()

```
objc_object::rootAutorelease2()
{
    assert(!isTaggedPointer());
    return AutoreleasePoolPage::autorelease((id)this);
}

```

page-> autorelease()

```
static inline id autorelease(id obj){
    assert(obj); // 对象不可为 NULL
    assert(!obj->isTaggedPointer()); // 对象不可为 tagged pointer
    id *dest __unused = autoreleaseFast(obj);
    assert(!dest  ||  dest == EMPTY_POOL_PLACEHOLDER  ||  *dest == obj); // 确保 dest 不存在或等于 EMPTY_POOL_PLACEHOLDER 或等于 obj
    return obj;
```

```
}

```

autorelease返回值优化策略

> prepareOptimizedReturn() 函数中主要通过 callerAcceptsOptimizedReturn() 函数来判断是否需要或者是否可以进行优化, 该函数的参数 __builtin_return_address(0)) 代表当前函数地址. 这个函数涉及更底层的东西, 目前还没有去接触.
> 但是通过对该函数的注释, 大致得出了这个函数的实现就是通过一系列对帧指针寄存器的判断来判断该地址指向的函数是否可进行优化。
> 优化的方案叫做 Tread Local Storage, TLS, 线程本地存储, 是线程对应的一块地址空间, 对返回值的优化就是将返回值存入到该段空间中, 避免了加入到 autoreleasepool 中的一系列操作.
>
> 在返回值身上调用`objc_autoreleaseReturnValue`方法时，runtime将这个返回值object储存在TLS中，然后直接返回这个object（不调用autorelease）；同时，在外部接收这个返回值的`objc_retainAutoreleasedReturnValue`里，发现TLS中正好存了这个对象，那么直接返回这个object（不调用retain）。

autorelease返回值优化策略一般是用于把对象作为返回值的情况。

```
+ (instancetype)createSark {
    return [self new];
}
callerSark *sark = [Sark createSark];

//编译后
+ (instancetype)createSark {
    id tmp = [self new];
    return objc_autoreleaseReturnValue(tmp); // 代替我们调用autorelease
}
// caller
id tmp = objc_retainAutoreleasedReturnValue([Sark createSark]) // 代替我们调用retain
Sark *sark = tmp;
objc_storeStrong(&sark, nil); // 相当于代替我们调用了release
```

> 在返回值身上调用`objc_autoreleaseReturnValue`方法时，runtime将这个返回值object储存在TLS中，然后直接返回这个object（不调用autorelease）；同时，在外部接收这个返回值的

> `objc_retainAutoreleasedReturnValue`里，发现TLS中正好存了这个对象，那么直接返回这个object（不调用retain）。

但是如果被调方和调用方如果只有一边是ARC环境，就会出现`createSark`里调用了`objc_autoreleaseReturnValue`但是外面没有`objc_retainAutoreleasedReturnValue`的情况。所以我们需要下面的黑魔法：

__builtin_return_address

这个内建函数原型是`char *__builtin_return_address(int level)`，作用是得到函数的返回地址，参数表示层数，如__builtin_return_address(0)表示当前函数体返回地址，传1是调用这个函数的外层函数的返回值地址，以此类推。
这样就可以定位到`[Sark createSark]`下一行指令是不是`objc_retainAutoreleasedReturnValue`，如果是就可以执行优化策略，否则就执行没被优化的逻辑：加入到自动释放池。

Pop(objc_autoreleasePoolPop)

```
1  static inline void pop(void *token)
2      {
3          AutoreleasePoolPage *page;
4          id *stop;
5          if (token == (void*)EMPTY_POOL_PLACEHOLDER) {
6              checkToken();
7              return;
8          }
9          page = pageForPointer(token);
10         stop = (id *)token;
11         checkStop(stop, page, token);
12         if (PrintPoolHiwat) printHiwat();
13         page->releaseUntil(stop);
14         killPage(page);
15     }
16
```

page->releaseUntil()

- 释放page里的对象

```
void releaseUntil(id *stop)
    {
        // Not recursive: we don't want to blow out the stack
        // if a thread accumulates a stupendous amount of garbage
        while (this->next != stop) {
            // Restart from hotPage() every time, in case -release
            // autoreleased more objects
            AutoreleasePoolPage *page = hotPage();
            // fixme I think this `while` can be `if`, but I can't prove it
            while (page->empty()) {
                page = page->parent;
                setHotPage(page);
            }
            page->unprotect();
            id obj = *--page->next;
            memset((void*)page->next, SCRIBBLE, sizeof(*page->next));
            page->protect();

            if (obj != POOL_BOUNDARY) {
                objc_release(obj);
            }
        }
        setHotPage(this);
#if DEBUG
        // we expect any children to be completely empty
        for (AutoreleasePoolPage *page = child; page; page = page->child) {
            assert(page->empty());
        }#endif
    }

```

- 从hotPage的顶部往下release对象；
- 如果hotPage被释放空了，把parent设为hotPage，重新开始上一步。如果遇到传入的参数哨兵对象，退出循环。

- 最后也要保证当前hotPage的所有子page都要为空。
- 该循环只是把page里的释放对象都删除完，但会剩下很多空的page。

page->kill

- 把哨兵对象的page的所有child置为空。

```
void kill()
    {
        // Not recursive: we don't want to blow out the stack
        // if a thread accumulates a stupendous amount of garbage
        AutoreleasePoolPage *page = this;
        while (page->child) page = page->child;

        AutoreleasePoolPage *deathptr;
        do {
            deathptr = page;
            page = page->parent;
            if (page) {
                page->unprotect();
                page->child = nil;
                page->protect();
            }
            delete deathptr;
        } while (deathptr != this);
    }

```

- 从哨兵对象page开始，遍历到最底层的child。
- 往前释放page，直到遇到哨兵对象page。

NSCache

- 类似字典的缓存方式；
- 自动释放引用，可以设置countLimit，超过数量添加时，移除最早添加的数据。

- 线程安全；

沙盒

- 概念
  - 每个文件都有自己的沙盒，沙盒就是文件目录；
  - 沙盒不能随意访问，必须有权限；
- 组成
  - Documents
    - 保存运行时需要持久化的数据，iTunes会自动备份该目录；
  - Library
    - Cache
      - 存储缓存文件，如视频照片，不会在应用退出时删除；
    - Preference
      - 保存应用程序的偏好设置，不应该直接创建文件，需要通过NSUserDefault来访问应用程序的偏好设置。会自动备份。
  - tmp
    - 临时文件目录，程序重新运行时和开机时会清空。

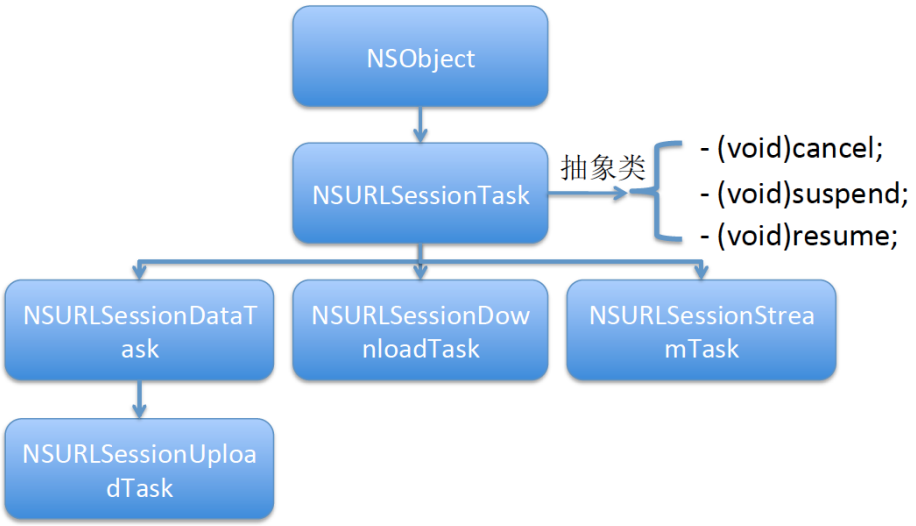
NSURLSession

- `AFNetworking`就是基于`NSURLSession`的，所以必须先了解`NSURLSession`；
- 主要类：
  - `NSURLSession`：
    - 全局session：单例全局会话，所有程序都可以使用，不能设置代理，因为delegate是只读的，只能在创建的时候设置；
    - 自定义session：使用`NSURLSessionConfiguration`生成；

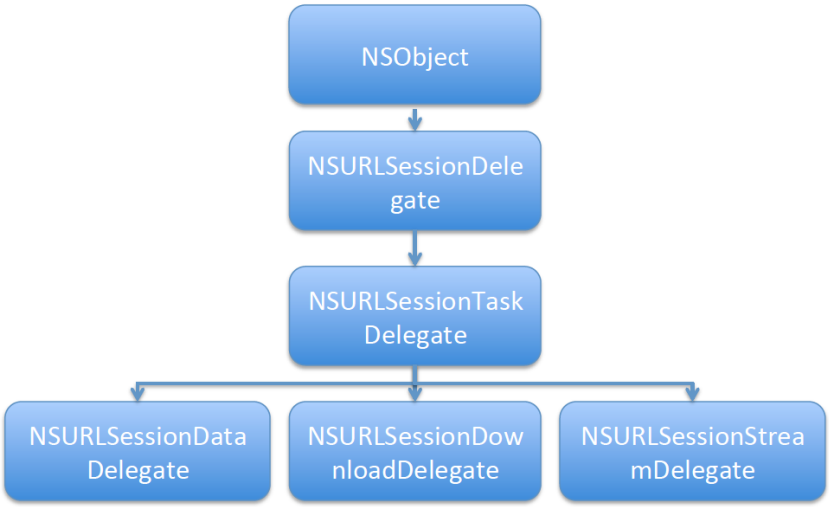
```
1 [NSURLSession sessionWithConfiguration:self.sessionConfiguration delegate:self
  delegateQueue:self.operationQueue];
```

```
//delegate是所有的task公用的，可以让self实现所有delegate
//delegateQueue是代理回调的队列，不是任务执行;必须是串行队列，否则按照顺序返回的数据的回调顺序就无法保证;
```

- `NSURLSessionConfiguration`
  - 分类：
    - `defaultSessionConfiguration`：进程内会话,用硬盘来缓存数据.账户信息存储到钥匙链，如果有cookie会携带cookie；
    - `ephemeralSessionConfiguration`：临时的进程内会话(数据存于内存),不会存cookie,当程序退出数据就会消失；[ɪ'femərəl] 短暂的；
    - `backgroundSessionConfiguration`：后台会话,相比默认会话,任务是交给后台守护线程完成的,属于别的进程非程序本身,来进行网络数据处理;(所以程序崩溃也不会中断下载,但是如果用户使用多界面强制退关闭程序,Session会断开连接.)
  - 功能：
    - 可以统一添加设置请求头信息config.HTTPAdditionalHeaders = @{"Authorization":xxxx}
    - 设置主机的最大连接数 : Config.HTTPMaximumConnectionsPerHost = 5
    - 系统自动选择最佳网络下载:Config.discretionary=YES;
    - 设置请求超时和缓存策略requestCachePolicy/Config.timeoutIntervalForRequest=15;
    - 是否允许蜂窝网络下载Config.allowsCellularAccess=true。
- `NSURLSessionTask`
  - `NSURLSessionDataTask`：请求数据；
  - `NSURLSessionUploadTask`：文件上传；
  - `NSURLSessionDownloadTask`：文件下载；
  - `NSURLSessionStreamTask`：以流的方式进行网络访问；



- NSURLSessionDelegate
  - `NSURLSessionDelegate`：Session级别的委托，负责Session的生命周期回调和身份验证；
  - `NSURLSessionTaskDelegate`：task级别的委托，面向所有的委托方法；
  - `NSURLSessionDataDelegate`：task级别的委托，面向upload和data的委托方法；
  - `NSURLSessionDownloadDelegate`：task级别的委托，面向download的委托方法；
  - `NSURLSessionStreamDelegate`：task级别的委托，面向stream的委托方法；



AFNetworking

- 核心是`AFURLSessionManager`；
  - 负责创建请求的`task`；
  - 有一个自定义NSURLSession，根据传入的`NSURLSessionConfigution`不同
    - 默认是default；
    - 默认委托队列是大小为1的并行队列；
  - `mutableTaskDelegatesKeyedByTaskIdentifier`
    - 为了让task和回调对应；
    - 一个字典，key是task的identifier，value是`AFURLSessionManagerTaskDelegate`；
      - `AFURLSessionManagerTaskDelegate`就是block的封装，就是下面这些回调；
      - 当发生回调，从字典根据task取出delegate，执行block回调

```
- (NSURLSessionDataTask *)GET:(NSString *)URLString
                   parameters:(id)parameters
                     progress:(void (^)(NSProgress * _Nonnull))downloadProgress
```

```
4                       success:(void (^)(NSURLSessionDataTask * _Nonnull, id
_Nullable))success
5                       failure:(void (^)(NSURLSessionDataTask * _Nullable, NSError
* _Nonnull))failure
```