Objective-C语言进阶

session 2

张宇 抖音iOS开发工程师

© 2019 Bytedance Inc. All rights reserved. Redistribution or public display not permitted without written permission from Bytedance.

主要内容

概述

内存管理

Blocks

通信方式

多线程

概述

如果想要开发出流畅、稳定的App，内存管理、Blocks、通信方式、多线程等知识点，

是必须要用到的，而且使用非常频繁，几乎每天都要使用

他们属于进阶技能，有一定难度

但是后面几节课都需要用到，学不好，后面都听不懂

并且这几个知识点是通向合格工程师，甚至高级工程师的必由之路

Objective-C内存管理

session 2

张宇 抖音iOS开发工程师

© 2019 Bytedance Inc. All rights reserved. Redistribution or public display not permitted without written permission from Bytedance.

主要内容

概述

引用计数

手动引用计数（MRC）

自动引用计数（ARC）

概述

为什么需要内存管理

创建对象后开辟内存空间，不释放对象，这种现象称为内存泄漏

物理内存资源有限，内存耗尽，程序崩溃，严重影响用户体验

内存管理即管理对象的创建、释放

内存管理 - 三类方式

显式内存释放（C-free、C++-delete）

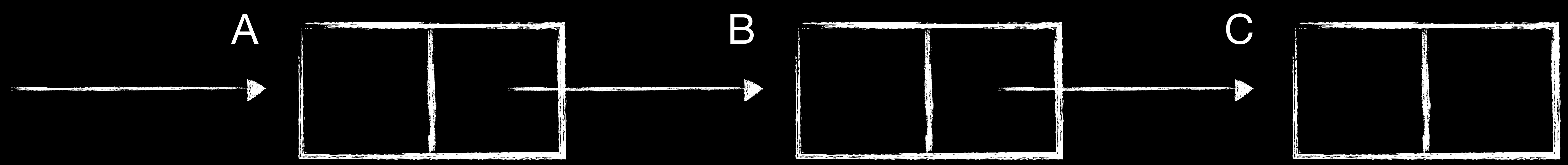
垃圾回收（Java、JavaScript、C#）

基于引用计数（smart pointer、Objective-C）

显式内存释放

内存被提前释放 (悬停指针 dangling pointer)

内存永远无法释放(内存泄露)



显式内存释放问题

```c
struct People{
    int age;
};

struct People* createPeople() {
    struct People *people = malloc(sizeof(struct People));
    people->age = 0;
    return people;
}

void showPeopleAge() {
    struct People *people = createPeople();
    printf("年龄：%i\n", people->age);
    free(people);
}
```

内存分配(malloc)和释放分离

容易遗漏释放逻辑，或重复释放

这种编程方式占用过多的精力

垃圾回收

不需要关心指针管理，对开发者透明

标记、清除算法

假死时间

OS X（macOS）曾经支持垃圾回收, 但是现在已经废弃掉了

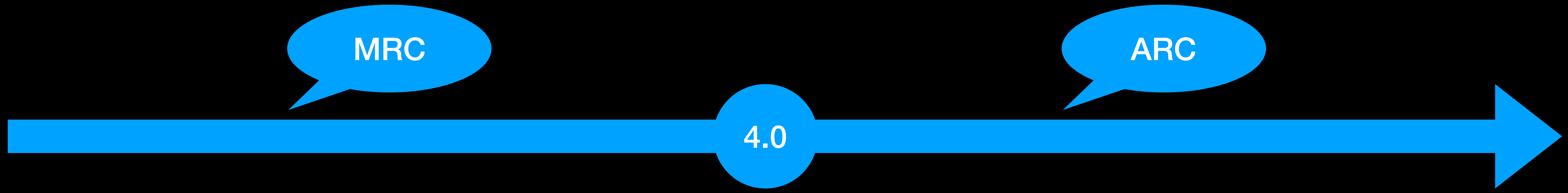
引用计数 - iOS 内存管理

iOS一直不支持垃圾回收

支持两套内存管理：

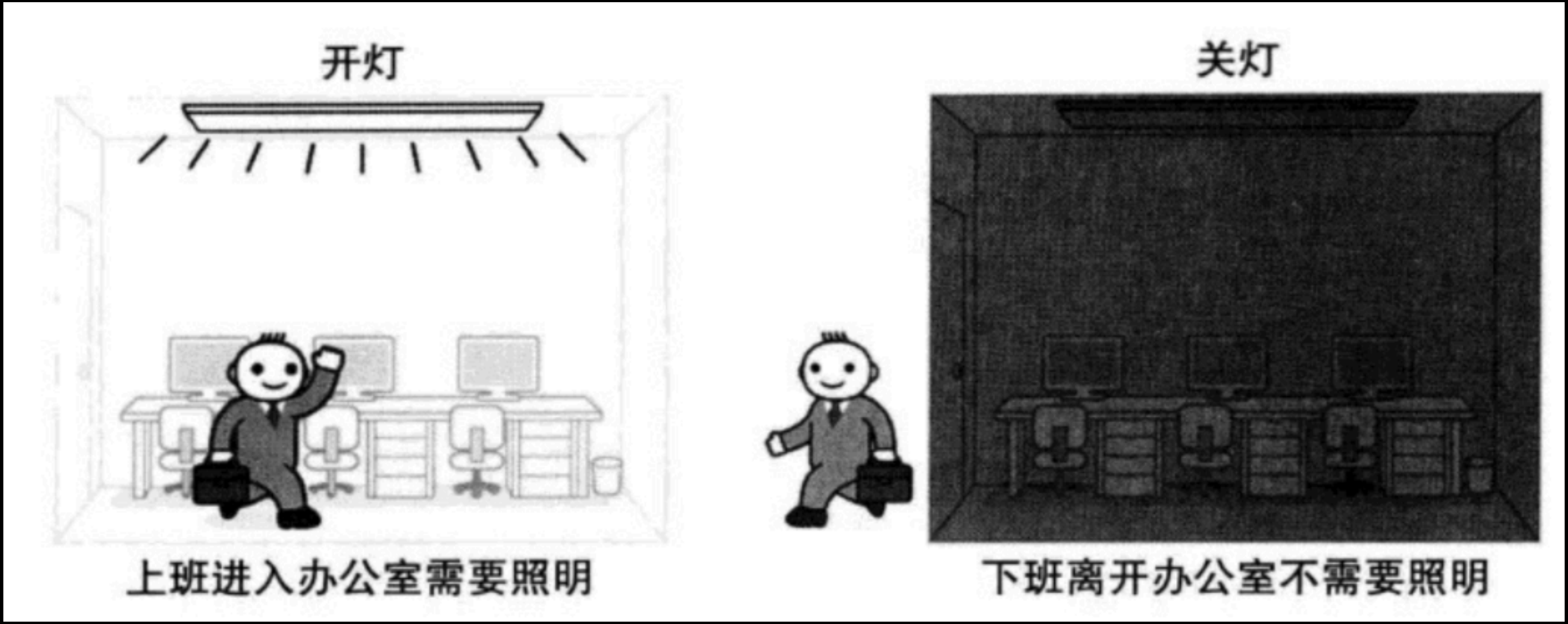
手动引用计数（Mannul Reference Counting）

自动引用计数（Automatic Reference Counting）



引用计数

引用计数 - 开灯、关灯



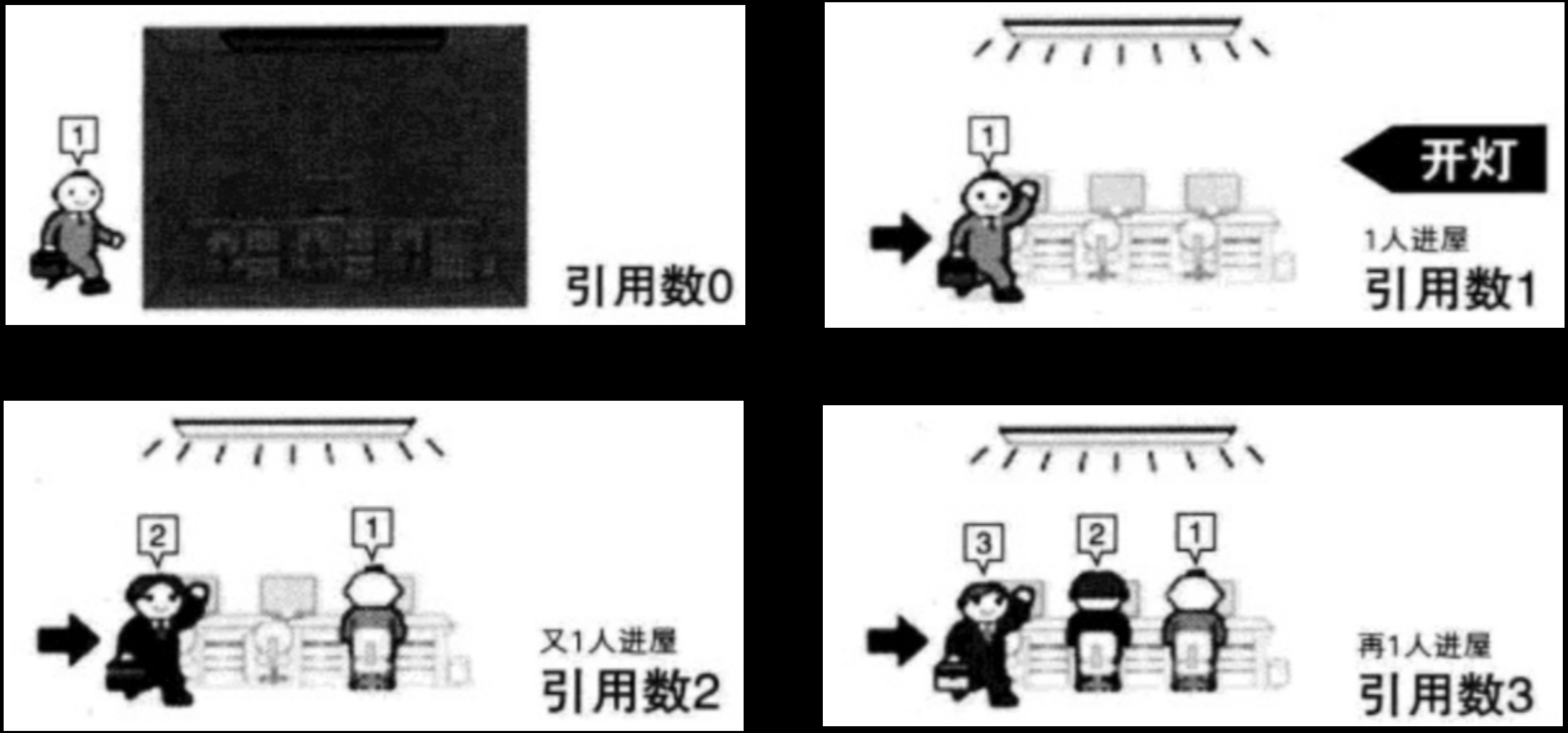
开灯

上班进入办公室需要照明

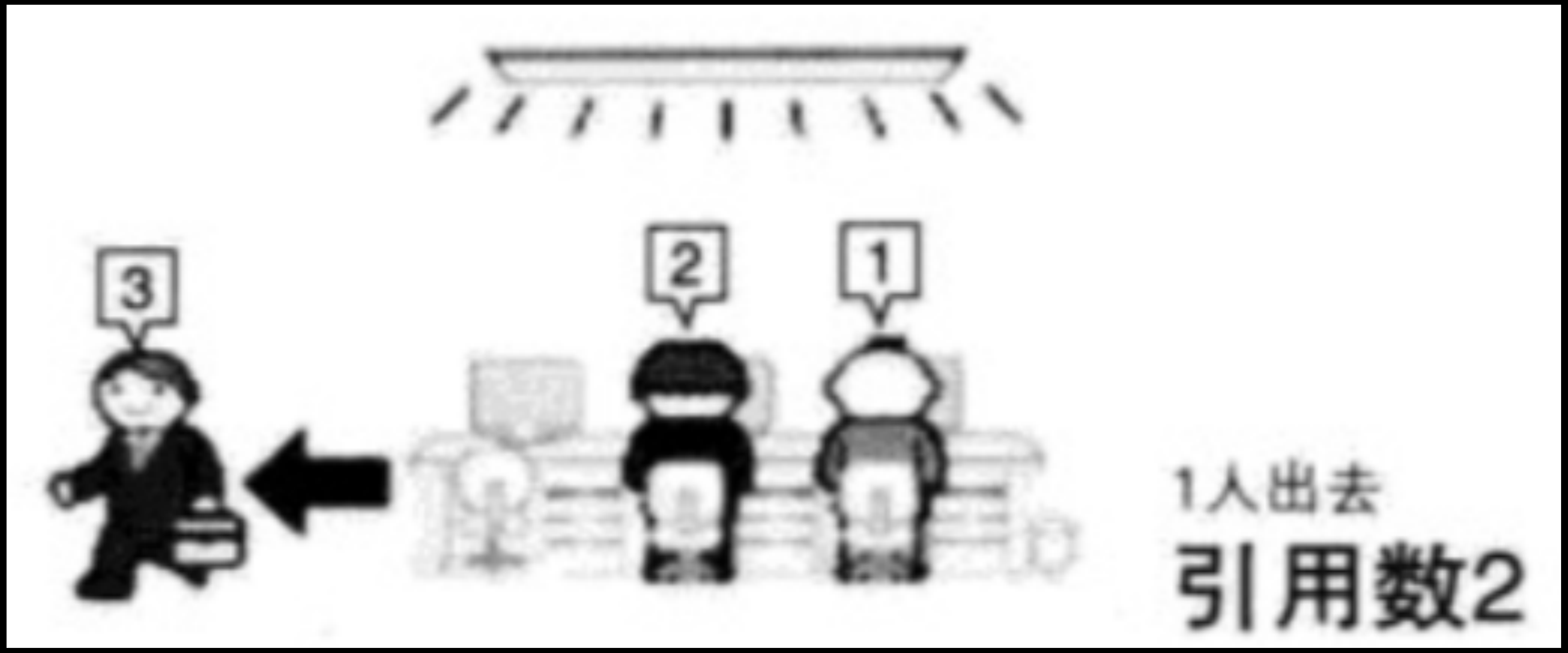
关灯

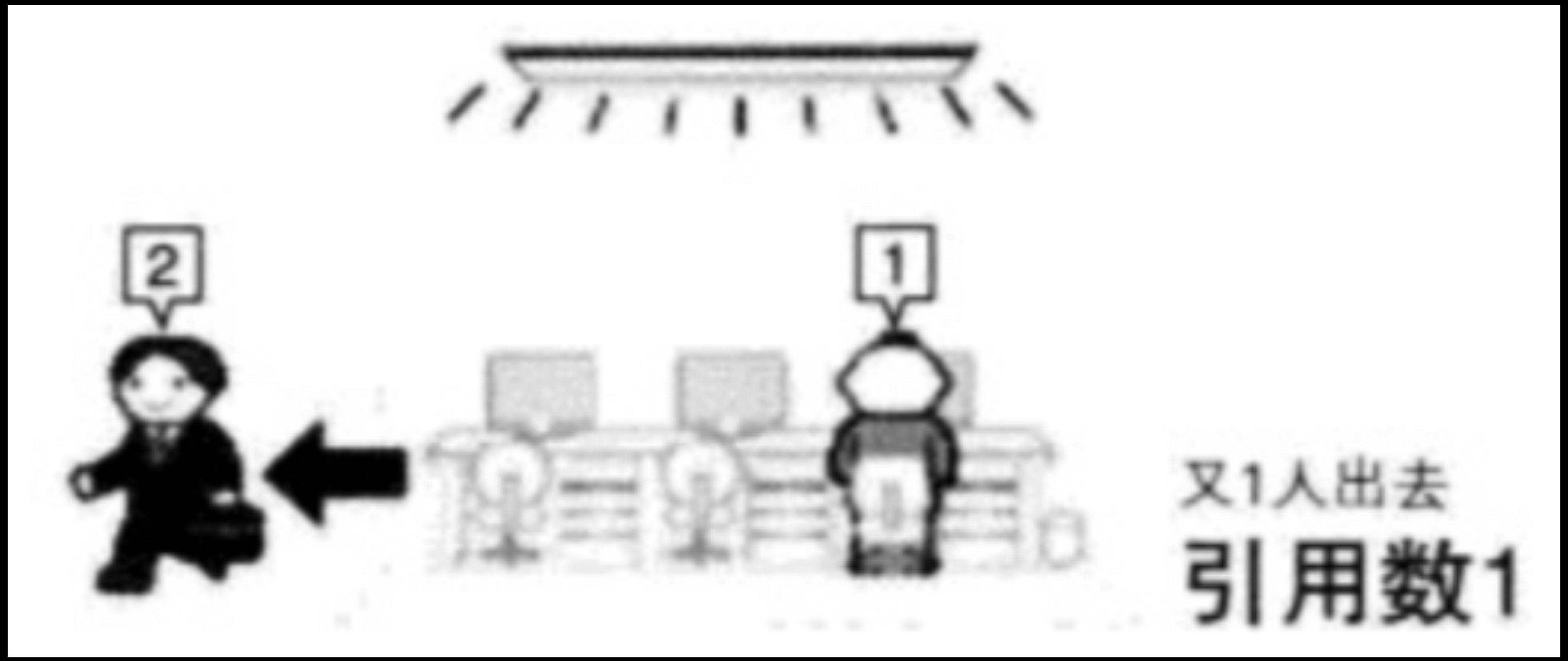
下班离开办公室不需要照明

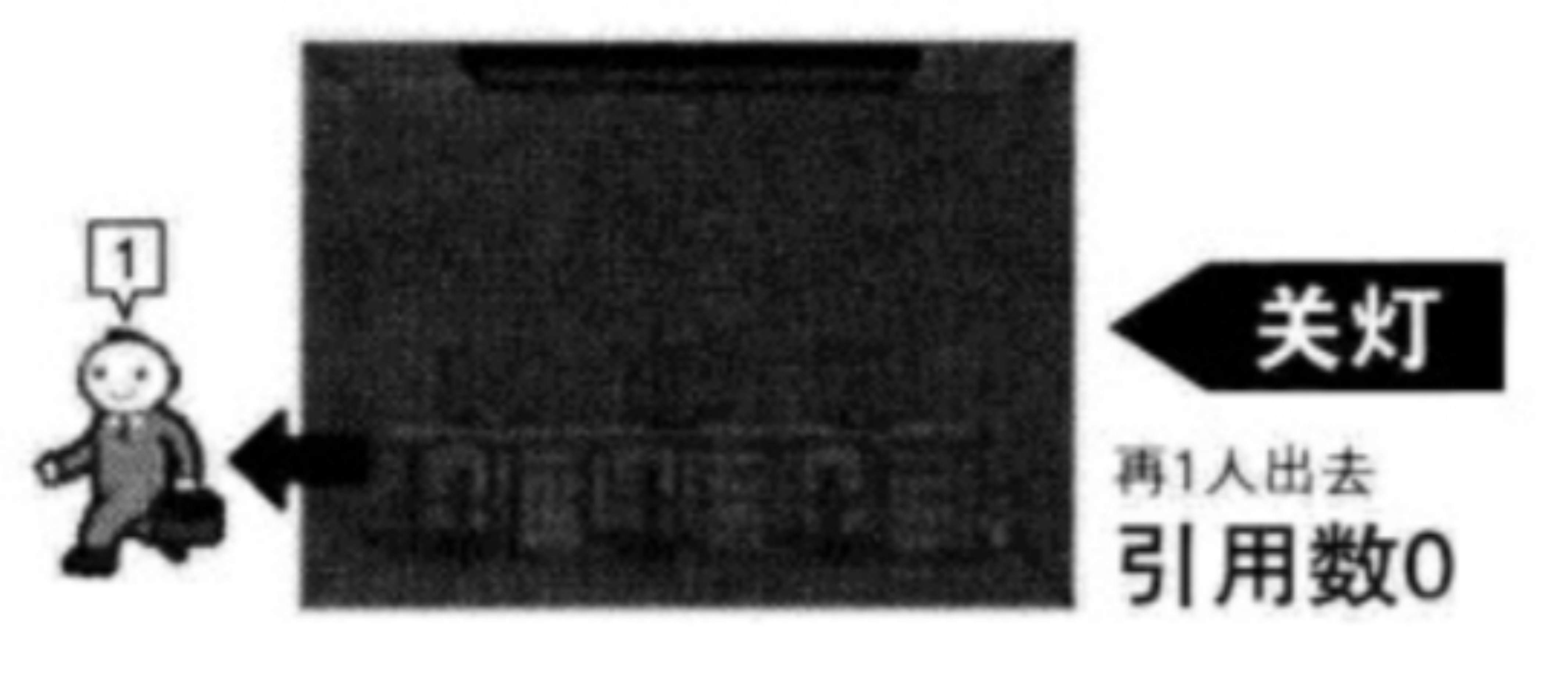
引用计数 - 增加



引用计数 - 减少

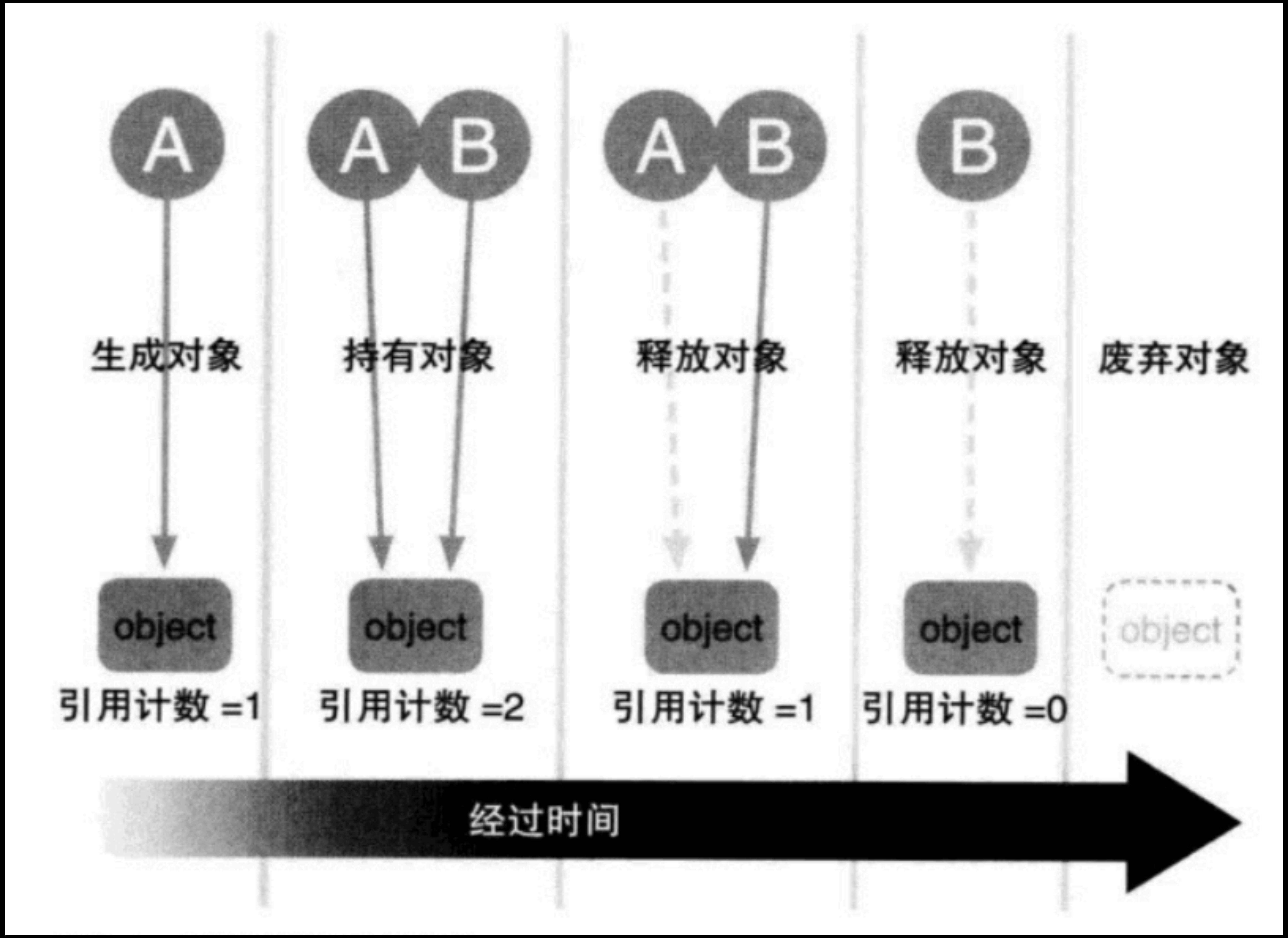






手动引用计数
（MRC）

MRC



生成：alloc、new、copy等

持有：retain

释放：release、autorelease

废弃：dealloc

显示：retainCount

MRC - 关键字

	MRC
生成	alloc、new、copy
持有	retain
不持有	assign
基本类型（int、float、bool）	assign
释放	release、autorelease
废弃	dealloc(需要调用super方法)
显示	retainCount

MRC - 代码示例

```
@class Dog;

@interface People : NSObject
@property(nonatomic, retain) NSString *name;
@property(nonatomic, assign) NSInteger age;
@property(nonatomic, assign) BOOL male;
@property(nonatomic, retain) Dog *dog;
@end


@implementation People
- (void)dealloc
{
    [_dog release];
    [_name release];
    [super dealloc];
}
@end
```

MRC - 代码示例

```
@class People;

@interface Dog : NSObject
@property(nonatomic, retain) NSString *name;
@property(nonatomic, assign) NSInteger weight;
@property(nonatomic, assign) People *owner;
@end


@implementation Dog
- (void)dealloc
{
    [_name release];
    [super dealloc];
}
@end
```

MRC - 代码示例

```
- (void)func0 {
    People *people = [[People alloc] init];
    [people release];
}

- (void)func1 {
    People *people = [[People alloc] init];
    Dog *dog = [[Dog alloc] init];
    NSLog(@"retain count0:%lu", [dog retainCount]); // 1
    people.dog = dog;
    NSLog(@"retain count1:%lu", [dog retainCount]); // 2
    [people release];
    NSLog(@"retain count2:%lu", [dog retainCount]); // 1
    [dog release];
    NSLog(@"retain count3:%lu", [dog retainCount]); // ?
}
```

MRC - AutoRelease解决的问题

```
- (People *)createPeople {
    People *people = [[People alloc] init];
    people.name = @"小王";
    people.age = 10;
    people.male = YES;
    return people;
}

- (void)func2 {
    People *people = [self createPeople];
    [people trainingDog];
    [people release];
}
```

MRC - AutoRelease

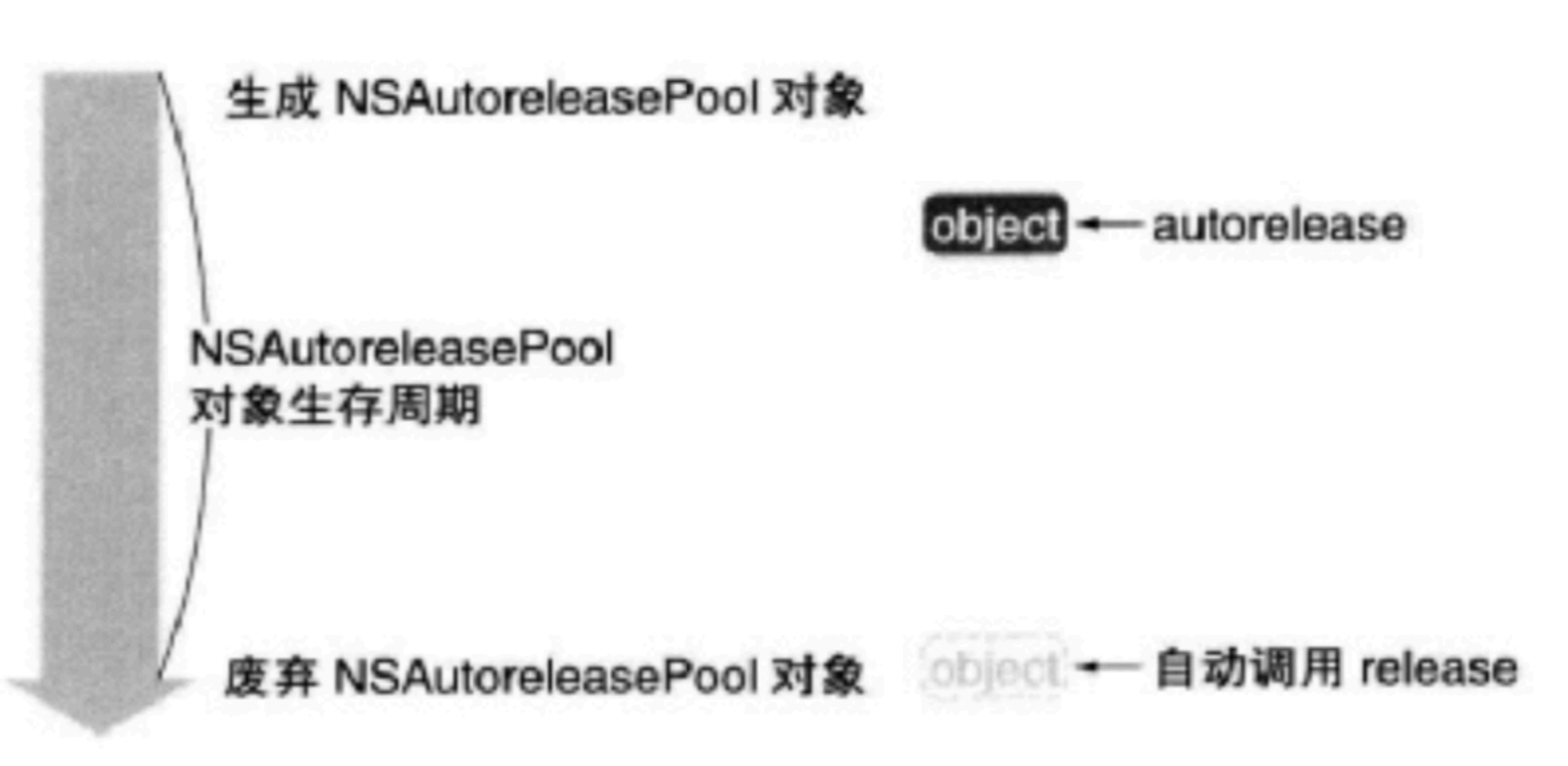
```
- (People *)createPeople {
    People *people = [[People alloc] init];
    people.name = @"小王";
    people.age = 10;
    people.male = YES;
    return people;
}

- (void)func2 {
    People *people = [self createPeople];
    [people trainingDog];
    [people release];
}
```

→

```
- (People *)createPeople {
    People *people = [[[People alloc] init] autorelease];
    people.name = @"小王";
    people.age = 10;
    people.male = YES;
    return people;
}

- (void)func2 {
    People *people = [self createPeople];
    [people trainingDog];
}
```

MRC - AutoReleasePool

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
People *people = [[[People alloc] init] autorelease];
people.name = @"小王";
[pool release]; // drain
```



MRC - AutoReleasePool

内存峰值问题：

100次循环，每次处理10MB文件

```
    for (int i = 0; i < 100; i++) {
        NSString *fileContents = [NSString stringWithContentsOfURL:urlArray[i]
encoding:NSUnicodeStringEncoding error:nil];
        NSLog(@"%@", fileContents);
    }
```

MRC - AutoReleasePool

减少内存峰值

```
    for (int i = 0; i < 100; i++) {
        NSString *fileContents = [NSString stringWithContentsOfURL:urlArray[i]
encoding:NSUnicodeStringEncoding error:nil];
        NSLog(@"%@", fileContents);
    }
```

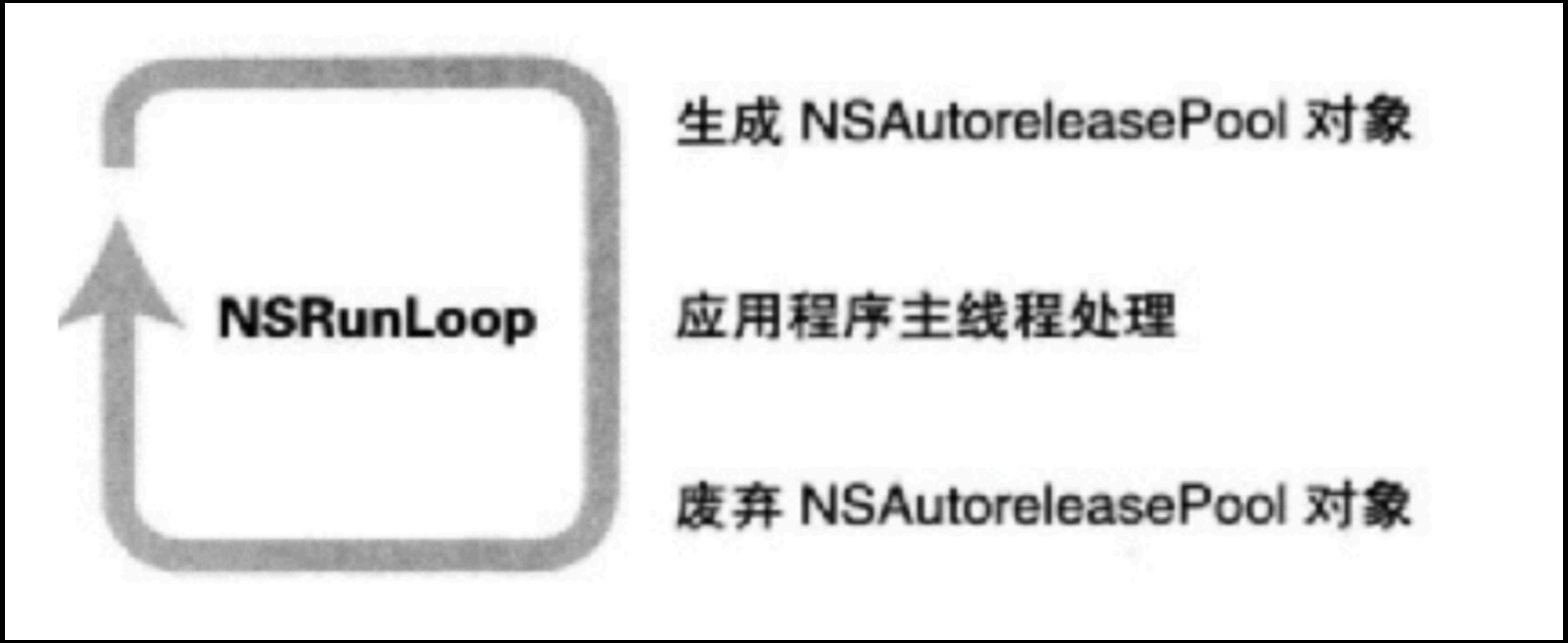
↓

```
    for (int i = 0; i < 100; i++) {
        NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
        NSString *fileContents = [NSString stringWithContentsOfURL:urlArray[i]
encoding:NSUnicodeStringEncoding error:nil];
        NSLog(@"%@", fileContents);
        [pool release];
    }
```

MRC - Autorelease Pool

线程在一个Autorelease Pool的上下文中执行，线程任务完成后销毁

主线程，不会销毁，在一次事件循环结束后，清理Autorelease Pool

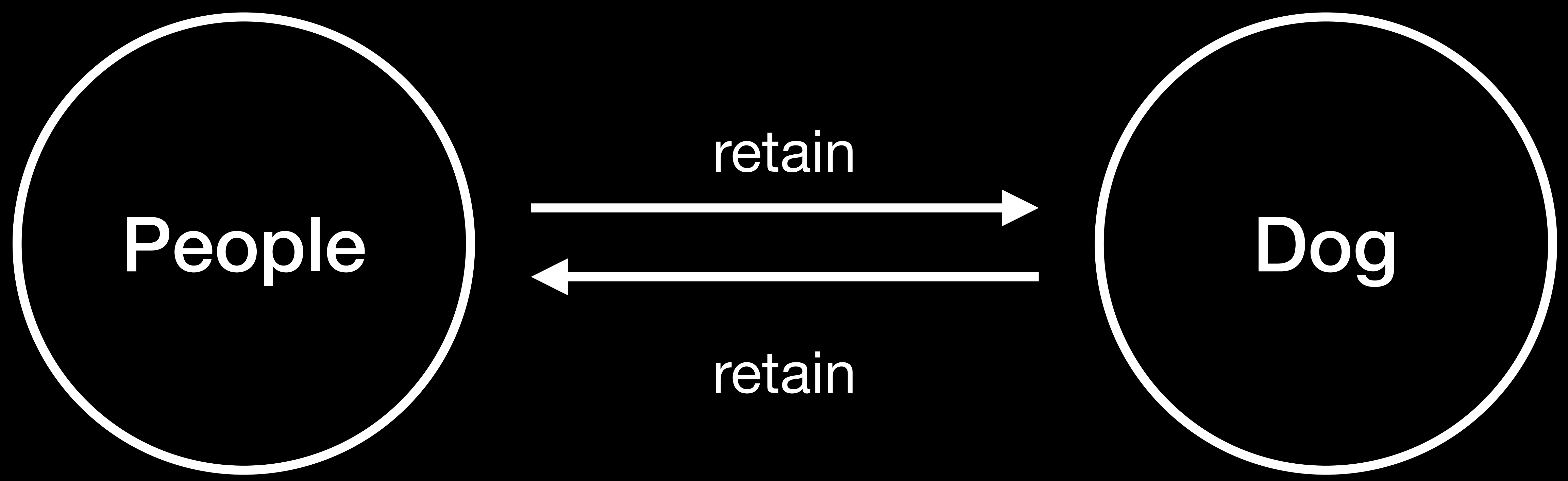


生成 NSAutoreleasePool 对象

应用程序主线程处理

废弃 NSAutoreleasePool 对象

MRC - 循环引用



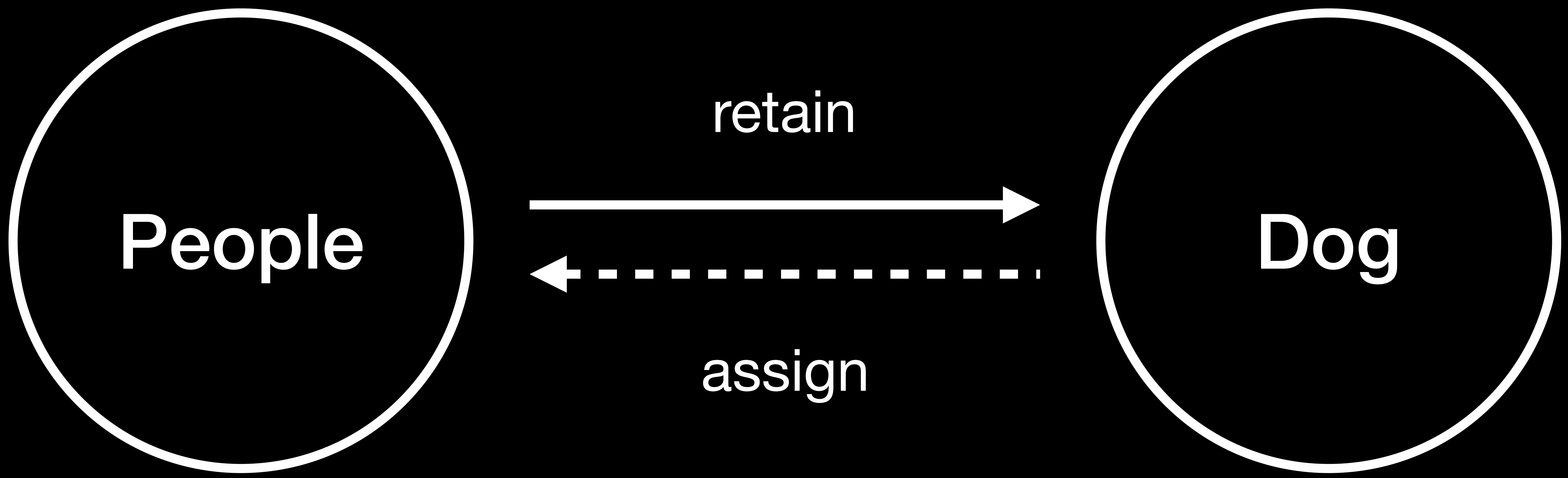
People 持有 Dog

Dog持有 People

互相持有，无法调用dealloc方法

```
@interface People : NSObjec
@property(nonatomic, retain) Dog *dog;
@end

@interface Dog : NSObject
@property(nonatomic, retain) People *owner;
@end
```

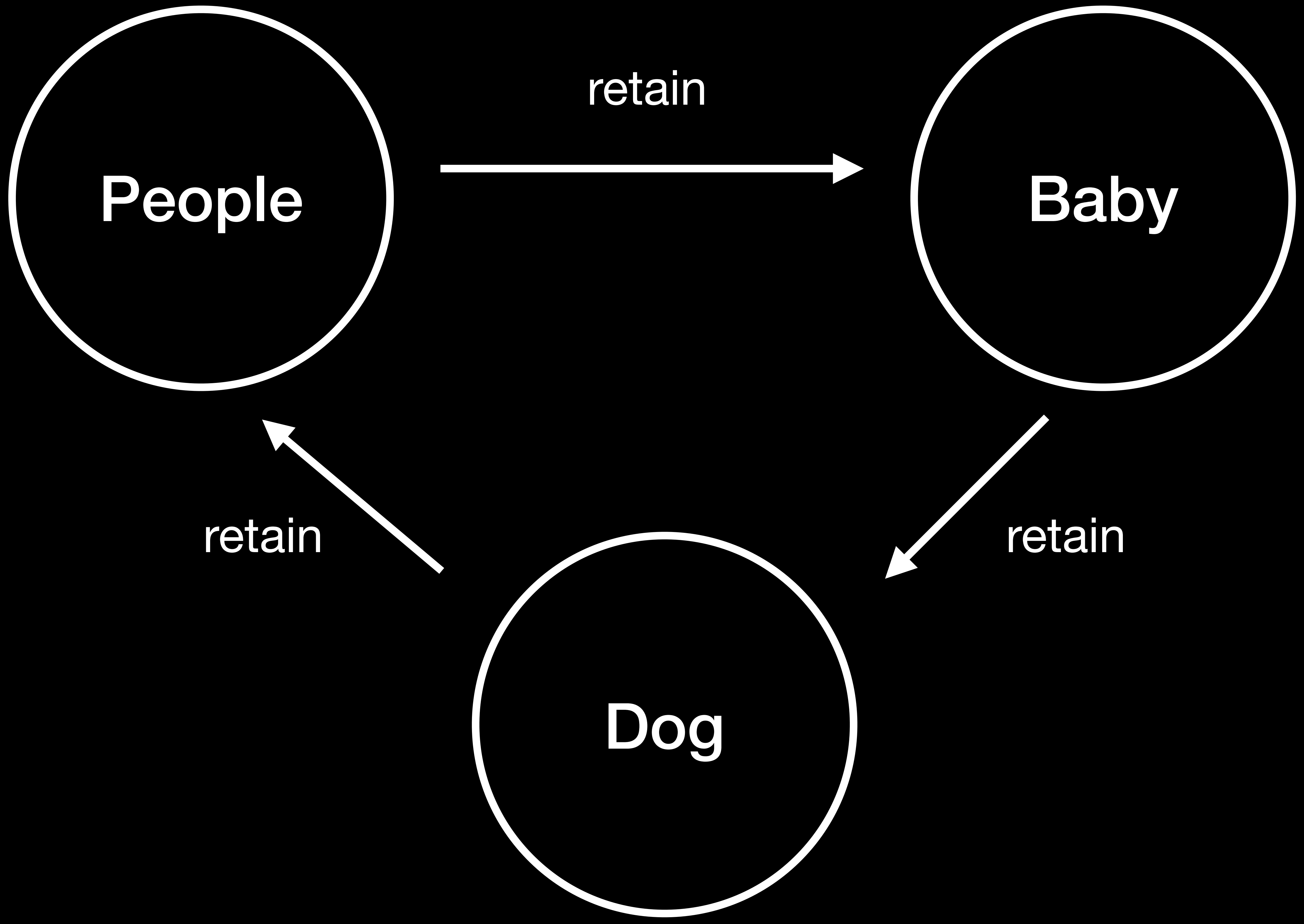
MRC - 解决循环引用



retain 改为 assign即可

```
@interface People : NSObjec
@property(nonatomic, retain) Dog *dog;
@end

@interface Dog : NSObject
@property(nonatomic, assign) People *owner;
@end
```

MRC - 多个对象成环



多个对象相互引用成环

解决任意一个retain即可

MRC - 问题

只是解决了部分显式内存释放问，并没有彻底根治

- 业务无关的retain、release、autorelease代码
- setter
- 野指针问题

MRC - 问题

```
@implementation People

- (void)setDog:(Dog *)dog {
    if (_dog != dog) { // 判断不同的dog
        [_dog release]; // 将原来的dog释放
        _dog = [dog retain]; // 持有新dog并赋值
    }
}
@end
```

MRC - 僵尸对象、野指针

一个对象被释放了，我们就称这个对象为 "僵尸对象(不能再使用的对象)"

当一个指针指向一个僵尸对象(不可用内存)，我们就称这个指针为野指针

只要给一个野指针发送消息就会报错(EXC_BAD_ACCESS错误)

```
int main(int argc, const char * argv[]) {
    People *people = [[People alloc] init]; // 执行完引用计数为1
    [people release]; // 执行完引用计数为0，实例对象被释放
    [people trainingDog]; // 此时，people变成了野指针，再给野指针就发送消息就会报错
    return 0;
}
```

为了避免给野指针发送消息，对象被释放后会将这个对象的指针设置为空指针

MRC - 空指针

没有指向存储空间的指针(里面存的是nil, 也就是0)

给空指针发消息是没有任何反应的

```
int main(int argc, const char * argv[]) {
    People *people = [[People alloc] init]; // 执行完引用计数为1
    [people release]; // 执行完引用计数为0，实例对象被释放
    people = nil;
    [people trainingDog]; // 此时，people变成了野指针，再给野指针就发送消息就会报错
    return 0;
}

@implementation People
- (void)dealloc
{
    [_dog release];
    _dog = nil;
    [super dealloc];
}
@end
```

MRC - Xcode上机练习

建立MRC新工程

创建People类，尝试retain、release、dealloc、retainCount方法

创建Dog类，和People相互依赖，尝试retain、assign修饰property

解决循环引用问题

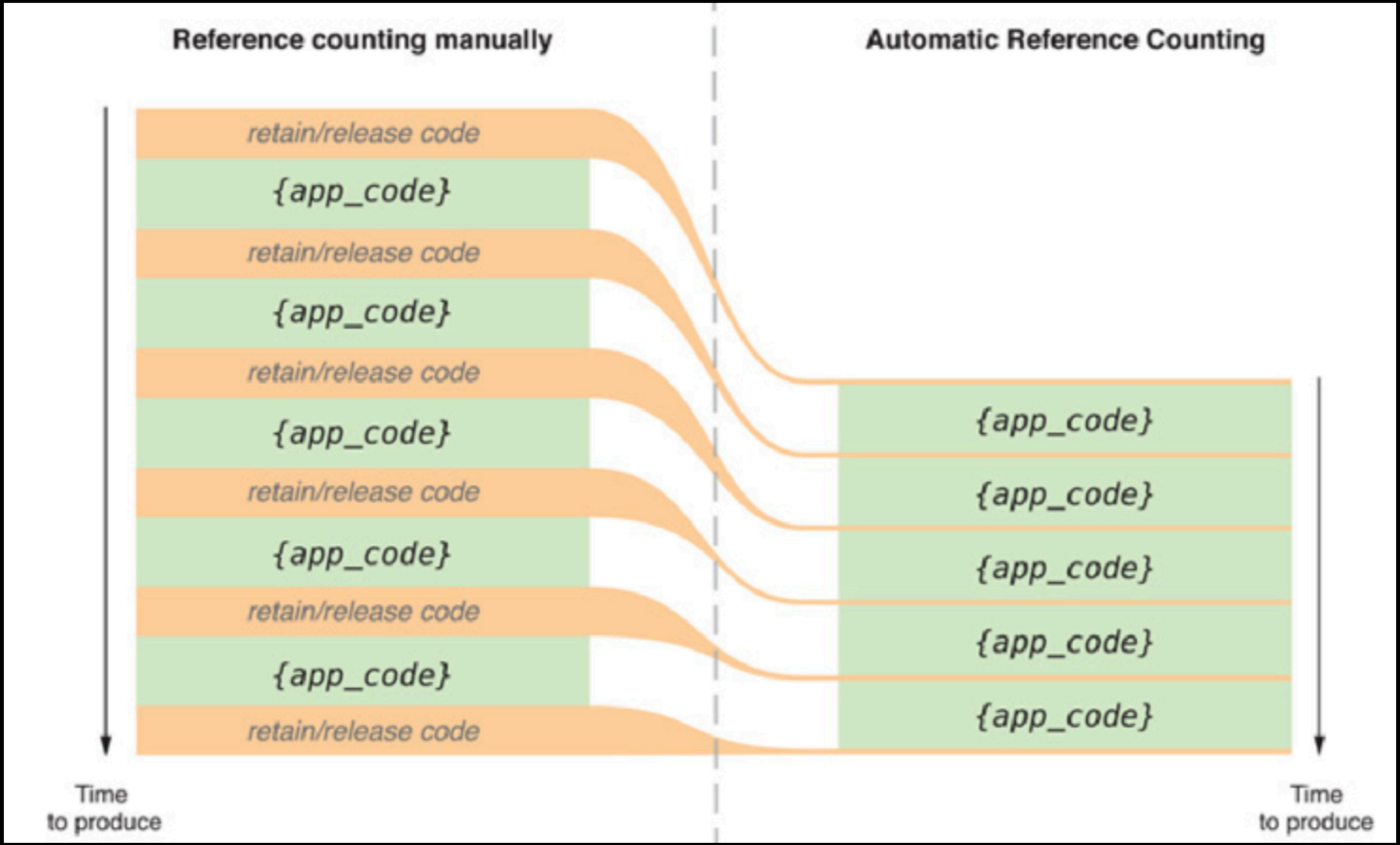
尝试autorelease方法

自动引用计数
（ARC）

ARC - Automatic Reference Counting

系统会检测出何时需要保持对象，何时需要自动释放对象，何时需要释放对象，编译器会管理好对象的内存，会在何时的地方插入retain, release和autorelease，通过生成正确的代码去自动释放或者保持对象。

ARC


Reference counting manually
Automatic Reference Counting
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
retain/release code
{app_code}
{app_code}
{app_code}
{app_code}
{app_code}
Time
to produce
Time
to produce

ARC - 关键字

	MRC	ARC
生成	alloc、new、copy	alloc、new、copy等
持有	retain	strong
不持有	assign	weak
基本类型	assign	assign
释放	release、autorelease	无
废弃	dealloc	dealloc(保留，但不需要调用super)
显示	retainCount	无

ARC - 代码示例

```
@class Dog;

@interface People : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, assign) NSInteger age;
@property(nonatomic, assign) BOOL male;
@property(nonatomic, strong) Dog *dog;
@end

@implementation People

@end
```

ARC - 代码示例

```
@class People;

@interface Dog : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, assign) NSInteger weight;
@property(nonatomic, weak) People *owner;
@end


@implementation Dog

@end
```

ARC - 代码示例

```
- (void)func0 {
    People *people = [[People alloc] init];
    NSLog(@"%@", people);
}

- (void)func1 {
    People *people = [[People alloc] init];
    Dog *dog = [[Dog alloc] init];
    people.dog = dog;
}

- (People *)createPeople {
    People *people = [[People alloc] init];
    people.name = @"小王";
    people.age = 10;
    people.male = YES;
    return people;
}
```

MRC - AutoReleasePool

减少内存峰值

```
    for (int i = 0; i < 100; i++) {
        NSString *fileContents = [NSString stringWithContentsOfURL:urlArray[i]
encoding:NSUnicodeStringEncoding error:nil];
        NSLog(@"%@", fileContents);
    }
```

MRC - AutoReleasePool

减少内存峰值

```
    for (int i = 0; i < 100; i++) {
        NSString *fileContents = [NSString stringWithContentsOfURL:urlArray[i]
encoding:NSUnicodeStringEncoding error:nil];
        NSLog(@"%@", fileContents);
    }
```

↓

```
    for (int i = 0; i < 100; i++) {
        @autoreleasepool {
            NSString *fileContents = [NSString stringWithContentsOfURL:urlArray[i]
encoding:NSUnicodeStringEncoding error:nil];
            NSLog(@"%@", fileContents);
        }
    }
```

ARC weak

不持有对象

当对象没有强引用的时候自动置为nil

ARC - Core Foundation

CFRetain

CFRelease

CFAutorelease

CFGetRetainCount

ARC - Core Foundation - TheCreate Rule

名字里有Create的对象创建方法

名字里有Copy的对象复制方法

CFTimeZoneRef CFTimeZoneCreateWithTimeIntervalFromGMT (CFAllocatorRef allocator, CFTimeInterval ti);

CFDictionaryRef CFTimeZoneCopyAbbreviationDictionary (void);

CFBundleRef CFBundleCreate (CFAllocatorRef allocator, CFURLRef bundleURL);

Copy

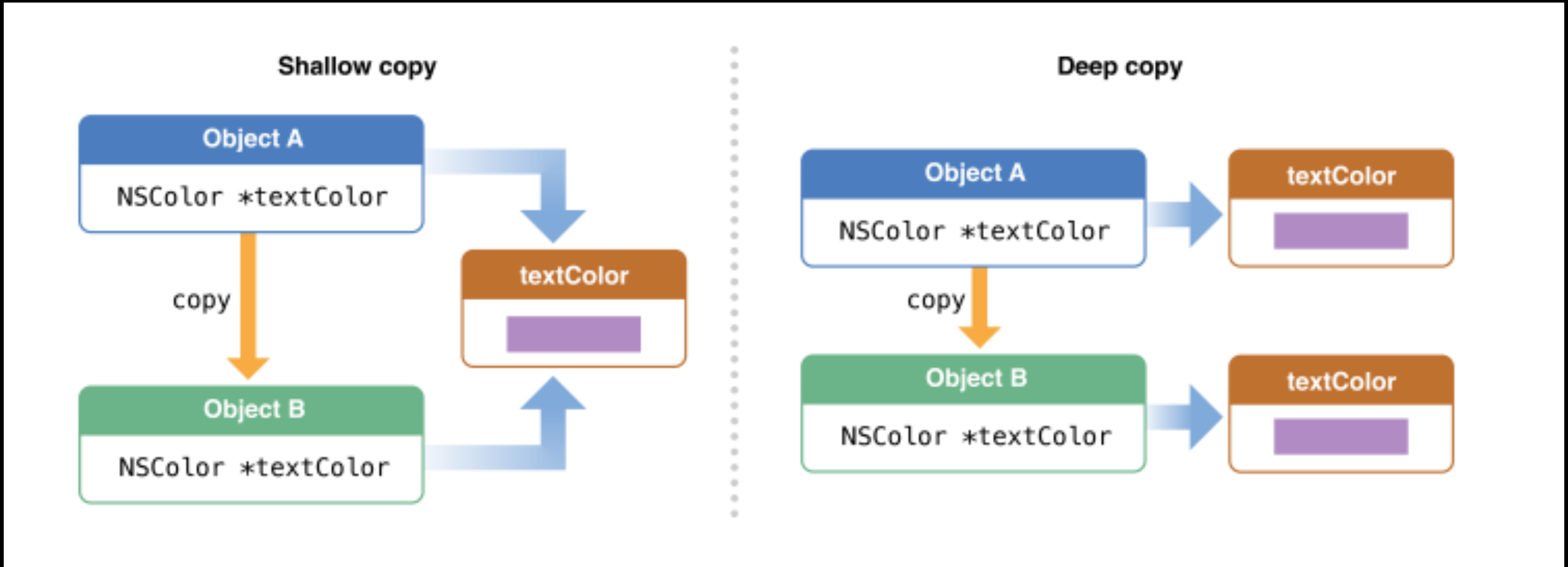
字面意思是“复制”、“拷贝”，产生一个副本

- 修改源对象的属性和行为，不会影响副本对象
- 修改副本对象的属性和行为，不会影响源对象

Copy - 拷贝类型

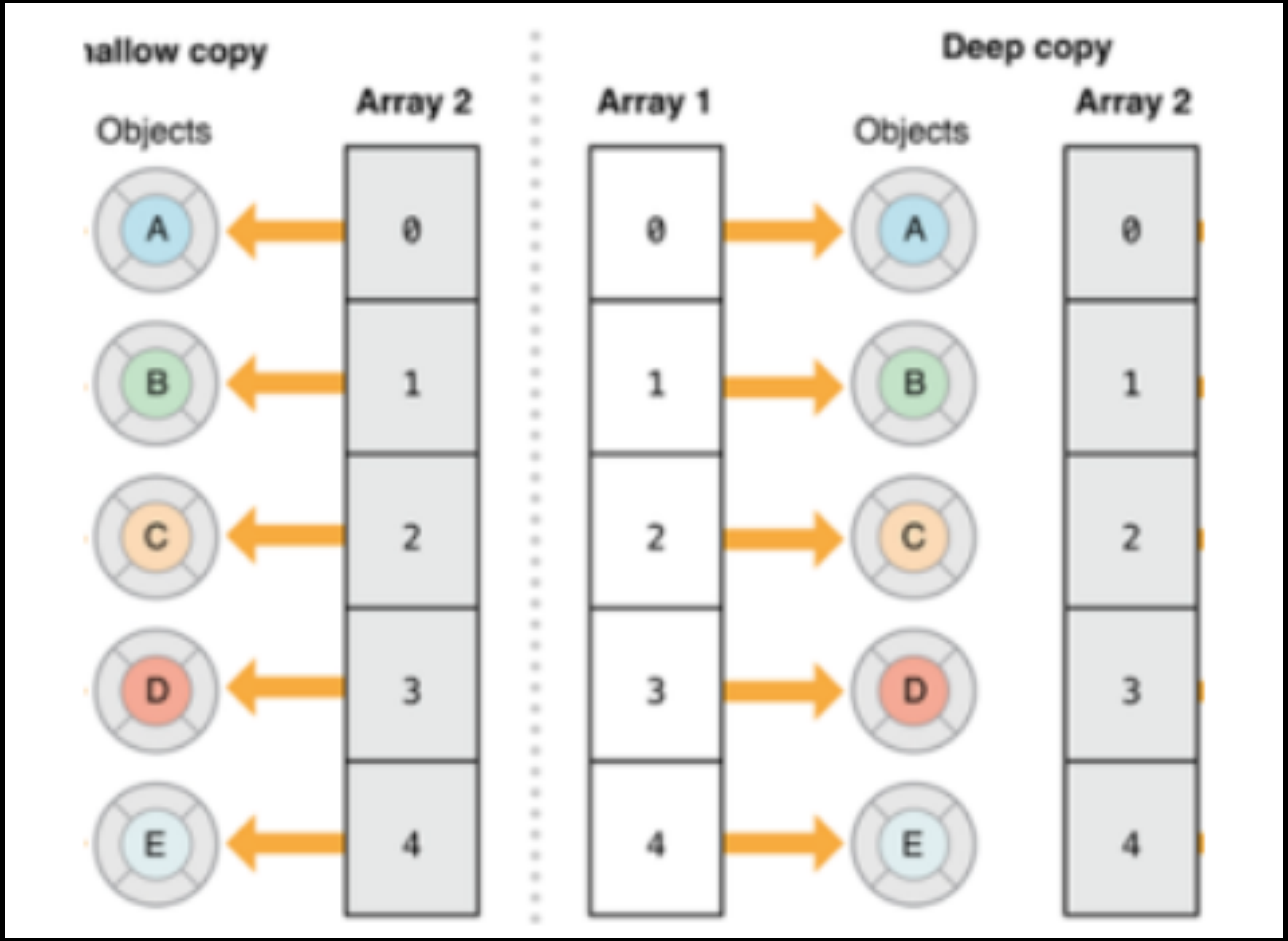
按类型分为：

- 浅拷贝：仅仅对内存地址进行了拷贝，并没有对源对象进行拷贝
- 深拷贝：不仅对内存地址进行拷贝，而且对源对象进行拷贝



Copy - 集合

NSArray、NSDictionary中的元素都是浅拷贝



Copy

属性声明

```
@interface People : NSObject
@property(nonatomic, copy) NSString *name;
@end
```

调用NSObject基类的copy方法

```
- (void)copyDog {
    People *people = [[People alloc] init];
    Dog *dog = [[Dog alloc] init];
    people.dog = [dog copy];
}
```

NSCopying协议

调用copy方法，需要实现NSCopying协议，否则crash

```
@class People;
@interface Dog : NSObject <NSCopying>
@property(nonatomic, strong) NSString *name;
@property(nonatomic, assign) NSInteger weight;
@property(nonatomic, weak) People *owner;

@end

@implementation Dog
- (id)copyWithZone:(nullable NSZone *)zone {
    Dog *dog = [[Dog alloc] init];
    dog.name = self.name;
    dog.weight = self.weight;
    dog.owner = self.owner;
    return dog;
}
@end
```

MutableCopy

字符串、数组、字典等对象可以设计成不可变、可变两种模式

不可变的优点：对外提供的对象无法修改，不会影响原对象的内容

```
@interface NSArray<ObjectType>: NSObject <NSCopying, NSMutableCopying>
@property (readonly) NSUInteger count;
- (ObjectType)objectAtIndex:(NSUInteger)index;
- (instancetype)initWithObjects:(const ObjectType _Nonnull [_Nullable])objects
count:(NSUInteger)cnt NS_DESIGNATED_INITIALIZER;
@end

@interface NSMutableArray<ObjectType>: NSArray
- (void)addObject:(ObjectType)anObject;
- (void)insertObject:(ObjectType)anObject atIndex:(NSUInteger)index;
- (void)removeLastObject;
@end
```

NSMutableCopying协议

```
@class People;
@interface Dog : NSObject <NSMutableCopying>
@property(nonatomic, strong) NSString *name;
@property(nonatomic, assign) NSInteger weight;
@property(nonatomic, weak) People *owner;

@end

@implementation Dog
- (id)mutableCopyWithZone:(nullable NSZone *)zone {
    Dog *dog = [[Dog alloc] init];
    dog.name = self.name;
    dog.weight = self.weight;
    dog.owner = self.owner;
    return dog;
}
@end
```

ARC - Xcode上机练习

建立ARC新工程

创建People类

创建Dog类，和People相互依赖，尝试strong、copy、assign修饰property

解决循环引用问题

注意事项

ARC - NSTimer内存泄漏问题

鼠标悬停NSTimer方法上方，Optional按键+鼠标左键点击API帮助文档，弹出提示框：

```
NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:0.25 target:self
selector:@selector(startTimer) userInfo:nil repeats:YES];
[[NSRunLoop currentRunLoop] addTimer:timer forMode:NSRunLoopCommonModes];
self.timer = timer;
```

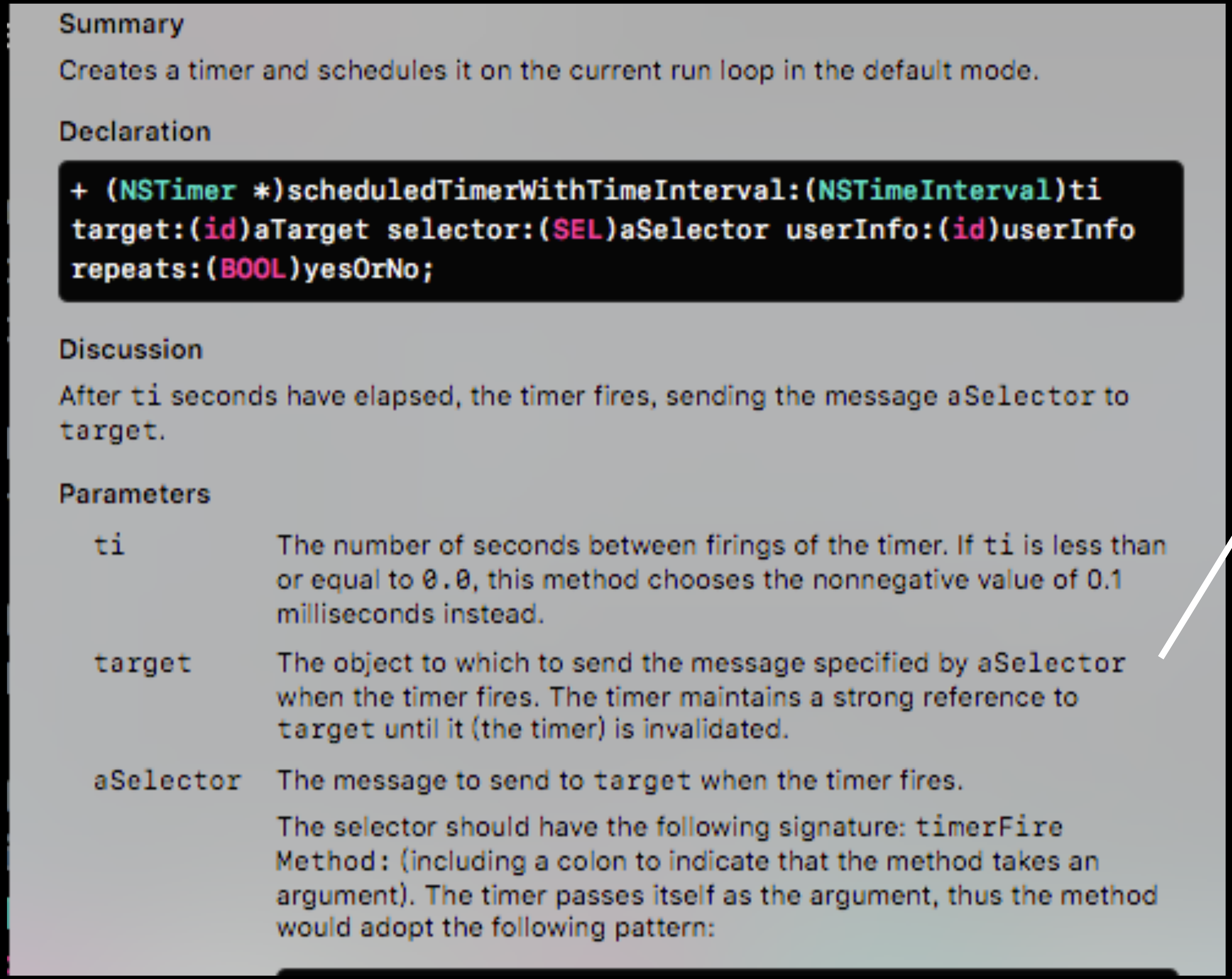
鼠标悬停NSTimer方法上方，Optional按键+鼠标左键点击API帮助文档，弹出提示框：

ARC - NSTimer内存泄漏问题

NSTimer是定时器，可以定时执行代码

```
NSTimer *timer = [NSTimer scheduledTimerWithTimeInterval:0.25 target:self
selector:@selector(startTimer) userInfo:nil repeats:YES];
[[NSRunLoop currentRunLoop] addTimer:timer forMode:NSRunLoopCommonModes];
self.timer = timer;
```

鼠标悬停NSTimer方法上方，Optional按键+鼠标左键点击API帮助文档，弹出提示框：



Summary

Creates a timer and schedules it on the current run loop in the default mode.

Declaration

```
+ (NSTimer *)scheduledTimerWithTimeInterval:(NSTimeInterval)ti
target:(id)aTarget selector:(SEL)aSelector userInfo:(id)userInfo
repeats:(BOOL)yesOrNo;
```

Discussion

After ti seconds have elapsed, the timer fires, sending the message aSelector to target.

Parameters

ti The number of seconds between firings of the timer. If ti is less than or equal to 0.0, this method chooses the nonnegative value of 0.1 milliseconds instead.

target The object to which to send the message specified by aSelector when the timer fires. The timer maintains a strong reference to target until it (the timer) is invalidated.

aSelector The message to send to target when the timer fires.

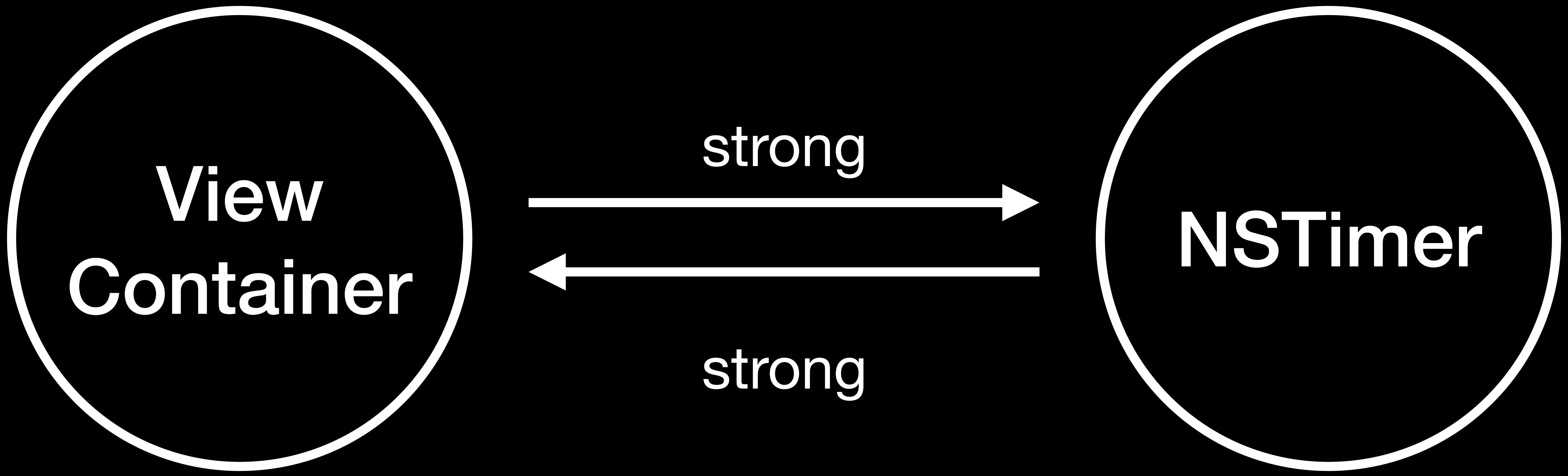
The selector should have the following signature: timerFireMethod: (including a colon to indicate that the method takes an argument). The timer passes itself as the argument, thus the method would adopt the following pattern:

The object to which to send the message specified by aSelector when the timer fires. The timer maintains a strong reference to target until it (the timer) is invalidated.

ARC - NSTimer问题

因为是循环引用，所以对象无法调用dealloc方法，在dealloc中无法处理释放过程

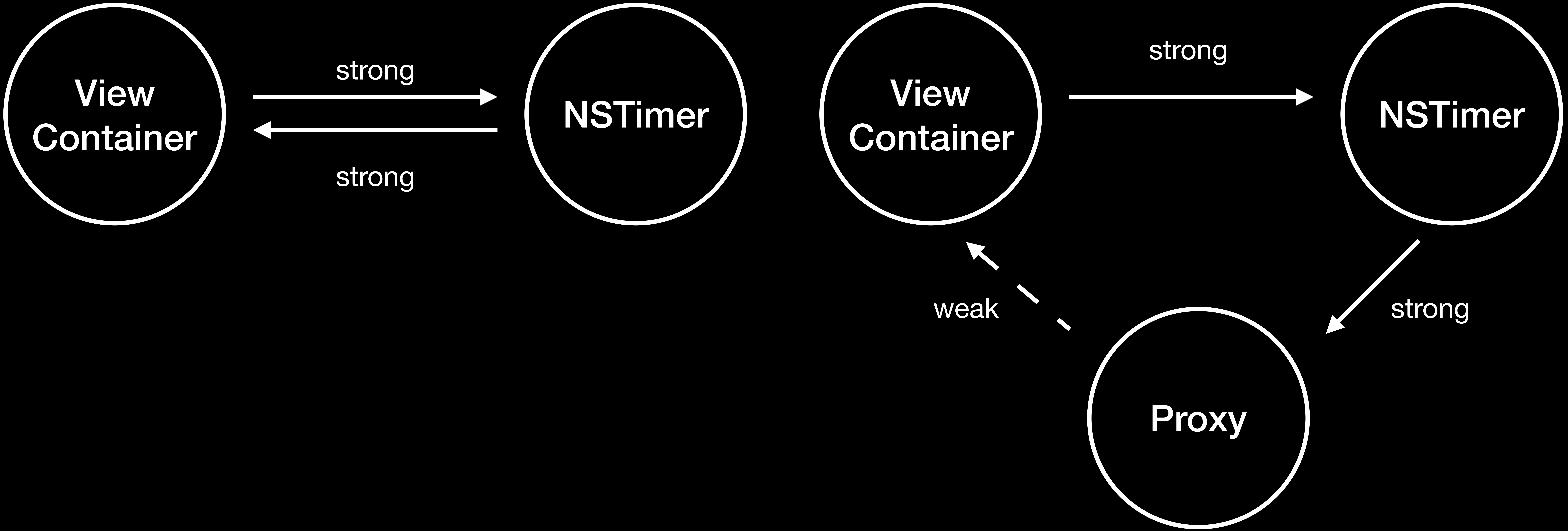
- 在页面隐藏的时候释放，页面显示时重新创建
- 代理



ARC - NSTimer问题

因为是循环引用，所以对象无法调用dealloc方法，在dealloc中无法处理释放过程

- 在页面隐藏的时候释放，页面显示时重新创建
- 代理



ByteDance
字节跳动

Objective-C Blocks

session 2

张宇 抖音iOS开发工程师

© 2019 Bytedance Inc. All rights reserved. Redistribution or public display not permitted without written permission from Bytedance.

Blocks - 基础知识

块是C、C++、Objective-C 中的词法闭包。

块可接受参数，也可返回值。

块可以分配在栈或堆上，也可以是全局的。分配在栈上的块可拷贝到堆里，和标准的 Objective-C 对象一样，具备引用计数了。

Blocks - 基础知识

```
^{
    // Block implementation here
};

^{
    // Block implementation here
}();

void (^someBlock)() = ^{
    // Block implementation here
};
someBlock();
```

Blocks - 基础知识

```
return_type (^block_name) (parameters)

int (^addBlock)(int a, int b) = ^(int a, int b){
    return a + b;
}
```

Blocks - 基础知识

```
int additional = 5;
int (^addBlock)(int a, int b) = ^(int a, int b){
    return a + b + addItional;
};

int add = addBlock(2, 5); // < add = 12
```

Blocks - 为常用的块类型创建typedef

typedef return_type (^block_name)(parameters);

```
typedef void (^CompletionHandler)(void);

typedef int (^ComputeHandler)(int a, id b, NSObject *c);
typedef NSObject * (^ComputeHandler)(int a, id b, NSObject *c);
```

Blocks - 修改局部变量

```
- (void)changeValue {
    int value = 0;

    void (^someBlock)(void) = ^{
        NSLog(@"value:%i", value); // value:?
    };
    value = 1;
    someBlock();
    NSLog(@"value:%i", value); // value:?
}
```

Blocks - 修改局部变量

```
- (void)changeValue {
    int value = 0;

    void (^someBlock)(void) = ^{
        NSLog(@"value:%i", value); // value:0
    };
    value = 1;
    someBlock();
    NSLog(@"value:%i", value); // value:1
}
```

Blocks - 修改局部变量

```
- (void)changeValue {
    __block int value = 0;

    void (^someBlock)(void) = ^{
        NSLog(@"value:%i", value); // value:1
        value = 2;
    };
    value = 1;
    someBlock();
    NSLog(@"value:%i", value); // value:2
}
```

Blocks - 循环引用

```
@interface ViewController ()
@property (nonatomic, copy) NSString *name;
@property (nonatomic, strong) CompletionHandler handler;
@end

@implementation ViewController

- (void)blocks {
    self.handler = ^{
        NSLog(@"%@", self.name);
    };
}

@end
```

Xcode warning： Capturing 'self' strongly in this block is likely to lead to a retain cycle

Blocks - 循环引用

```
@interface ViewController ()
@property (nonatomic, copy) NSString *name;
@property (nonatomic, strong) CompletionHandler handler;
@end

@implementation ViewController

- (void)blocks {
    __weak __typeof(self)weakSelf = self;
    self.handler = ^{
        __strong __typeof(weakSelf)strongSelf = weakSelf;
        NSLog(@"%@", strongSelf.name);
    };
}

@end
```

Blocks - 循环引用

```
@interface ViewController ()
@property (nonatomic, copy) NSString *name;
@property (nonatomic, strong) CompletionHandler handler;
@end

@implementation ViewController

- (void)blocks {
    self.handler = ^{
        NSLog(@"%@", _name);
    };
}

@end
```

Blocks - 数组遍历

```
NSArray *array = @[@0, @1, @2, @3, @4, @5];
for (int i =0; i < array.count; i++) {
    // code
}

for (NSNumber *number in array) {
    // code
}

[array enumerateObjectsUsingBlock:^(NSNumber *number, NSUInteger idx, BOOL *
_Nonnull stop) {
    if([number isEqualToNumber:@(2)]) {
        *stop = YES; // 等于break
    }
}];
```

Blocks - Xcode上机练习

执行匿名block，输入block

定义简单的block并执行，输入block

定义复杂的addBlock并执行

block修改局部变量

解决block循环引用问题

NSArray block变量

ByteDance
字节跳动

Objective-C 通信方式

session 2

张宇 抖音iOS开发工程师

© 2019 Bytedance Inc. All rights reserved. Redistribution or public display not permitted without written permission from Bytedance.

主要内容

Delegate

Block

NSNotification

KVC、KVO

Delegate

通信方式 - Delegate

delegate是委托模式，委托模式是将一件属于委托者做的事情，交给另外一个被委托者来处理

通信方式 - 强耦合问题

```
@class Dog;

@interface People : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, assign) NSInteger age;
@property(nonatomic, assign) BOOL male;
@property(nonatomic, strong) Dog *dog;
@end


@class People;

@interface Dog : NSObject
@property(nonatomic, strong) NSString *name;
@property(nonatomic, assign) NSInteger weight;
@property(nonatomic, weak) People *owner;
@end
```

通信方式 - 强耦合问题

```
@implementation Dog

- (void)hungry {
    [self.owner feed];
}

@end
```

通信方式 - Delegate

一个标准的委托由以下部分组成

```
@protocol FeederDelegate <NSObject>
- (void)feed;
@end


@interface People : NSObject<FeederDelegate>
@end

@interface Dog : NSObject
@property(nonatomic, weak) id<FeederDelegate> delegate;
@end
```

通信方式 - Delegate

```
- (People *)createPeople {
    People *people = [[People alloc] init];
    Dog *dog = [[Dog alloc] init];
    dog.delegate = people;
    people.dog = dog;
    return people;
}
```

通信方式 - Delegate

```
@implementation Dog

- (void)hungry {
    // eat
    if (self.delegate && [self.delegate respondsToSelector:@selector(feed)]) {
        [self.delegate feed];
    }
}

@end
```

通信方式 - Delegate

```
- (void)createButton {
    UIButton *button = [[UIButton alloc] init];
    [button addTarget:self action:@selector(buttonHandler:)
forControlEvents:UIControlEventTouchUpInside];
}

- (void)buttonHandler:(UIButton *)button {
    // 点击按钮
}
```

通信方式 - Delegate - Xcode上机练习

基于People和Dog对象，定义Protocol，并实现方法

Block

通信方式 - Block

```
typedef void (^FeedHandler)(void);

@class People;
@interface Dog : NSObject
@property (nonatomic, copy) FeedHandler handler;
@end

@implementation Dog
- (void)hungry {
    if (self.handler != nil) {
        self.handler();
    }
}
@end
```

通信方式 - Block

```
@implementation People

- (void)setDog:(Dog *)dog {
    if (_dog != dog) { // 判断不同的dog
        _dog = dog;
        __weak __typeof(self)weakSelf = self;
        _dog.handler = ^{
            __strong __typeof(weakSelf)strongSelf = weakSelf;
            [strongSelf cook];
            [strongSelf feedPet];
            [strongSelf clear];
        };
    }
}

@end
```

通信方式 - Block - Xcode上机练习

基于People和Dog对象，实现block方式通信

NSNotification

通信方式 - Delegate、Block问题

Delegate、Block可以解决大部分问题

但是以下几种情况无法解决或不方便：

- 一对多
- 依赖关键远（比如两个页面间的通信）

通信方式 - NSNotification

添加通知

```
[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(notificationFirst:)

name:@"people.name" object:nil];
```

发送通知

```
[[NSNotificationCenter defaultCenter] postNotificationName:@"people.name"
object:@"小王"];
```

回调方法

```
- (void)notificationFirst:(NSNotification *)notification {
    NSLog(@"people.name: %@", notification.object); // people.name: 小王
}
```

通信方式 - NSNotification

```
@implementation AppDelegate
- (void)applicationWillEnterForeground:(UIApplication *)application {
    [[NSNotificationCenter defaultCenter] postNotificationName:@"name.change"
object:nil];
}
@end
```

例如实现从后台激活时，通知需要的对象

通信方式 - NSNotification - Xcode上机练习

基于People和Dog对象，实现NSNotification方式通信

KVC、KVO

通信方式 - 什么是KVC

Key-Value Coding，即键值编码。它是一种不通过存取方法，而通过属性名称字符串间接访问属性的机制。

```
-(void)setValue:(nullable id)value forKey:(NSString *)key;
-(nullable id)valueForKey:(NSString *)key;
-(nullable id)valueForKeyPath:(NSString *)keyPath;
-(void)setValue:(nullable id)value forKeyPath:(NSString *)keyPath;
```

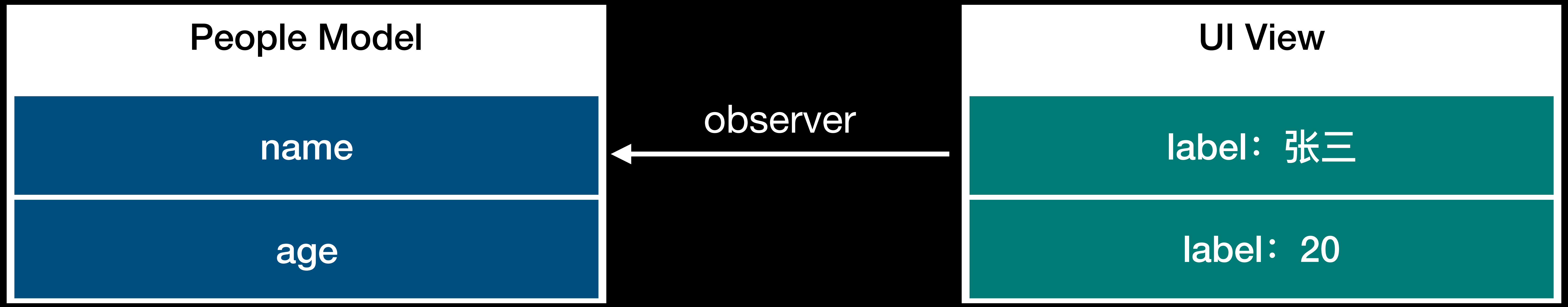
前两个方法无论获取值还是赋值，只需要传入属性名称的字符串就行了。但KVC也提供了传入path的方法。所谓path，就是用点号连接的多层级的属性，比如people.name，people属性里的name属性

通信方式 - KVC编程

```
- (void)kvcFunc0 {
    People *people = [self createPeople];
    [people setValue:@"李四" forKey:@"name"];
    NSLog(@"people name:%@", people.name); // people.name： 小王

    [people setValue:@"小黑" forKeyPath:@"dog.name"];
    NSLog(@"dog name:%@", people.dog.name); // dog name:小黑
}

- (void)kvcFunc1 {
    NSObject *people = [self createPeople]; // 抽象NSObject类型
    [people setValue:@"李四" forKey:@"name"];
    NSLog(@"people name:%@", [people valueForKey:@"name"]); // people.name： 小王

    [people setValue:@"小黑" forKeyPath:@"dog.name"];
    NSLog(@"dog name:%@", [people valueForKeyPath:@"dog.name"]); // dog name:小黑
}
```

通信方式 - 什么是KVO

Key-Value Obersver，即键值观察。它是观察者模式的一种衍生。基本思想是，对目标对象的某属性添加观察，当该属性发生变化时，会自动的通知观察者。这里所谓的通知是触发观察者对象实现的KVO的接口方法。



通信方式 - KVO

首先给目标对象的属性添加观察

```
- (void)addObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath options:
(NSKeyValueObservingOptions)options context:(nullable void *)context;
```

```
NSKeyValueObservingOptions:
    NSKeyValueObservingOptionNew // 提供更改前的值
    NSKeyValueObservingOptionOld // 提供更改后的值
    NSKeyValueObservingOptionInitial // 观察最初的值（在注册观察服务时会调用一次触发方法）
    NSKeyValueObservingOptionPrior // 分别在值修改前后触发方法（即一次修改有两次触发）
```

实现下面方法来接收通知，需要注意各个参数的含义

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:
(NSDictionary<NSString *,id> *)change context:(void *)context
```

最后要移除观察者（必须要移除，否则无法释放observer）

```
- (void)removeObserver:(NSObject *)observer forKeyPath:(NSString *)keyPath;
```

通信方式 - KVO编程

```
- (void)kvoFunc {
    self.people = [self createPeople];
    [self.people addObserver:self forKeyPath:@"name" options:NSKeyValueObservingOptionNew
context:nil];
    self.people.name = @"李四";
}
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:
(NSDictionary<NSString *,id> *)change context:(void *)context
{
    if(object == self.people && [keyPath isEqualToString:@"name"]) {
        NSLog(@"new:%@", change[@"new"]); // new:李四
    } else {
        [super observeValueForKeyPath:keyPath ofObject:object change:change
context:context];
    }
}
- (void)dealloc
{
    // 移除观察者
    [self.people removeObserver:self forKeyPath:@"name"];
}
```

通信方式 - KVO - Xcode上机练习

基于People和Dog对象，实现KVO方式通信

ByteDance

字节跳动