CS193P – 第三课

iPhone应用程序开发

自定义类
对象的生命周期
自动释放
属性

摘要

- 1A和1B课程在1月十三日周三晚上11：59
  - 登记的斯坦福学生可以发送任何问题到 cs193p@cs.stanford.edu
  - 尽快提交程序！操作说明在网站上...
    - 删掉bulid文件夹，Xcode不会做这件事

摘要

- 2A和2B课程在1月20日周三晚上11：59
  - 2A：继续介绍基本工具
    - 添加自定义类
    - 基本的内存管理
  - 2B：开始第一个iPhone程序
    - 讨论的问题1/14，周四
    - 作业包含许多练习

在校学生和iTunes U

- 讲座已经开始在iTunes U上发布
- 发布时间比去年长
- 开始讲座！！
  - 讲座后没有安排时间做练习

工作时间

- 保罗的工作时间: 周四 2-4, 在 B26B
- 大卫的工作时间: 周一4-6pm: 在360

今天的话题

- 解决1A和1B功课中的问题
- 创建自定义类
- 对象的生命周期
- 自动释放
- Objective-C特性

自定义类

设计阶段

- 创建一个类
  - person
- 确定父类
  - NSObject（在这里）
- 有什么样的属性
  - Name,age,是否能投票
- 执行什么样的功能？
  - 投票

定义一个类

一个公共的头文件和一个私有的实现文件

Header File

Implementation File

定义一个类

一个公共的头文件和一个私有的实现文件

在头文件中声明类

```
#import <Foundation/Foundation.h>
@interface Person: NSObject
{
    // 实例变量
    NSString *name;
    int age;
}
// 声明方法
- (NSString *)name;
- (void)setName:(NSString *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;
- (void)castBallot;
@end
```

定义一个类

一个公共的头文件和一个私有的实现文件

实现这个类

- 实现getter、setter方法
- 实现功能方法

类的实现

```
#import "Person.h“

@implementation Person

- (int)age {
return age;
}
- (void)setAge:(int)value {
age = value;
}

//... 其它方法

@end
```

调用自己的方法

```
#import "Person.h“

@implementation Person

- (BOOL)canLegallyVote {
}

- (void)castBallot {


}
@end
```

调用自己的方法

```
#import "Person.h“

@implementation Person

-(BOOL)canLegallyVote {
    return ([self age] >= 18);
}
- (void)castBallot {



}
@end
```

调用自己的方法

```
#import "Person.h“

@implementation Person

-(BOOL)canLegallyVote {
    return ([self age] >= 18);
}

-(void)castBallot {
    if ([self canLegallyVote]) {
    // 填入投票实现
    } else {
    NSLog (@“我没有权利投票!”);
    }
}
@end
```

父类方法

- 就像刚才看到的，使用“self”调用变量。
  - 像是Java和C++中的“this”
- 使用super请求调用父类方法

```
- (void)doSomething {
// 首先调用父类实现
[super doSomething];

// 然后实现我们的自定义行为
int foo = bar;
// ...
}
```

类的生命周期

对象的生命周期

- 对象的创建
- 内存管理
- 对象的销毁

类的创建

- 两步过程
  - 分配内存空间来存储对象
  - 初始化对象

+ alloc

  - 用来确定分配多少内存的静态方法

- init

  - 初始化变量，执行其它设置

类的创建=分配内存+初始化

```
Person *person = nil;

person = [[Person alloc] init];
```

执行你自己的-init方法

```
#import "Person.h"

@implementation Person

-(id)init {
 // 让父类首先初始化
if (self = [super init]) {
            age = 0;
            name = @“Bob”;

            // 做其它初始化...
            }

return self;
}

@end
```

多个初始化方法

- 类可以定义多个初始化方法

```
-(id)init;
-(id)initWithName:(NSString *)name;
-(id)initWithName:(NSString *)name age:(int)age;
```

- 简单的通常使用默认值调用复杂的初始化方法

```
-(id)init {
        return [initWithName:@“No Name”];
}
-(id)initWithName:(NSString *)name {
        return [self initWithName:name age:0];
}
```

完成一个对象的初始化

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@“Jimmy Jones”];
 [person setAge:32];

[person castBallot];
[person doSomethingElse];
```

完成一个对象的初始化

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@“Jimmy Jones”];
 [person setAge:32];

[person castBallot];
[person doSomethingElse];

// 当我们做完时我们怎么处理person对象?
```

内存管理

	分配内存	销毁对象
C	malloc	free
Objective-C	alloc	dealloc

- 方法调用必须平衡
  - 否则你的程序会内存泄露或是崩溃
- 无论怎样，你将从不直接调用-dealloc
  - 有个例外，我们等会儿看...

引用计数

- 每个对象都有个retain count
  - 有关NSObject的定义
  - 只要retain count 大于零,对象就是存在和有效的
- +alloc和-copy创建对象并使retain count 等于1
- -retain增加retain count
- -release减少retain count
- 当retain count 减少到零,对象就会被清理掉
  - -dealloc自动调用
  - 单向街,一旦你调用-dealloc方法就不会返回

匹配调用

```
Person *person = nil;
person = [[Person alloc] init];
[person setName:@“Jimmy Jones”];
[person setAge:32];
[person castBallot];
[person doSomethingElse];
// 当用完person时释放它

[person release]; // person 将在这里被销毁
```

引用计数在起作用

Person *person = [[Person alloc] init];

+alloc 使Retain count为1

[person retain];

-retain 使 Retain count增加到2

[person release];

-release 使 Retain count减少到1

[person release];

Retain count 减少到0,调用-dealloc 方法

关于已释放的对象

```
Person *person = [[Person alloc] init];
// ...

[person release]; // 对象被释放了
```

关于已释放的对象

```
Person *person = [[Person alloc] init];
// ...

[person release]; // 对象被释放了

[person doSomething]; // 崩溃了!
```

关于已释放的对象

```
Person *person = [[Person alloc] init];
// ...
[person release]; // 对象被释放了
```

关于已释放的对象

```
Person *person = [[Person alloc] init];
// ...
[person release]; // 对象被释放了
person = nil;
```

关于已释放的对象

```
Person *person = [[Person alloc] init];
// ...
[person release]; // 对象被释放了
person = nil;

[person doSomething]; // 没有反应
```

执行一个-dealloc方法

```
#import "Person.h"

@implementation Person

- (void)dealloc {
    // 需要做些清理
    // ...

    // 当我们做完后调用父类来清理
    [super dealloc];
}
@end
```

对象生命周期总结

- 对象刚创建时retain count为1
- retain count随-retain和-release增加减少
- 当retain count减少到0，对象自动调用dealloc方法
- 你从不在你的代码中直接调用dealloc方法
  - 除了调用[super dealloc]
  - 你只需处理alloc、copy、retain、release

对象所有权

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // 实例变量
    NSString *name; // Person 类拥有name属性
    int age;
}
// 方法声明
- (NSString *)name;
- (void)setName:(NSString *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;
- (void)castBallot;
@end
```

对象所有权

```
#import "Person.h“

@implementation Person
```

对象所有权

```
#import "Person.h“

@implementation Person

- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)newName {




}
@end
```

对象所有权

```
#import "Person.h“

@implementation Person

- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)newName {
    if (name != newName) {
    [name release];
    name = [newName retain];
    // name的retain count增加1
    }
}
@end
```

对象所有权

```
#import "Person.h“

@implementation Person

- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)newName {




}
@end
```

对象所有权

```
#import "Person.h“

@implementation Person

- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)newName {
    if (name != newName) {
    [name release];
    name = [newName copy];
    // name的retain count增加1
    }
}
@end
```

释放实例变量

```
#import "Person.h“

@implementation Person

-(void)dealloc {
    // 需要做些清理
    [name release];
    // 当我们做完时调用父类进行清理
    [super dealloc];
}
@end
```

自动释放

返回一个新创建的对象

```
- (NSString *)fullName {
    NSString *result;
    result = [[NSString alloc] initWithFormat:@“%@ %@”,       firstName,
    lastName];
    return result;
}
```

错误: 结果是内存泄露!

返回一个新创建的对象

```
- (NSString *)fullName {
    NSString *result;

    result = [[NSString alloc] initWithFormat:@“%@ %@”,
            firstName, lastName];
    [result release];
    return result;
}
```

错误: 结果是释放太早!

方法返回一个虚值

返回一个新创建的对象

```
- (NSString *)fullName {
    NSString *result;

    result = [[NSString alloc] initWithFormat:@“%@ %@”,
            firstName, lastName];
    [result autorelease];
    return result;
}
```

这就对了：结果被释放，担不是立刻

Caller获得真实的对象如果需要会retain它

自动释放对象

- 调用-autorelease标志着一个对象将在未来的某个时刻被释放掉
- 让你在需要时执行你的retain/release方法，在此期间允许对象延长一段生命
- 使内存管理变得更容易
- 在返回一个新的对象时很有用

方法名字和自动释放

- 名字为alloc、copy、或是返回一个已分配内存的对象的方法需要调用release方法

```
NSMutableString *string = [[NSMutableString alloc] init];
// 我们要负责调用-release或是-autorelease
[string autorelease];
```

- 所有的其它方法反回自动释放的对象

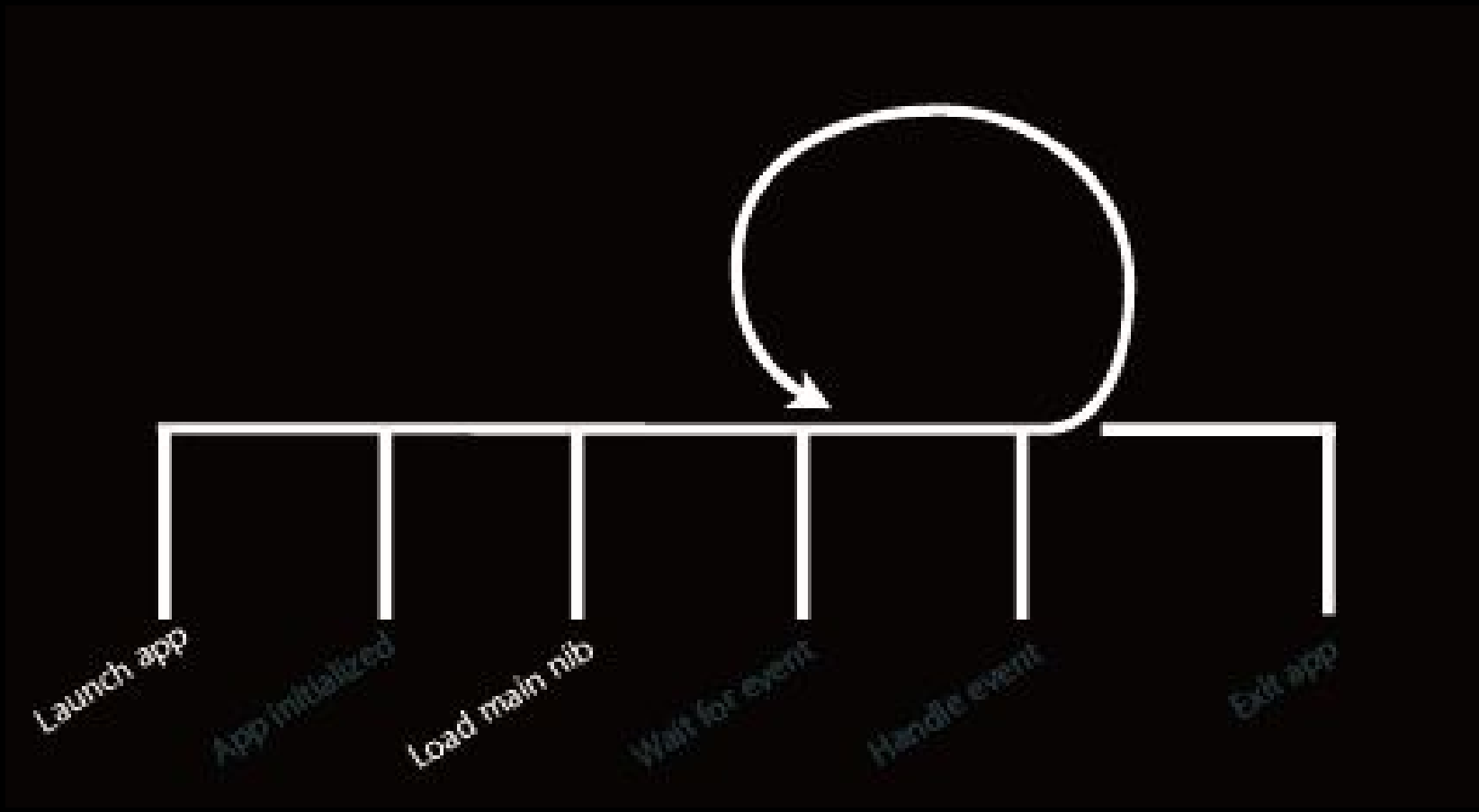
```
NSMutableString *string = [NSMutableString string];
// 方法名字没有指示我们去释放它
// 所以我们不用-没我们的事!
```

- 这是约定 –在你的方法中遵循它！

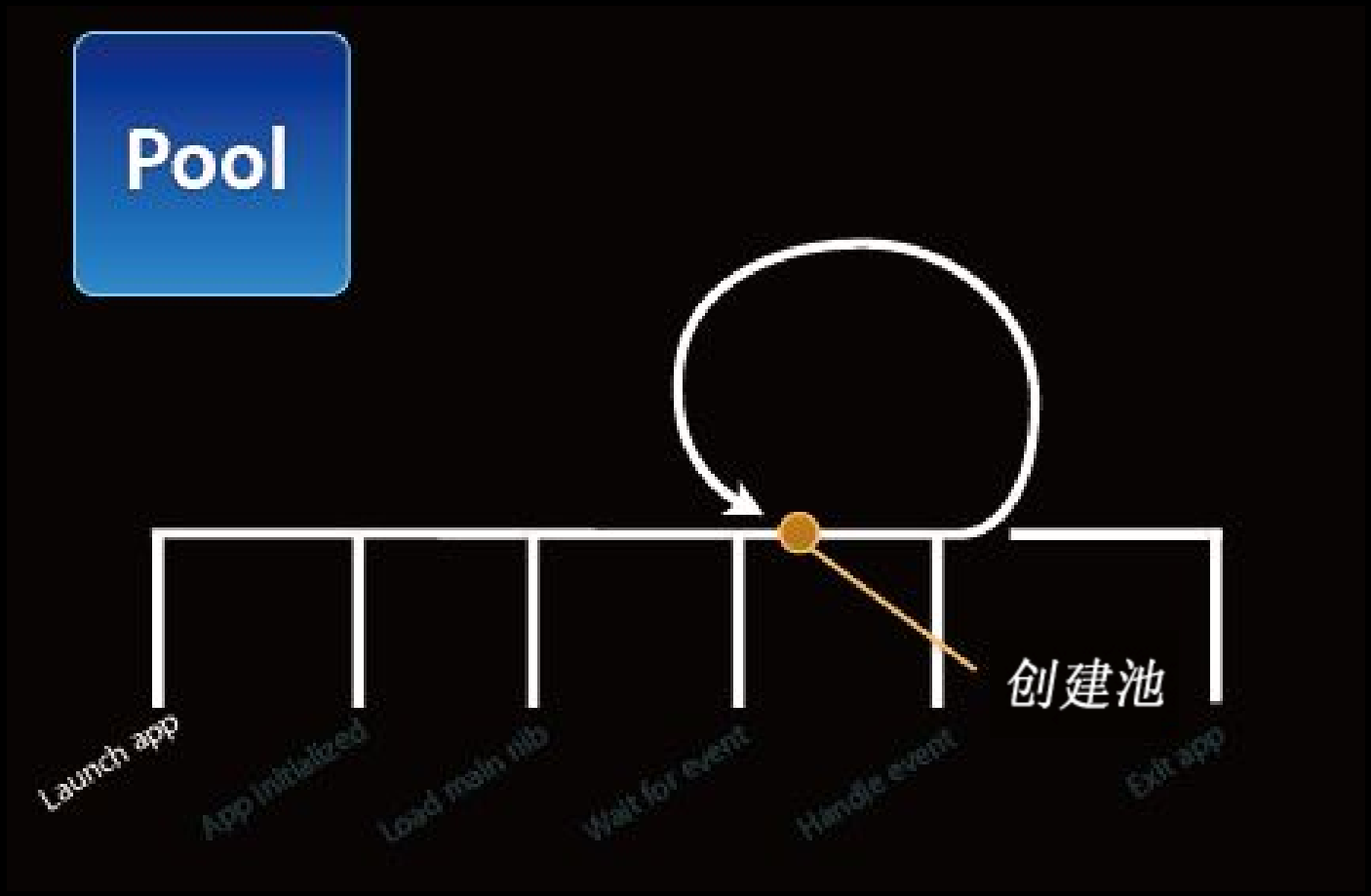
自动释放是如何工作的？

- 对象被加到自动释放池中
- 自动释放池跟踪安排要被释放的对象
  - 当自动释放池自己被释放时，它发送-release到它所有的对象
- UIKit框架自动打包了一个自动释放池存放所有的事件调用

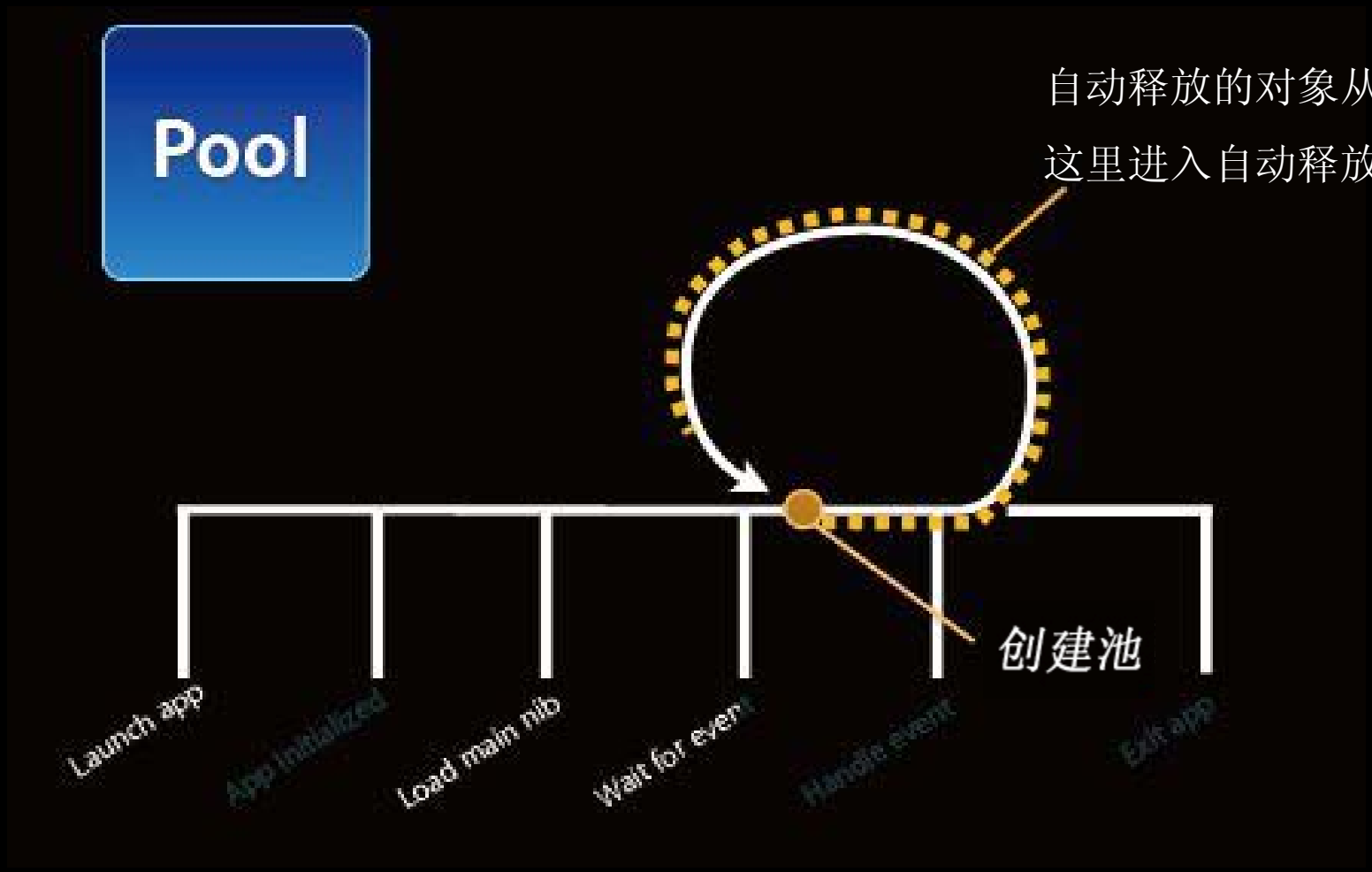
自动释放池


Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

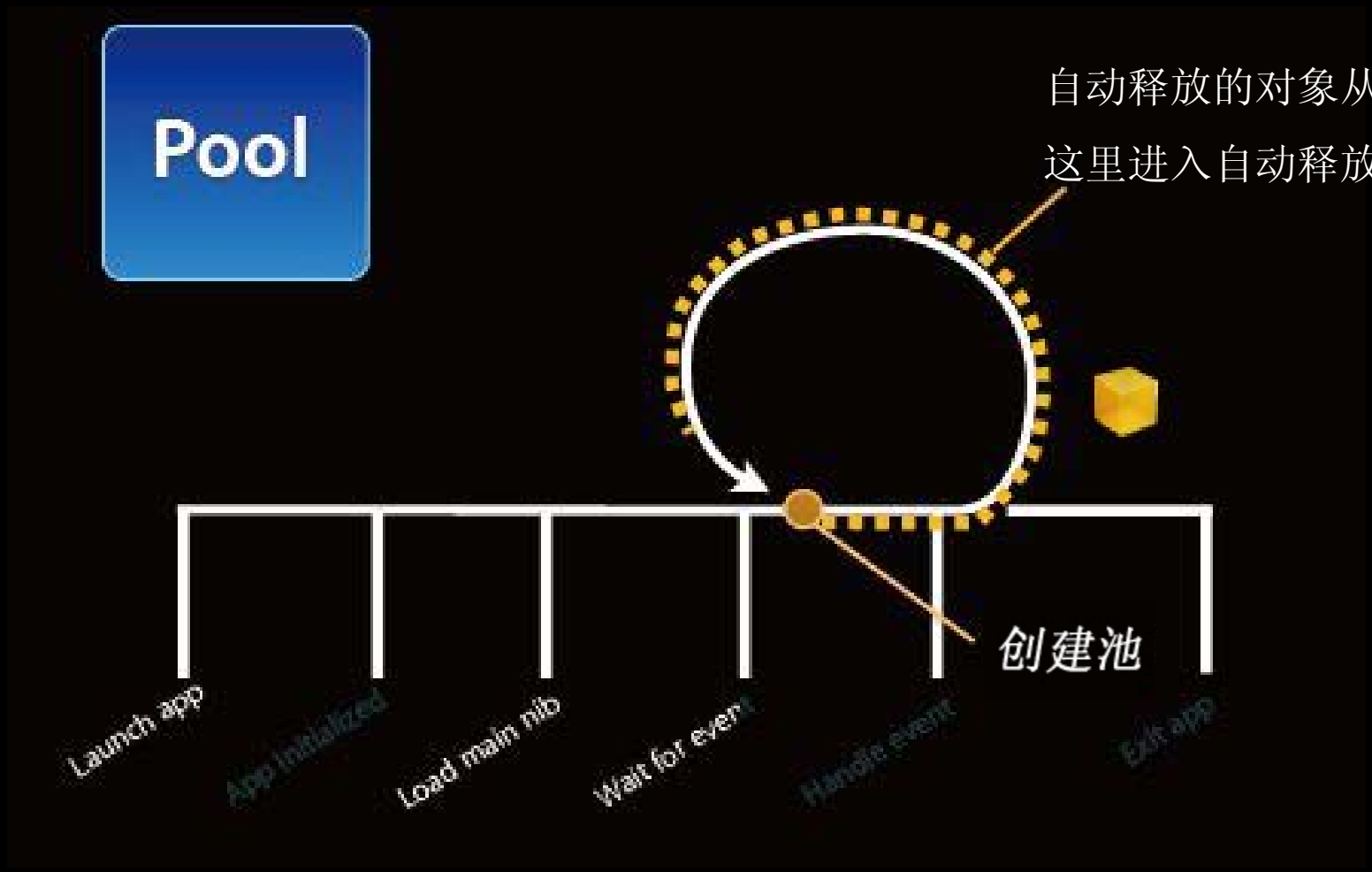
自动释放池


Pool
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

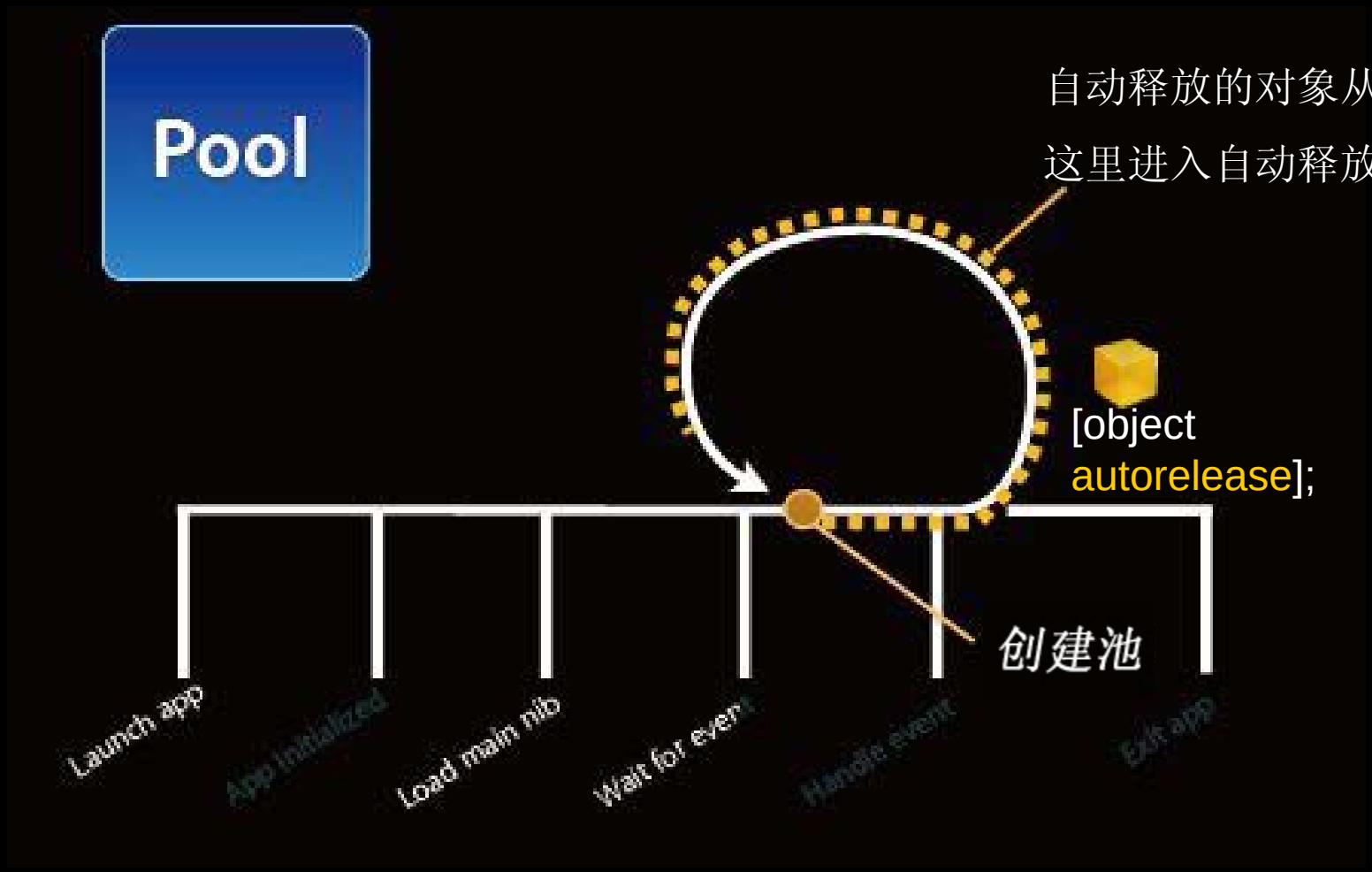
自动释放池


Pool
自动释放的对象从
这里进入自动释放池
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

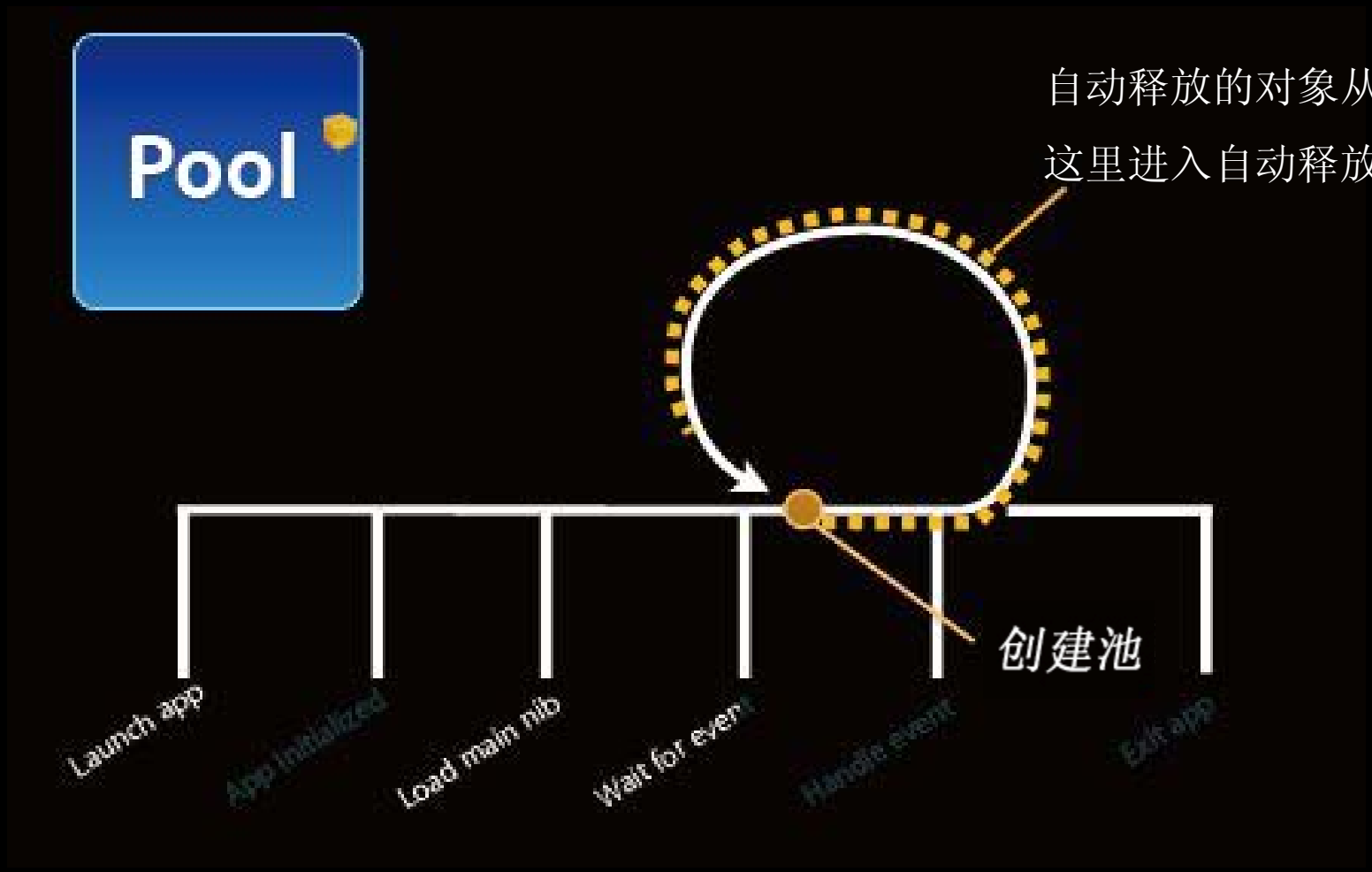
自动释放池


Pool
自动释放的对象从
这里进入自动释放池
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

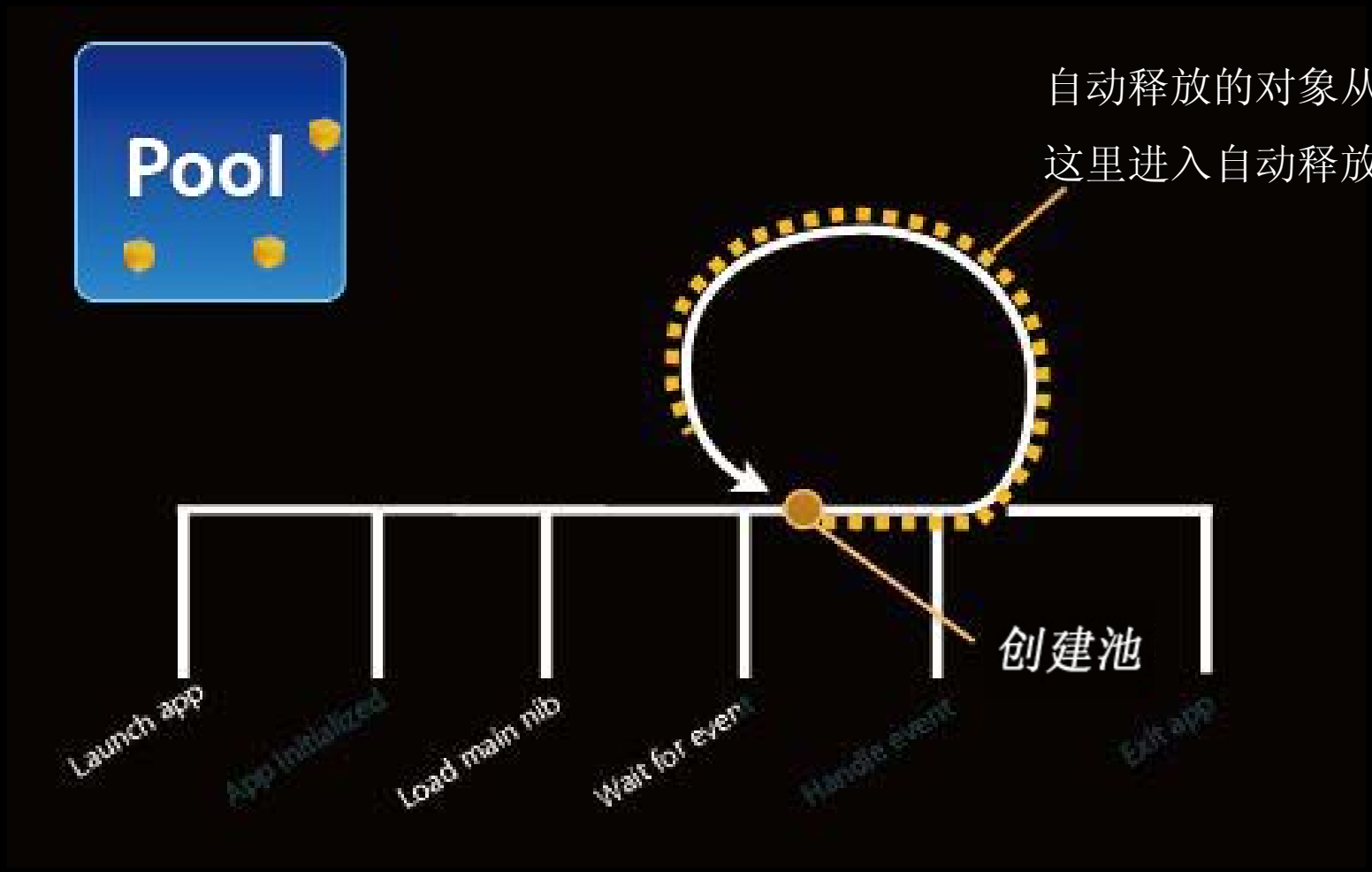
自动释放池


Pool
自动释放的对象从
这里进入自动释放池
[object
autorelease];
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

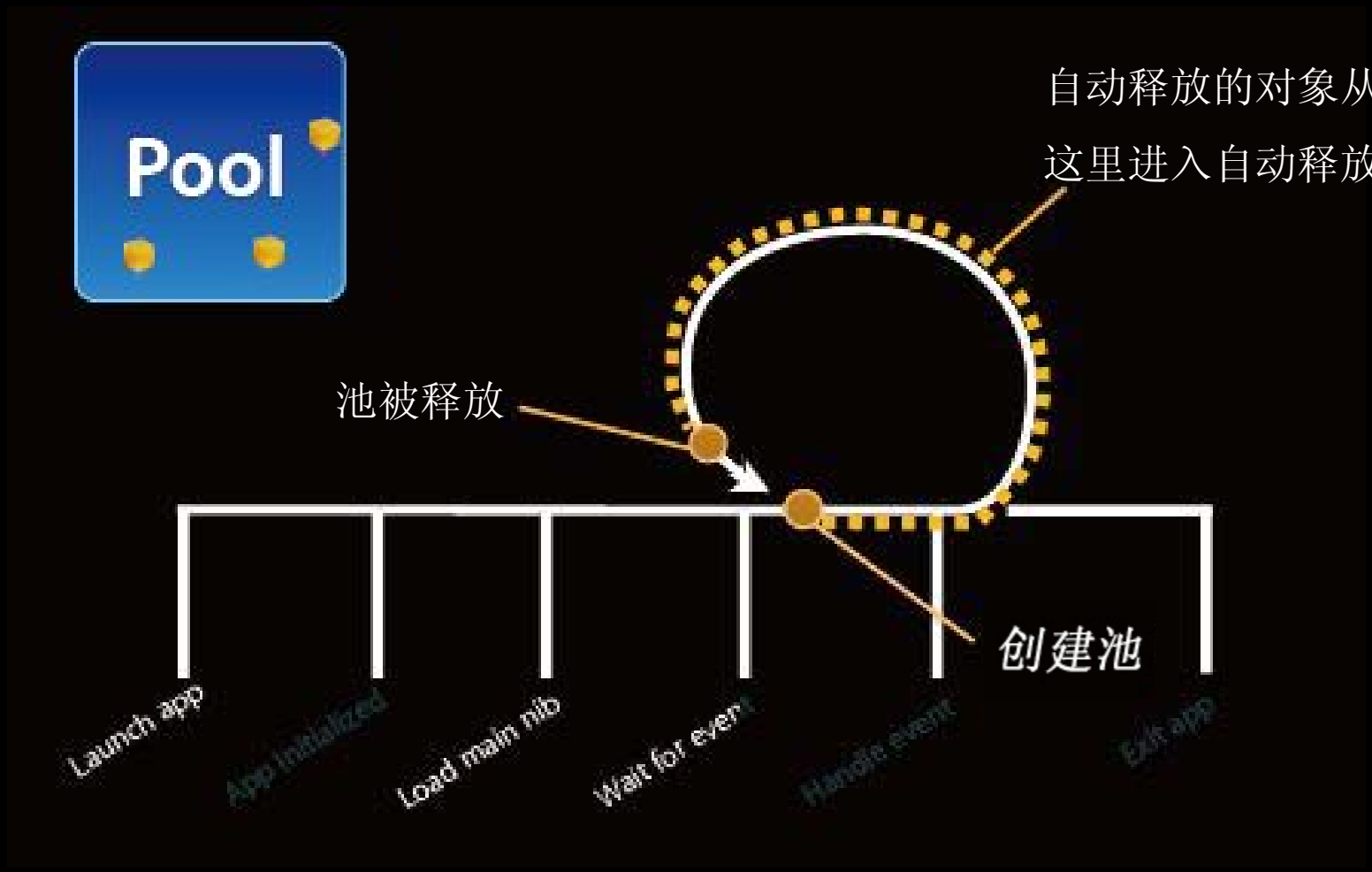
自动释放池


Pool
自动释放的对象从
这里进入自动释放池
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

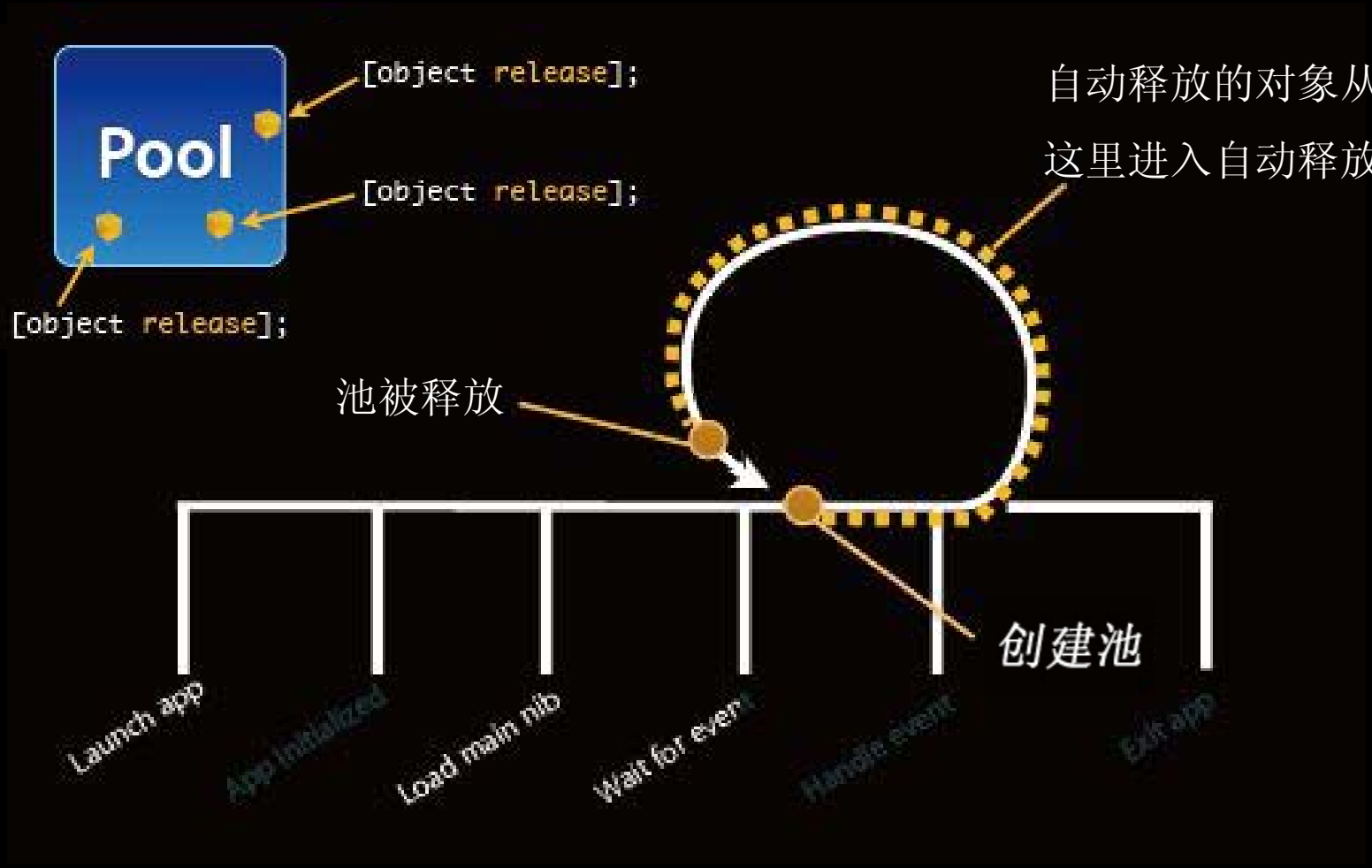
自动释放池


Pool
自动释放的对象从
这里进入自动释放池
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

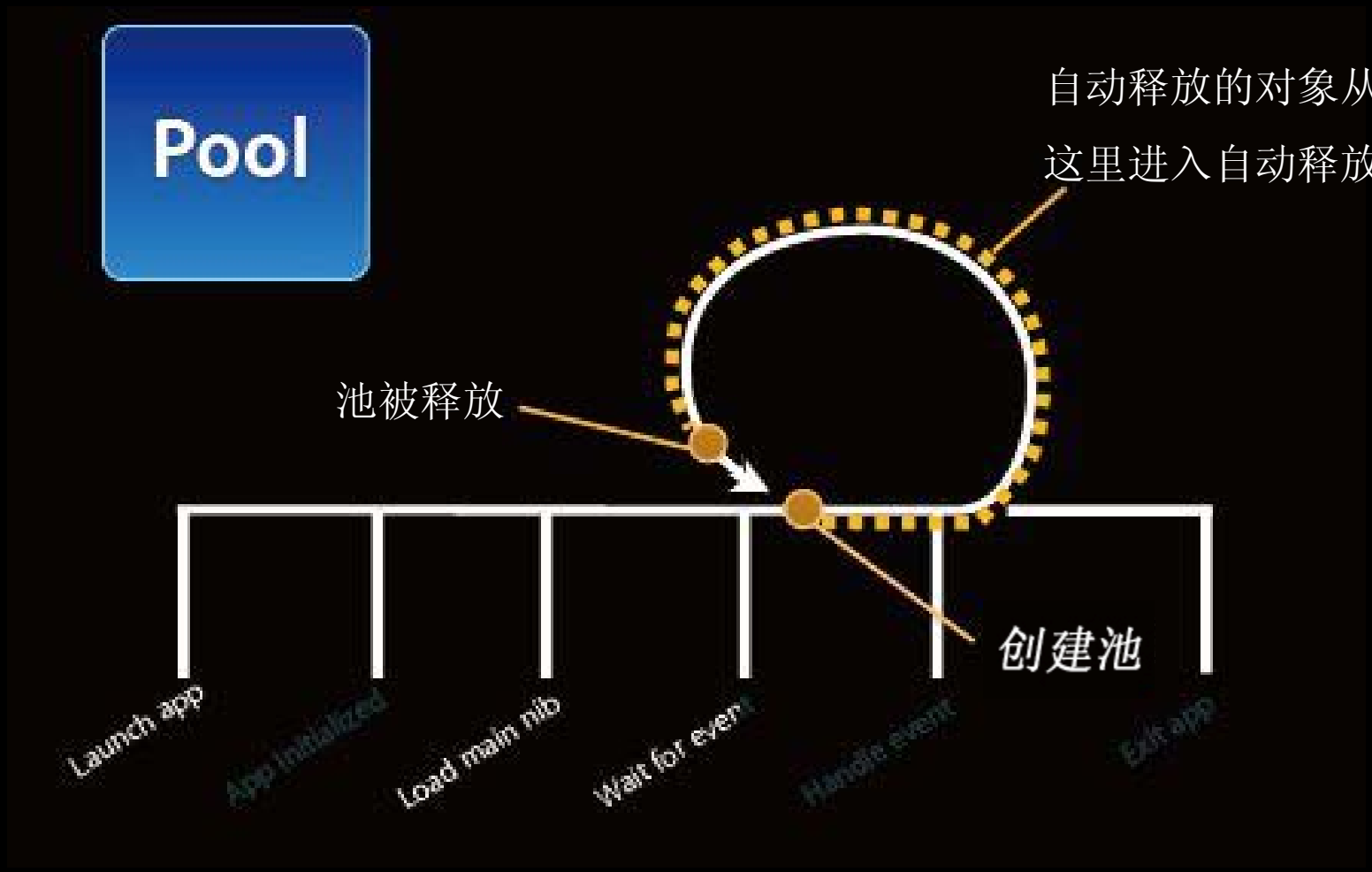
自动释放池


Pool
自动释放的对象从
这里进入自动释放池
池被释放
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

自动释放池


Pool
[object release];
[object release];
[object release];
自动释放的对象从
这里进入自动释放池
池被释放
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

自动释放池


Pool
自动释放的对象从
这里进入自动释放池
池被释放
创建池
Launch app
App initialized
Load main nib
Wait for event
Handle event
Exit app

抓住一个自动释放的对象

- 许多方法返回自动释放的对象
  - 记住命名规约
  - 它们被放到池中并且之后将被释放
- 如果你需要处理这些对象你需要retain它们
  - 在释放之前retain count突然增加

```
name = [NSMutableString string];
// 我们想保留name的有效值
[name retain];
// ...
// 最后,我们释放它 (或许在我们的-dealloc方法中？)
[name release];
```

边注：垃圾回收器

- 自动释放不是垃圾回收器
- iPhone OS 中的Objective-C没有垃圾回收器

Objective-C特性

特性

- 提供访问对象属性的方法
- 简化了 implementing中 getter/setter 方法
- 还允许您指定：
  - 只读与读写权限
  - 内存管理策略

定义特性

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // 实例变量
    NSString *name;
    int age;
}
// 方法声明
- (NSString *)name ;
- (void)setName:(NSString*)value;
- (int) age ;
- (void)setAge:(int)age;
- (BOOL) canLegallyVote

- (void)castBallot;
@end
```

定义特性

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // 实例变量
    NSString *name;
    int age;
}

// 方法声明
- (NSString *)name;
- (void)setName:(NSString *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;

- (void)castBallot;
@end
```

定义特性

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // 实例变量
    NSString *name;
    int age;
}
```

```
// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;
- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;
```

```
- (void)castBallot;

@end
```

定义特性

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // 实例变量
    NSString *name;
    int age;
}
```

```
// property declarations
@property int age;
@property (copy) NSString *name;
@property (readonly) BOOL canLegallyVote;
```

```
- (void)castBallot;
@end
```

定义特性

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // 实例变量
    NSString *name;
    int age;
}
// 特性声明

@property int age;
@property (copy) NSString *name;
@property (readonly) BOOL canLegallyVote;


- (void)castBallot;
@end
```

生成特性

```
@implementation Person
- (int)age {
    return age;
}
- (void)setAge:(int)value {
    age = value;
}
- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)value {
    if (value != name) {
    [name release];
    name = [value copy];
    }
}
- (void)canLegallyVote { ...
```

生成特性

@implementation Person

```
- (int)age {
    return age;
}
- (void)setAge:(int)value {
    age = value;
}
- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)value {
    if (value != name) {
        [name release];
        name = [value copy];
    }
}
```

- (void)canLegallyVote { ...

生成特性

@implementation Person

```
- (int)age {
    return age;
}
- (void)setAge:(int)value {
    age = value;
}
- (NSString *)name {
    return name;
}
- (void)setName:(NSString *)value {
    if (value != name) {
        [name release];
        name = [value copy];
    }
}
```

- (void)canLegallyVote { …

生成特性

```
@implementation Person
@synthesize age;
@synthesize name;
- (BOOL)canLegallyVote {
    return (age > 17);
}
@end
```

特性的属性

- 只读与读写权限

```
@property int age; // 默认读写权限
@property (readonly) BOOL canLegallyVote;
```

- 内存管理策略 (只对对象属性)

```
@property (assign) NSString *name; // 指针 assignment
@property (retain) NSString *name; // 保留
@property (copy) NSString *name; // 复制
```

属性名 vs. 实例变量

- 特性的名可以和实例变量的名称不相同

```
@interface Person : NSObject {
    int numberOfYearsOld;
}
@property int age;

@end
@implementation Person
@synthesize age = numberOfYearsOld;
@end
```

特性

- 生成与执行特性

```
@implementation Person
@synthesize age;
@synthesize name;
- (void)setAge:(int)value {
    age = value;
// 对有新值的age变量进行操作
}
@end
```

- 执行Setter 方法
- 执行Getter方法

属性的实际应用

- 新的API使用@property
- 旧的API使用 getter/setter 方法
- 属性在整个UIKit API中大量使用
  - Foundation API中没那么多
- 可以使用两种方法的一种
  - 属性意味着编写更少的代码，但有时它的神奇的效果不那么显而易见

点语法和self

- 当使用自定义方法，注意在类中定义的属性使用及其使用的点语法
- 引用属性和不加self.是不一样的

```
@interface Person : NSObject
{
    NSString *name;
}
@property (copy) NSString *name;
@end
@implementation Person
-(void)doSomething {
    name = @“Fred”; // accesses ivar directly!
    self.name = @“Fred”; // calls accessor method
}
```

点语法与常见的错误

什么时候会执行此代码？

```
@implementation Person
- (void)setAge:(int)newAge {
        self.age = newAge;
}
@end
```

这相当于：

```
@implementation Person
-(void)setAge:(int)newAge {
        [self setAge:newAge]; // 无限循环!
}
@end
```

延伸阅读

- Objective - C的2.0编程语言
  - “定义类”
  - “声明属性”
- 内存管理Cocoa编程指南

问题？