罗朝辉（飘飘白云）

关注C++，iOS，Android，3D 图形图像以及嵌入式系统开发。个人博客：http://kesalin.github.io(翻墙可见)

[深入浅出Cocoa]详解键值观察（KVO）及其实现机理

分类： iOS 开发 2012-11-17 17:23 16597人阅读 评论(6) 收藏 举报

目录(?)

[深入浅出Cocoa]详解键值观察（KVO）及其实现机理

罗朝辉 (http://blog.csdn.net/kesalin/)

一，前言

Objective-C 中的键(key)-值(value)观察(KVO)并不是什么新鲜事物，它来源于设计模式中的观察者模式，其基本思想就是：

> 一个目标对象管理所有依赖于它的观察者对象，并在它自身的状态改变时主动通知观察者对象。这个主动通知通常是通过调用各观察者对象所提供的接口方法来实现的。观察者模式较完美地将目标对象与观察者对象解耦。

在 Objective-C 中有两种使用键值观察的方式：手动或自动，此外还支持注册依赖键（即一个键依赖于其他键，其他键的变化也会作用到该键）。下面将一一讲述这些，并会深入 Objective-C 内部一窥键值观察是如何实现的。

本文源码下载：点此下载

二，运用键值观察

1，注册与解除注册

如果我们已经有了包含可供键值观察属性的类，那么就可以通过在该类的对象（被观察对象）上调用名为 NSKeyValueObserver**Registration** 的 category 方法将观察者对象与被观察者对象注册与解除注册：

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

这两个方法的定义在 Foundation/NSKeyValueObserving.h 中，NSObject，NSArray，NSSet均实现了以上方法，因此我们不仅可以观察普通对象，还可以观察数组或结合类对象。在该头文件中，我们还可以看到 NSObject 还实现了 NSKeyValueObserver**Notification** 的 category 方法（更多类似方法，请查看该头文件）：

```
- (void)willChangeValueForKey:(NSString *)key;
- (void)didChangeValueForKey:(NSString *)key;
```

这两个方法在手动实现键值观察时会用到，暂且不提。

值得注意的是：不要忘记解除注册，否则会导致资源泄露。

2，设置属性

Android (8)
3D技术 (21)
移动开发 (42)
OpenGL ES (9)
算法 (9)
Windows (33)
游戏开发 (6)
Java (7)
.NET (2)
游戏人生 (4)
读书新知 (5)
开发环境 (16)
Mac 技巧 (5)
医疗开发 (3)
网络技术 (8)
脚本语言 (4)
软件工程 (6)

文章存档

2014年10月 (2)
2014年07月 (5)
2014年06月 (2)
2014年04月 (4)
2014年03月 (1)
展开

最新评论

XCode 下的 iOS 单元测试
赵茼: 已收藏，还没看懂
Android多线程分析之三：Handle
飘飘白云: 回复flysongpeng: callback 适用于简单的场合，只会在当前投递消息的线程中处理消息。...
Android多线程分析之三：Handle
天天黑芝麻糊: 那设置 Callback 回调和子类化 Handler，这两种处理消息的方式分别适用于哪些场合呢?
[OpenGL ES 03]3D变换：模型，
xiwrong: 非常非常系统 多谢
[深入浅出Cocoa]详解键值观察（
DowneyJr: 非常不错的借鉴方案,用到的是非ARC,而且,最重要的是诠释了set和get方法中怎么实现观察者
[OpenGL ES 03]3D变换：模型，
wqlineky1: 好东西，顶起来
[Cocoa]深入浅出 Cocoa 之 Core
huangxiongbiao1: 赞一个 但是我能问一下 用caredata 创建的model类 怎么样才能使用new方式...
[OpenGL ES 01]OpenGL ES之初
飘飘白云: @u011240067:我那时候使用的是xcode 4.2,你根据最新的 xcode 做相应的调整就...
[OpenGL ES 01]OpenGL ES之初
想象的产儿: 博主，我最近在看ios的EGL，搜到你的文章，在新的xcode下，那几个图的操作，有明显的不一致
在遍历中使用 iterator/reverse_ite
zoupingyuan: 呵呵 ，百度搜reverse_iterator 的erase ， 第一个是你啊

阅读排行

[Cocoa]深入浅出 Cocoa (69251)
使用System.arraycopy()和 (32721)

将观察者与被观察者注册好之后，就可以对观察者对象的属性进行操作，这些变更操作就会被通知给观察者对象。注意，只有遵循 KVO 方式来设置属性，观察者对象才会获取通知，也就是说遵循使用属性的 setter 方法，或通过 key-path 来设置：

```
[target setAge:30];
[target setValue:[NSNumber numberWithInt:30] forKey:@"age"];
```

3，处理变更通知

观察者需要实现名为 NSKeyValueObserving 的 category 方法来处理收到的变更通知：

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

在这里，change 这个字典保存了变更信息，具体是哪些信息取决于注册时的 NSKeyValueObservingOptions。

4，下面来看看一个完整的使用示例：

观察者类：

```
// Observer.h
@interface Observer : NSObject
@end

// Observer.m
#import "Observer.h"
#import <objc/runtime.h>
#import "Target.h"

@implementation Observer

- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context
{
    if ([keyPath isEqualToString:@"age"])
    {
        Class classInfo = (Class)context;
        NSString * className = [NSString stringWithCString:object_getClassNam
e(classInfo)
                                                   encoding:NSUTF8StringEncodin
g];
        NSLog(@" >> class: %@, Age changed", className);

        NSLog(@" old age is %@", [change objectForKey:@"old"]);
        NSLog(@" new age is %@", [change objectForKey:@"new"]);
    }
    else
    {
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}
```

```
@end
```

注意：在实现处理变更通知方法 observeValueForKeyPath 时，要将不能处理的 key 转发给 super 的 observeValueForKeyPath 来处理。

使用示例：

```
Observer * observer = [[[Observer alloc] init] autorelease];

Target * target = [[[Target alloc] init] autorelease];
[target addObserver:observer
         forKeyPath:@"age"
            options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
            context:[Target class]];

[target setAge:30];
//[target setValue:[NSNumber numberWithInt:30] forKey:@"age"];

[target removeObserver:observer forKeyPath:@"age"];
```

在这里 observer 观察 target 的 age 属性变化，运行结果如下：

```
>> class: Target, Age changed
old age is 10
new age is 30
```

三，手动实现键值观察

上面的 Target 应该怎么实现呢？首先来看手动实现。

```
@interface Target : NSObject
{
    int age;
}

// for manual KVO - age
- (int) age;
- (void) setAge:(int)theAge;

@end

@implementation Target

- (id) init
{
    self = [super init];
    if (nil != self)
    {
        age = 10;
    }

    return self;
```

```
}

// for manual KVO - age
- (int) age
{
    return age;
}

- (void) setAge:(int)theAge
{
    [self willChangeValueForKey:@"age"];
    age = theAge;
    [self didChangeValueForKey:@"age"];
}

+ (BOOL) automaticallyNotifiesObserversForKey:(NSString *)key {
    if ([key isEqualToString:@"age"]) {
        return NO;
    }

    return [super automaticallyNotifiesObserversForKey:key];
}

@end
```

首先，需要手动实现属性的 setter 方法，并在设置操作的前后分别调用 **willChangeValueForKey:** 和 **didChangeValueForKey**方法，这两个方法用于通知系统该 key 的属性值即将和已经变更了；

其次，要实现类方法 **automaticallyNotifiesObserversForKey**，并在其中设置对该 key 不自动发送通知（返回 NO 即可）。这里要注意，对其它非手动实现的 key，要转交给 super 来处理。

四，自动实现键值观察

自动实现键值观察就非常简单了，只要使用了自动属性即可。

```
@interface Target : NSObject
// for automatic KVO - age
@property (nonatomic, readwrite) int age;
@end

@implementation Target
@synthesize age; // for automatic KVO - age

- (id) init
{
    self = [super init];
    if (nil != self)
    {
        age = 10;
    }

    return self;
}
```

```
@end
```

五，键值观察依赖键

有时候一个属性的值依赖于另一对象中的一个或多个属性，如果这些属性中任一属性的值发生变更，被依赖的属性值也应当为其变更进行标记。因此，object 引入了依赖键。

1，观察依赖键

观察依赖键的方式与前面描述的一样，下面先在 Observer 的 observeValueForKeyPath:ofObject:change:context: 中添加处理变更通知的代码：

```
#import "TargetWrapper.h"

- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context
{
    if ([keyPath isEqualToString:@"age"])
    {
        Class classInfo = (Class)context;
        NSString * className = [NSString stringWithCString:object_getClassNam
e(classInfo)
                                                   encoding:NSUTF8StringEncodin
g];
        NSLog(@" >> class: %@, Age changed", className);

        NSLog(@" old age is %@", [change objectForKey:@"old"]);
        NSLog(@" new age is %@", [change objectForKey:@"new"]);
    }
    else if ([keyPath isEqualToString:@"information"])
    {
        Class classInfo = (Class)context;
        NSString * className = [NSString stringWithCString:object_getClassNam
e(classInfo)
                                                   encoding:NSUTF8StringEncodin
g];
        NSLog(@" >> class: %@, Information changed", className);
        NSLog(@" old information is %@", [change objectForKey:@"old"]);
        NSLog(@" new information is %@", [change objectForKey:@"new"]);
    }
    else
    {
        [super observeValueForKeyPath:keyPath
                             ofObject:object
                               change:change
                              context:context];
    }
}
```

2，实现依赖键

在这里，观察的是 TargetWrapper 类的 information 属性，该属性是依赖于 Target 类的 age 和 grade 属性。为此，我在 Target 中添加了 grade 属性：

```
@interface Target : NSObject
@property (nonatomic, readwrite) int grade;
@property (nonatomic, readwrite) int age;
@end

@implementation Target
@synthesize age; // for automatic KVO - age
@synthesize grade;
@end
```

下面来看看 TragetWrapper 中的依赖键属性是如何实现的。

```
@class Target;

@interface TargetWrapper : NSObject
{
@private
    Target * _target;
}

@property(nonatomic, assign) NSString * information;
@property(nonatomic, retain) Target * target;

-(id) init:(Target *)aTarget;

@end

#import "TargetWrapper.h"
#import "Target.h"

@implementation TargetWrapper

@synthesize target = _target;

-(id) init:(Target *)aTarget
{
    self = [super init];
    if (nil != self) {
        _target = [aTarget retain];
    }

    return self;
}

-(void) dealloc
{
    self.target = nil;
    [super dealloc];
}

- (NSString *)information
{
    return [[[NSString alloc] initWithFormat:@"%d#%d", [_target grade], [_targ
et age]] autorelease];
```

```
}

- (void)setInformation:(NSString *)theInformation
{
    NSArray * array = [theInformation componentsSeparatedByString:@"#"];
    [_target setGrade:[[array objectAtIndex:0] intValue]];
    [_target setAge:[[array objectAtIndex:1] intValue]];
}

+ (NSSet *)keyPathsForValuesAffectingInformation
{
    NSSet * keyPaths = [NSSet setWithObjects:@"target.age", @"target.grade", n
il];
    return keyPaths;
}

//+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key
//{
//    NSSet * keyPaths = [super keyPathsForValuesAffectingValueForKey:key];
//    NSArray * moreKeyPaths = nil;
//
//    if ([key isEqualToString:@"information"])
//    {
//        moreKeyPaths = [NSArray arrayWithObjects:@"target.age", @"target.gra
de", nil];
//    }
//
//    if (moreKeyPaths)
//    {
//        keyPaths = [keyPaths setByAddingObjectsFromArray:moreKeyPaths];
//    }
//
//    return keyPaths;
//}

@end
```

首先，要手动实现属性 information 的 setter/getter 方法，在其中使用 Target 的属性来完成其 setter 和 getter。

其次，要实现 **keyPathsForValuesAffectingInformation** 或 **keyPathsForValuesAffectingValueForKey**: 方法来告诉系统 information 属性依赖于哪些其他属性，这两个方法都返回一个key-path 的集合。在这里要多说几句，如果选择实现 **keyPathsForValuesAffectingValueForKey**，要先获取 super 返回的结果 set，然后判断 key 是不是目标 key，如果是就将依赖属性的 key-path 结合追加到 super 返回的结果 set 中，否则直接返回 super的结果。

在这里，information 属性依赖于 target 的 age 和 grade 属性，target 的 age/grade 属性任一发生变化，information 的观察者都会得到通知。

3，使用示例：

```
Observer * observer = [[[Observer alloc] init] autorelease];
Target * target = [[[Target alloc] init] autorelease];

TargetWrapper * wrapper = [[[TargetWrapper alloc] init:target] autorelease];
[wrapper addObserver:observer
          forKeyPath:@"information"
```

```
                options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionO
ld
                context:[TargetWrapper class]];

[target setAge:30];
[target setGrade:1];
[wrapper removeObserver:observer forKeyPath:@"information"];
```

输出结果：

```
>> class: TargetWrapper, Information changed
old information is 0#10
new information is 0#30
>> class: TargetWrapper, Information changed
old information is 0#30
new information is 1#30
```

六，键值观察是如何实现的

1，实现机理

键值观察用处很多，Core Binding 背后的实现就有它的身影，那键值观察背后的实现又如何呢？想一想在上面的自动实现方式中，我们并不需要在被观察对象 Target 中添加额外的代码，就能获得键值观察的功能，这很好很强大，这是怎么做到的呢？答案就是 Objective C 强大的 runtime 动态能力，下面我们一起来窥探下其内部实现过程。

当某个类的对象第一次被观察时，系统就会在运行期动态地创建该类的一个派生类，在这个派生类中重写基类中任何被观察属性的 setter 方法。

派生类在被重写的 setter 方法实现真正的通知机制，就如前面手动实现键值观察那样。这么做是基于设置属性会调用 setter 方法，而通过重写就获得了 KVO 需要的通知机制。当然前提是要通过遵循 KVO 的属性设置方式来变更属性值，如果仅是直接修改属性对应的成员变量，是无法实现 KVO 的。

同时派生类还重写了 class 方法以“欺骗”外部调用者它就是起初的那个类。然后系统将这个对象的 isa 指针指向这个新诞生的派生类，因此这个对象就成为该派生类的对象了，因而在该对象上对 setter 的调用就会调用重写的 setter，从而激活键值通知机制。此外，派生类还重写了 dealloc 方法来释放资源。

如果你对类和对象的关系不太明白，请阅读《深入浅出Cocoa之类与对象》；如果你对如何动态创建类不太明白，请阅读《深入浅出Cocoa 之动态创建类》。

苹果官方文档说得很简洁：

> **Key-Value Observing Implementation Details**
>
> Automatic key-value observing is implemented using a technique called **isa-swizzling**.
>
> The `isa` pointer, as the name suggests, points to the object's class which maintains a dispatch table. This dispatch table essentially contains pointers to the methods the class implements, among other data.
>
> When an observer is registered for an attribute of an object the isa pointer of the observed object is modified, pointing to an intermediate class rather than at the true class. As a result the value of the isa pointer does not necessarily reflect the actual class of the instance.
>
> You should never rely on the `isa` pointer to determine class membership. Instead, you should use the `class` method to determine the class of an object instance.

2，代码分析

由于派生类中被重写的 class 对我们撒谎（它说它就是起初的基类），我们只有通过调用 runtime 函数才能揭开派生类的真面目。 下面来看 Mike Ash 的代码：

首先是带有 x, y, z 三个属性的观察目标 Foo:

```
@interface Foo : NSObject
{
    int x;
    int y;
    int z;
}

@property int x;
@property int y;
@property int z;

@end

@implementation Foo
@synthesize x, y, z;
@end
```

下面是检验代码:

```
#import <objc/runtime.h>

static NSArray * ClassMethodNames(Class c)
{
    NSMutableArray * array = [NSMutableArray array];

    unsigned int methodCount = 0;
    Method * methodList = class_copyMethodList(c, &methodCount);
    unsigned int i;
    for(i = 0; i < methodCount; i++) {
        [array addObject: NSStringFromSelector(method_getName(methodLis
t[i]))];
    }

    free(methodList);

    return array;
}

static void PrintDescription(NSString * name, id obj)
{
    NSString * str = [NSString stringWithFormat:
                      @"\n\t%@: %@\n\tNSObject class %s\n\tlibobjc class
%s\n\timplements methods <%@>",
                      name,
                      obj,
                      class_getName([obj class]),
                      class_getName(obj->isa),
                      [ClassMethodNames(obj->isa) componentsJoinedByString:@",
"]];
    NSLog(@"%@", str);
}
```

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        // Deep into KVO: kesalin@gmail.com
        //
        Foo * anything = [[Foo alloc] init];
        Foo * x = [[Foo alloc] init];
        Foo * y = [[Foo alloc] init];
        Foo * xy = [[Foo alloc] init];
        Foo * control = [[Foo alloc] init];

        [x addObserver:anything forKeyPath:@"x" options:0 context:NULL];
        [y addObserver:anything forKeyPath:@"y" options:0 context:NULL];

        [xy addObserver:anything forKeyPath:@"x" options:0 context:NULL];
        [xy addObserver:anything forKeyPath:@"y" options:0 context:NULL];

        PrintDescription(@"control", control);
        PrintDescription(@"x", x);
        PrintDescription(@"y", y);
        PrintDescription(@"xy", xy);

        NSLog(@"\n\tUsing NSObject methods, normal setX: is %p, overridden setX: is %p\n",
              [control methodForSelector:@selector(setX:)],
              [x methodForSelector:@selector(setX:)]);
        NSLog(@"\n\tUsing libobjc functions, normal setX: is %p, overridden setX: is %p\n",
              method_getImplementation(class_getInstanceMethod(object_getClass(control),
                                                               @selector(setX:))),
              method_getImplementation(class_getInstanceMethod(object_getClass(x),
                                                               @selector(setX:))));
    }

    return 0;
}
```

在上面的代码中，辅助函数 ClassMethodNames 使用 runtime 函数来获取类的方法列表，PrintDescription 打印对象的信息，包括通过 -class 获取的类名， isa 指针指向的类的名字以及其中方法列表。

在这里，我创建了四个对象，x 对象的 x 属性被观察，y 对象的 y 属性被观察，xy 对象的 x 和 y 属性均被观察，参照对象 control 没有属性被观察。在代码的最后部分，分别通过两种方式(对象方法和 runtime 方法)打印出参数对象 control 和被观察对象 x 对象的 setX 方面的实现地址，来对比显示正常情况下 setter 实现以及派生类中重写的 setter 实现。

编译运行，输出如下：

```
control: <Foo: 0x10010c980>

NSObject class Foo

libobjc class Foo

implements methods <x, setX:, y, setY:, z, setZ:>
```

```
x: <Foo: 0x10010c920>

NSObject class Foo

libobjc class NSKVONotifying_Foo

implements methods <setY:, setX:, class, dealloc, _isKVOA>

y: <Foo: 0x10010c940>

NSObject class Foo

libobjc class NSKVONotifying_Foo

implements methods <setY:, setX:, class, dealloc, _isKVOA>

xy: <Foo: 0x10010c960>

NSObject class Foo

libobjc class NSKVONotifying_Foo

implements methods <setY:, setX:, class, dealloc, _isKVOA>

Using NSObject methods, normal setX: is 0x100001df0, overridden setX: is 0x100001df0

Using libobjc functions, normal setX: is 0x100001df0, overridden setX: is 0x7fff8458e025
```

从上面的输出可以看到，如果使用对象的 -class 方面输出类名始终为：Foo，这是因为新诞生的派生类重写了 -class 方法声称它就是起初的基类，只有使用 runtime 函数 object_getClass 才能一睹芳容：NSKVONotifying_Foo。注意看：x，y 以及 xy 三个被观察对象真正的类型都是 NSKVONotifying_Foo，而且该类实现了：setY:, setX:, class, dealloc, _isKVOA 这些方法。其中 setX:, setY:, class 和 dealloc 前面已经讲到过，私有方法 _isKVOA 估计是用来标示该类是一个 KVO 机制声称的类。在这里 Objective C 做了一些优化，它对所有被观察对象只生成一个派生类，该派生类实现所有被观察对象的 setter 方法，这样就减少了派生类的数量，提供了效率。所有 NSKVONotifying_Foo 这个派生类重写了 setX，setY方法（留意:没有必要重写 setZ 方法）。

接着来看最后两行输出，地址 0x100001df0 是 Foo 类中的实现，而地址是 0x7fff8458e025 是派生类 NSKVONotifying_Foo 类中的实现。那后面那个地址到底是什么呢？可以通过 GDB 的 info 命令加 symbol 参数来查看该地址的信息：

```
(gdb) info symbol 0x7fff8458e025

_NSSetIntValueAndNotify in section LC_SEGMENT.__TEXT.__text of /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
```

看起来它是 Foundation 框架提供的私有函数：_NSSetIntValueAndNotify。更进一步，我们来看看 Foundation 到底提供了哪些用于 KVO 的辅助函数。打开 terminal，使用 nm -a 命令查看 Foundation 中的信息：

```
nm -a /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
```

其中查找到我们关注的函数：

```
00000000000233e7 t __NSSetDoubleValueAndNotify
00000000000f32ba t __NSSetFloatValueAndNotify
0000000000025025 t __NSSetIntValueAndNotify
000000000007fbb5 t __NSSetLongLongValueAndNotify
00000000000f33e8 t __NSSetLongValueAndNotify
000000000002d36c t __NSSetObjectValueAndNotify
0000000000024dc5 t __NSSetPointValueAndNotify
00000000000f39ba t __NSSetRangeValueAndNotify
00000000000f3aeb t __NSSetRectValueAndNotify
00000000000f3512 t __NSSetShortValueAndNotify
```

```
00000000000f3c2f t __NSSetSizeValueAndNotify
00000000000f363b t __NSSetUnsignedCharValueAndNotify
000000000006e91f t __NSSetUnsignedIntValueAndNotify
0000000000034b5b t __NSSetUnsignedLongLongValueAndNotify
00000000000f3766 t __NSSetUnsignedLongValueAndNotify
00000000000f3890 t __NSSetUnsignedShortValueAndNotify
00000000000f3060 t __NSSetValueAndNotifyForKeyInIvar
00000000000f30d7 t __NSSetValueAndNotifyForUndefinedKey
```

Foundation 提供了大部分基础数据类型的辅助函数（Objective C中的 Boolean 只是 unsigned char 的 typedef，所以包括了，但没有 C++中的 bool），此外还包括一些常见的 Cocoa 结构体如 Point, Range, Rect, Size，这表明这些结构体也可以用于自动键值观察，但要注意除此之外的结构体就不能用于自动键值观察了。对于所有 Objective C 对象对应的是 __NSSetObjectValueAndNotify 方法。

七，总结

KVO 并不是什么新事物，换汤不换药，它只是观察者模式在 Objective C 中的一种运用，这是 KVO 的指导思想所在。其他语言实现中也有"KVO"，如 WPF 中的 binding。而在 Objective C 中又是通过强大的 runtime 来实现自动键值观察的。至此，对 KVO 的使用以及注意事项，内部实现都介绍完毕，对 KVO 的理解又深入一层了。Objective 中的 KVO 虽然可以用，但却非完美，有兴趣的了解朋友请查看《KVO 的缺陷》 以及改良实现 MAKVONotificationCenter 。

八，引用

Key-value observing：官方文档

Key-Value Observing Done Right：官方 KVO 实现的缺陷

MAKVONotificationCenter：一个改良的 Notification 实现，托管在 GitHub 上

Friday Q&A 2009-01-23

深入浅出Cocoa 之动态创建类

深入浅出Cocoa之类与对象

上一篇 [深入浅出Cocoa]之消息（二）-详解动态方法决议(Dynamic Method Resolution)

下一篇 [OpenGL ES 01]OpenGL ES之初体验