```
0x10f416068, 0x80d0026d2df230be, 0x6000022b2200
__NSCFString, __NSCFString, __NSCFString
0x600002cad2f0, 0x600002cad650, 0x600002cad170
HelloHi
```

从打印信息可以看到，对 NSString 对象调用 copy 方法后，其类型和地址都没有任何变化，调用 mutableCopy 方法后，从对象地址可以看出全部创建了新的对象，新创建出的对象一律为 __NSCFString 类型。

对于 NSMutableString 类型的对象，无论调用 copy 方法还是 mutableCopy 方法，其都会进行深考虑，不同的是，调用 copy 方法会重新创建出一个不可变 NSString 对象，使用 mutableCopy 会重新创建出一个可变的 NSMutableString 对象，示例代码如下：

```
NSMutableString *mString1 = [NSMutableString stringWithString:@"HelloWorld"];
NSLog(@"%@, %@, %p", mString1, mString1.class, mString1);
NSString *mString11 = [mString1 copy];
NSLog(@"%@, %@, %p", mString11, mString11.class, mString11);
NSMutableString *mString111 = [mString1 mutableCopy];
NSLog(@"%@, %@, %p", mString111, mString111.class, mString111);
[mString111 appendString:@"Hi"];
NSLog(@"%@, %@", mString1, mString111);
```

控制台的打印信息如下：

```
HelloWorld, __NSCFString, 0x6000014bdbc0
HelloWorld, __NSCFString, 0x600001a84e40
HelloWorld, __NSCFString, 0x60000148cab0
HelloWorld, HelloWorldHi
```

下面我们来做一个简单的总结：

（1）对于 NSString 对象，copy 方法进行浅复制，mutableCopy 方法进行深复制并创建出 NSMutableString 对象。

（2）对于 NSMutableString 对象，copy 方法和 mutableCopy 方法都进行深复制，copy 方法创建出不可变的 NSString 对象，mutableCopy 方法创建出可变的 NSMutableString 对象。

在面试时，只要把握住上面两条原则，关于字符串的复制问题就不会难倒你。

3.2　深入理解 NSArray 类

通过上一小节的学习，我们知道 NSString 的实现采用了类簇的方式，其实 NSArray 也是这样，通过类簇的方式在不同的场景下生成不同类型的数组实例，这种实现极大地优化了内存的存储和使用效率。本节我们就一起来探索 NSArray 内部平时容易忽略的深入内容。

3.2.1 NSArray 相关类簇

首先使用 Xcode 创建一个新的工程，编写如下测试代码：

```
NSArray *array0 = [NSArray alloc];
NSArray *array00 = [NSArray alloc];
NSArray *array1 = @[];
NSArray *array2 = [[NSArray alloc] init];
NSArray *array3 = [NSArray arrayWithObject:@"1"];
NSArray *array4 = @[@"1"];
NSArray *array5 = [[NSArray alloc] initWithObjects:@"1", @"2", nil];
NSArray *array6 = @[@"1", @"2", @"3"];
NSLog(@"array0:%@, %p", array0.class, array0);
NSLog(@"array00:%@, %p", array00.class, array00);
NSLog(@"array1:%@, %p", array1.class, array1);
NSLog(@"array2:%@, %p", array2.class, array2);
NSLog(@"array3:%@, %p", array3.class, array3);
NSLog(@"array4:%@, %p", array4.class, array4);
NSLog(@"array5:%@, %p", array5.class, array5);
NSLog(@"array6:%@, %p", array6.class, array6);
```

运行代码，控制台的打印效果如下：

```
array0:__NSPlaceholderArray, 0x10060a5e0
array00:__NSPlaceholderArray, 0x10060a5e0
array1:__NSArray0, 0x100701250
2array2:__NSArray0, 0x100701250
array3:__NSSingleObjectArrayI, 0x1006208a0
array4:__NSSingleObjectArrayI, 0x1006206b0
array5:__NSArrayI, 0x1006029d0
array6:__NSArrayI, 0x10061ecc0
```

从打印信息可以看到，当使用 alloc 创建对象时，实际上生成的是一个占位对象，其类型为__NSPlaceholderArray（其实，NSString 使用 alloc 方法也会先生成一个__NSPlaceholderString 占位对象），这个占位对象是一个单例，当对数组真正地进行初始化时才会创建具体的类簇对象。

无论是使用字面量的方式还是初始化方法的方式，创建出的 NSArray 对象的类型都是__NSArray0，其表示空数组，从打印出的内存地址可以发现所有__NSArray0 实例的地址都一样，这里应用了我们第 2 章中所学习的单例设计模式，无论我们在程序中创建多少个空的不可变数组，都不会增加内存的消耗。

当不可变数组中只有一个元素时，这时创建的数组对象的类型为__NSSingleObjectArrayI，从名字也可以看出，这是一个单元素的数组。由于其单元素的特性，Objective-C 对其实现时，无须考虑数组的列表特性，从而实现优化。

当创建的数组对象中元素个数多于 1 个时，会创建__NSArrayI 对象，__NSArrayI 就是传统意义上的数组对象类型。需要注意的是，如果使用 NSMutableArray 创建实例，则会创建出__NSArrayM 类型的对象，代码如下：

```
NSMutableArray *mArray1 = [[NSMutableArray alloc] init];
NSLog(@"mArray1:%@, %p", mArray1.class, mArray1);
```

打印效果如下：

```
mArray1:__NSArrayM, 0x100557ca0
```

3.2.2　NSArray 数组的内存分布

在理解 NSArray 数组的内存分布之前，我们先来回忆一下在 C 语言中数组的内存分布性质。

我们知道，在 C 语言中，数组实际上是一块连续的内存。C 语言的数组一旦创建完成，其内部元素类型和个数就已经确定。例如：

```
int int_array[] = {1,2,3,4,5,6,7,8,9,10};
```

在取值时，一种方式是使用数组下标来获取对应元素，例如：

```
int_array[3]
```

另一种方式是通过元素的地址来获取元素具体的值。C 语言中的数组地址实际上就是首个元素的地址，可以通过数学运算的方式来获取数组中各个元素的地址，例如：

```
*(int_array + 3)
*(int_array + 5)
```

根据 C 语言数组的原理进行推测，NSArray 可能也是按照这种思路实现的，实际上对于不可变的 NSArray 对象，其内存布局与 C 语言数组基本类似，只是其中存储了更多数组对象本身的信息。例如编写如下测试代码：

```
NSArray *array = @[];
for (int i = 0; i < 10; i++) {
    array = [array arrayByAddingObject:(__bridge id)((void *)(1))];
}
```

需要注意，上面的示例代码中使用了一些小技巧，通常情况下 NSArray 中只能存放对象，即只能存放指针变量。上面我们使用__bridge 桥接的模式将 int 数据存入到数组中，方便调试与观察。

可以在 for 循环代码的最后添加一个断点，之后运行代码。我们使用 lldb 调试器进行数组对象内存分布的观察，在 Xcode 调试区输入如下命令：

```
x/128xb array
```

上面的命令用来查看内存中的数据，其中 128 表示打印出 128 个字节的数据。执行命令，

打印效果如下所示：

```
0x103163990: 0xa9 0x22 0xea 0x8c 0xff 0xff 0x1d 0x01
0x103163998: 0x0a 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639a0: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639a8: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639b0: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639b8: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639c0: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639c8: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639d0: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639d8: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639e0: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1031639e8: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

在如上打印信息中，0x10316399 就是 NSArray 对象的地址，其后的 8 个字节，即 0x103163998 所存储的数据实际上是数组中元素的个数，当前数组中有 10 个元素。其后连续的内存中存储的便是真正的元素数据，可以看到，其中存储的数据都是数值 1。

需要注意，对于不可变的数组，并非都是以如上所述的方式进行内存布局，上面的内存布局适用于__NSArrayl 类型的对象，对于单元素的数组__NSSingleObjectArrayI 对象，则不进行数组元素个数信息的存储，第 2 个 8 字节的内存直接存储着数组中唯一的对象数据。

相对不可变的 NSArray，可变的 NSMutableArray 内存分布复杂得多。由于 NSMutableArray 要经常进行增删，其是采用了效率更高的环形缓冲区结构来布局内存。

3.3 NSDictionary 的相关内容

NSDictionary 是 iOS 开发中使用非常频繁的一种数据类型。其通过键值对的方式进行数据的存储，使用 NSDictionary 对元素进行增删改查都非常容易，在较高级 iOS 开发职位的面试中，关于 NSDictionary 也有非常多的深入内容可以探究。

3.3.1 NSDictionary 类簇

与 NSString 和 NSArray 类型相似，NSDictionary 类型也是通过类簇的方式实现的，根据使用场景的差异，Objective-C 语言对这些数据类型都做了语言层面的优化。首先，使用 Xcode 开发工具创建一个新的工程，在其中编写如下测试代码：

```
    NSDictionary *dic = [NSDictionary alloc];
    NSDictionary *dic0 = @{};
    NSDictionary *dic1 = @{@"1":@"1", @"2" : @"2"};
    NSDictionary *dic2 = [[NSDictionary alloc] initWithObjectsAndKeys:@"1", @"1",
nil];
```