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];
```

```
NSMutableDictionary *mdic = [[NSMutableDictionary alloc] init];
NSLog(@"dic:%@, %p", dic.class, dic);
NSLog(@"dic0:%@, %p", dic0.class, dic0);
NSLog(@"dic1:%@, %p", dic1.class, dic1);
NSLog(@"dic2:%@, %p", dic2.class, dic2);
NSLog(@"mdic:%@, %p", mdic.class, mdic);
```

运行上面的测试代码，控制台打印效果如下：

```
dic:__NSPlaceholderDictionary, 0x100701a90
dic0:__NSDictionary0, 0x100701100
dic1:__NSDictionaryI, 0x1007bdea0
dic2:__NSSingleEntryDictionaryI, 0x1007bd720
mdic:__NSDictionaryM, 0x1007be1a0
```

从打印信息可以看出，在 NSDictionary 的类簇中，比较常见的类型有__NSPlaceholderDictionary、__NSDictionary0、__NSDictionaryI、__NSSingleEntryDictionaryI和__NSDictionaryM。

其中，__NSPlaceholderDictionary 类是单例类，其实例对象用来进行未初始化字典的占位。__NSDictionary0 类也是单例类，当一个字典为不可变的空字典时，就会实例化出这个类的对象。__NSSingleEntryDictionaryI 是单元素的字典类，由于字典中只有一个元素，因此在数据结构的设计和内存布局时都可以进行优化。__NSDictionaryI 是常规的不可变字典类，__NSDictionaryM 是可变字典类。

3.3.2　了解哈希表

我们知道，NSDictionary 类型是通过键值对的方式进行数据储存的，在深入理解 NSDictionary 的原理前，首先需要对哈希表这种数据结构有简单的认识。

哈希表又称为散列表，其是存储大量数据的一种方式，哈希表有着非常高的查找效率，理想状态下，其时间复杂度为 $O(1)$。哈希表是通过关键码值而直接进行数据访问的数据结构，即通过 key 值快速地查找 value 值。其核心是通过关键码（key）的哈希变换，将其映射到表中的一个位置来进行数据的访问，这个映射的过程通常叫作映射函数或散列函数，用来存放数据的集合叫作哈希表或散列表。

例如，以教师对象的存储为例，每一个教师对象都有一个编号，我们可以取教师编号的最后两位作为表中的地址进行数据的存储，那么此哈希函数可以简单编写如下：

```
int hash(NSString* number) {
    return [[number substringFromIndex:number.length - 2] intValue];
}
```

这样我们就可以将任意一个教师对象映射到 0~99 之间的一个位置进行存放，可以通过代码来尝试实现此哈希表结构，首先定义教师对象如下：

```
@interface Teacher : NSObject
```

```
    @property (nonatomic, copy) NSString *name;
    @property (nonatomic, copy) NSString *number;
    @end
    @implementation Teacher
    - (NSString *)description {
        return [NSString stringWithFormat:@"number:%@ name:%@", self.number,
self.name];
    }
    @end
```

定义一个简易的哈希表类：

```
@interface HashTable : NSObject
@end
@implementation HashTable
{
    void*  _table[100];
}
- (void)setV:(id)v forK:(NSString *)k {
    _table[hash(k)] = (__bridge void *)(v);
}
- (id)getV:(NSString *)k {
    return (__bridge id)(*(_table + hash(k)));
}
@end
```

如上代码所示，在 HashTable 对象的 nebula 实际上创建了一个指针数组，数组的容量为 100 个元素，之所以定义为 100 个元素的容量，是与具体的哈希函数相关的，由于我们选择的哈希函数会将教师的变换映射为 0~99 之间的数值，因此实际上只有 100 个位置可以进行数据的存储。

可以在 main 函数中测试上面哈希表的功能，代码如下所示：

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        HashTable *table = [[HashTable alloc] init];
        Teacher *t1 = [[Teacher alloc] init];
        t1.name = @"珲少";
        t1.number = @"001";
        Teacher *t2 = [[Teacher alloc] init];
        t2.name = @"Lucy";
        t2.number = @"002";
        [table setV:t1 forK:t1.number];
        [table setV:t2 forK:t2.number];
```

```
        NSLog(@"%@", [table getV:@"001"]);
        NSLog(@"%@", [table getV:@"002"]);
    }
    return 0;
}
```

运行代码，从控制台的打印信息可以看出，我们编写的简易哈希表已经可以正常使用。哈希表这种数据结构在查找数据时，直接根据关键码转换成地址位置，查找效率非常高，但是其也会引入许多复杂的问题，我们在下一小节中具体讨论。

3.3.3　处理哈希碰撞

通过上一小节的学习，我们实现了一个简易的哈希表，还以教师存储为例，假设教师数量不会超过 100 个，但是按照上一小节的实现，这个哈希表依然存在很严重的问题，即会产生碰撞，例如：

```
HashTable *table = [[HashTable alloc] init];
Teacher *t1 = [[Teacher alloc] init];
t1.name = @"珲少";
t1.number = @"011";
Teacher *t2 = [[Teacher alloc] init];
t2.name = @"Lucy";
t2.number = @"211";
[table setV:t1 forK:t1.number];
[table setV:t2 forK:t2.number];
NSLog(@"%@", [table getV:@"011"]);
NSLog(@"%@", [table getV:@"311"]);
```

运行上面的代码，程序会产生很严重的问题。首先，我们向哈希表中存储了两个编号不同的教师，结果后存入的教师对象将先存入的教师对象覆盖了。其次，当我们使用教师编码获取教师对象时，一个错误的教师编码也获取到了教师对象。造成这两个问题的原因在于我们所设定的哈希函数的局限性，此哈希函数截取了字符串最后两位进行映射，很容易产生哈希碰撞。

哈希碰撞是指使用不同的关键码计算出相同的哈希位置，即哈希表实际上是将无限的定义域映射到了有限的值域中，因此从原理上一定会出现哈希碰撞，产生碰撞的元素通常被称为溢出元素。

处理哈希碰撞通常有两种思路：一种是将溢出的元素存储到当前散列表中未存放元素的位置；一种是将溢出的元素存储到散列外面的线性表中。

将溢出的元素存储到当前散列中空闲的位置上常用的方法有线性探针法、二次探测法、再散列法。线性探针法是指当出现碰撞时，从映射到的位置开始依次向后查找空位置，如果需要也会重置到首位进行查找，使用这种方式进行碰撞处理会使表中大量的连续位置被占用，并且可能会引发更多的碰撞。二次探测法与线性探针类似，只是在查找空闲位置时是从离初始位

置远的地方开始，修正了线性探针法造成的初始聚集问题。再散列法是指除了使用一个哈希函数计算散列表中位置外，如果发生碰撞，会使用第二个哈希函数计算偏移步长，即计算出需要移动的长度将数据存入散列表。这 3 种处理碰撞的方法都没有开辟新的存储空间，但是会使位置的计算过程变得复杂，降低效率。

将溢出的元素存入散列表以外的地方通常使用拉链法，即将同一个散列地址的数据都存入一个链表中（或者将溢出的数据存入关联到当前散列位置的链表中）。通常情况下，只要哈希函数的设计合适、散列范围的定义合适，尽量保证散列的均匀，碰撞的概率并不会很大，这时使用拉链法效率处理碰撞效率很高。

我们尝试使用拉链法处理之前编写的哈希表，完整代码如下：

```
#import <Foundation/Foundation.h>
int hash(NSString* number) {
    return [[number substringFromIndex:number.length - 2] intValue];
}

@interface Teacher : NSObject
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *number;

@end
    @implementation Teacher
    - (NSString *)description {
        return [NSString stringWithFormat:@"number:%@ name:%@", self.number,
self.name];
    }
    - (BOOL)isEqual:(Teacher *)object {
        return [self.number isEqualToString:object.number];
    }
    @end

    @interface ListNode : NSObject
    @property (nonatomic, strong) Teacher *value;
    @property (nonatomic, strong) ListNode *nextNode;
    @end
    @implementation ListNode
    @end
    @interface HashTable : NSObject
    @end
    @implementation HashTable
    {
        void*  _table[100];
```

```
}
- (void)setV:(Teacher *)v forK:(NSString *)k {
    ListNode *firstNode = (__bridge id)(*(_table + hash(k)));
    if (firstNode == nil) {
        firstNode = [[ListNode alloc] init];
        firstNode.value = v;
        _table[hash(k)] = (__bridge_retained void *)(firstNode);
    } else {
        while (firstNode != nil) {
            if ([firstNode.value isEqual:v]) {
                firstNode.value = v;
                return;
            }
            firstNode = firstNode.nextNode;
        }
        ListNode *node = [[ListNode alloc] init];
        node.value = v;
        firstNode = (__bridge id)(*(_table + hash(k)));
        firstNode.nextNode = node;
    }
}
- (Teacher *)getV:(NSString *)k {
    ListNode *firstNode = (__bridge id)(*(_table + hash(k)));
    while (firstNode != nil) {
        if ([firstNode.value.number isEqualToString:k]) {
            return firstNode.value;
        }
        firstNode = firstNode.nextNode;
    }
    return nil;
}
@end

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...
        HashTable *table = [[HashTable alloc] init];
        Teacher *t1 = [[Teacher alloc] init];
        t1.name = @"珲少";
        t1.number = @"011";
        Teacher *t2 = [[Teacher alloc] init];
        t2.name = @"Lucy";
```

```
        t2.number = @"211";
        [table setV:t1 forK:t1.number];
        [table setV:t2 forK:t2.number];
        NSLog(@"%@", [table getV:@"011"]);
        NSLog(@"%@", [table getV:@"211"]);
        NSLog(@"%@", [table getV:@"311"]);
    }
    return 0;
}
```

运行代码，通过打印信息可以看到，我们编写的哈希表已经可以高效地处理碰撞问题了。上面的代码只是从原理上为大家演示了哈希表的工作方式，其实在实际应用中 NSDictionary 的实现要更加复杂一些。万变不离其宗，理解了哈希表，后面理解 NSDictionary 就会游刃有余。

3.3.4 NSDictionary 的实现原理

NSDictionary 是 Objective-C 中用来进行键值存储的一种数据结构。其内部实际上是使用 NSMapTable 这种数据类型实现的。NSMapTable 其实就是一种哈希表。NSMapTabel 对哈希碰撞的处理采用了关联链表的方式，因此最理想的状态下，使用其进行数据的查询时间复杂度为 $O(1)$，最坏的情况下（所有元素都产生的哈希碰撞）时间复杂度为 $O(n)$。

在日常开发中使用 NSDictionary 时，通常使用字符串作为数据的键，其实并非只有字符串可以作为 NSDictionary 的键。某一种数据类型如果想要作为键，就需要满足下面两个条件：

（1）遵守 NSCopying 协议，支持复制操作。在 NSDictionary 内部，键会被复制一份，而值会采用引用计数的方式进行强引用。

（2）遵守 NSObject 协议，并实现其中的如下两个方法：

```
- (BOOL)isEqual:(id)object;
@property (readonly) NSUInteger hash;
```

其中，实现 isEqual 方法用来对产生碰撞的键进行比较操作；hash 是一个只读属性，为其实现 get 方法用来定义哈希函数。

还有一点需要注意，在 Objective-C 中，NSArray 通常也被称作有序的列表，NSSet 被称为无序的集合，实际上 NSSet 的实现也是采用的哈希表的方式。

3.4 Swift 语言中的字符串、数组与字典类型

Swift 语言在设计思想上与 Objective-C 语言有着很大的差异。在 Swift 语言中，枚举、结构体等数据类型十分强大，并且在其核心库里相关数据类型的实现中大量使用了协议与扩展的方式，从设计上讲 Swift 语言有着更加现代化的特性，编程更加面向协议，安全性与扩展性都