第 35 卷 第 8 期 计 算 机 工 程 2009 年 4 月
Vol.35 No.8 Computer Engineering April 2009
·软件技术与数据库· 文章编号：1000—3428(2009)08—0044—03 文献标识码：A 中图分类号：TP311.12

基于散列表的 CANopen 对象字典的设计

徐 喆，闫士珍，宋 威
(北京工业大学电子信息与控制工程学院，北京 100022)

摘 要：对象字典的实现是整个 CANopen 协议的关键。对象字典的构建可以采用数组或链表的方式。数组方式占用大量内存空间并且不便于动态的更新，链表方式对于大数据量搜索效率较低。而散列表由于其自身的结构特点则可以克服这些缺点。该文采用散列表的方式构建对象字典，这种方式构建的对象字典具有可动态更新、搜索效率高和存储空间利用率高等优点。

关键词：CANopen 协议；对象字典；散列表

Object Dictionary Design of CANopen Based on Hash Table

XU Zhe, YAN Shi-zhen, SONG Wei
(College of Electronic Information and Control Engineering, Beijing University of Technology, Beijing 100022)

【Abstract】The realization of object dictionary is very important to CANopen proposal. It can be realized using array or linked list. The way of array requires large amounts of memory space and it is hard to update object dictionary dynamically. The way of linked list is so low in search efficiency for a large quantity of data. Because of the characteristic of structure, Hash table can overcome these disadvantages. In this paper, object dictionary is designed using Hash table, which can update dynamically and search efficiently. In addition, it is more efficient to make use of memory space.

【Key words】CANopen protocol; object dictionary; Hash table

1 概述

对象字典[1]是 CANopen 协议的核心，是一个有序的对象组，其中定义了 CANopen 网络中设备的所有信息，每个设备的对象字典具有结构相同、内容不同的特点。对象字典主索引范围是 0000H~FFFFH，特定的参数分布在临近的寻址区域，如表 1 所示。

表 1 对象字典结构

索引	对象
0000H	未用
0001H~001FH	静态数据类型
0020H~003FH	复杂数据类型
0040H~005FH	制造商规定的复杂数据类型
0060H~007FH	设备子协议规定的静态数据类型
0080H~009FH	设备子协议规定的复杂数据类型
00A0H~0FFFH	保留
1000H~1FFFH	通信子协议区域
2000H~5FFFH	制造商特定子协议区域
6000H~9FFFH	标准的设备子协议区域
A000H~FFFFH	保留

对象字典虽然占用很大的空间，但并不是所用的参数都要定义。其中必须要实现的只有 1000H~1FFFH 和 6000H~9FFFH。所以，在建立设备对象字典时，主要工作应集中在这 2 个区域的定义。

对象字典中每个对象采用一个 16 位的主索引值和一个 8 位的子索引值来寻址。每个对象的主要信息包括索引、子索引、数据、数据类型和访问类型。

根据对象包含的主要信息，可以用 C 语言实现对象的结构如下所示：

```
typedef struct _ODIndex {
int    index;
char   subIndex;
char   indexType;
long   data;
char   (*pFun)(struct _ODIndex *, char, long);
struct _ODIndex * next;
} ODIndex, *pODIndex
```

其中，index 是对象的索引；subIndex 是对象的子索引；indexType 是索引类型，由数据类型和访问类型共同决定；data 是对象中存储的数据；*pFun 为与该对象相关的函数的指针。

在 CANopen 网络中主站行使网络管理、配置从站的功能。主站对象字典的构建必须考虑网络的灵活扩展能力，使其可以灵活地增减节点。网络中节点的增减必然要求修改主站对象字典，甚至大量添加、删除对象字典中的对象。所以，主站的对象字典必须具有可快速查找、添加、删除对象的特性。

2 对象字典的实现

2.1 对象字典的实现方式

对象字典的实现可以采用数组方式和链表方式。数组将元素在内存中连续存放，由于每个元素占用内存相同，因此可以通过下标迅速访问数组中任何一个元素。数组的优点很明显，搜索效率高。但是用数组存放数据时，必须事先定义固定的长度(即元素个数)。然而主站对象字典中对象的个数具有不确定性，并且所实现的对象字典具有不连续性。因此，如果采用数组方式必然要把数组定义得足够大，以便存放足

基金项目：北京市教委科技创新平台基金资助项目(05002011200701)
作者简介：徐 喆(1968－)，女，副教授、博士，主研方向：网络控制系统，推理控制，软测量；闫士珍、宋 威，硕士
收稿日期：2008-08-14 **E-mail：**xuzhe@bjut.edu.cn

够多的对象。链表恰好相反，链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。它根据需要动态开辟内存单元。但是如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置，所以链表的搜索效率比较低。

以 1000H~1FFFH 处的对象为例，在不考虑子索引的情况下有 4 096 个对象，这些对象中只有 1000H, 1001H 和 1018H 是必选的，其他对象都是可选的。用数组的方式建立这部分对象字典需要分配 4 096 个存储空间。知道索引 1000H 的对象的存储地址即可根据偏移量快速访问 1000H~1FFFH 中任何一个对象。搜索数组中的每一个元素时，平均比较次数为 1。用链表方式建立这部分对象字典时存储空间可以根据需要动态分配。搜索链表中的一个结点时，平均比较次数可表示为$(n+1)/2$，其中，n 代表所建对象个数。

显然，数组方式构建对象字典搜索效率高，但浪费存储空间；链表方式构建对象字典节省存储空间，但搜索效率低。对于对存储空间和代码执行效率要求较高的嵌入式系统来说，这 2 种实现方式显然都存在严重的缺点。本设计采用散列表[2]的方式实现对象字典。该方式的结构介于数组和链表之间，既避免了数组浪费存储空间的缺点，又克服了链表搜索效率低的不足。

2.2 散列表构建对象字典基本原理

使用一个下标范围比较大的数组来存储对象。可以设计一个函数(哈希函数，也叫作散列函数)，使得每个对象的索引和子索引都与一个函数值(即数组下标)相对应，于是用这个数组单元来存储这个对象；也可以简单地理解为，按照索引和子索引为每一个对象“分类”，然后将这个对象存储在相应“类”所对应的地方。数组定义如下：

ODIndex iOD[HT_WIDTH]

其中，HT_WIDTH 为散列表的宽度。

但是，不能够保证每个对象的索引和子索引与函数值是一一对应的，因此，极有可能出现对于不同的对象，却计算出了相同的函数值，这样就产生了冲突[2]。

对象的索引和子索引通过哈希函数[2]转化为散列地址。应尽可能地使索引和子索引经过哈希函数得到一个随机的地址，以便将对象均匀地分布到整个地址区间中。如果保存在散列表中的索引和子索引不连续分布，映射函数具有伪随机性，则可将对象合理地得分布到散列表中。

由于对象字典访问的频繁性，映射函数需要足够简单以减少运算时间，这里采用如下哈希函数：

$$HS\_index = \mathrm{mod}(((index \times ran\_num) >> shift\_num + sub\_index), HS\_index\_range) \quad (1)$$

其中，*HS_index* 为散列地址；mod()为取模运算；*index* 为对象索引；*ran_num* 为随机因子；*shift_num* 为位移因子；*sub_index* 为对象子索引；*HS_index_range* 为散列地址的范围常量。

如前所述，不能保证从 *index* 和 *sub_index* 到 *HS_index* 的映射不会重叠，会产生冲突。这里处理冲突的方法采用链地址法[2]，将映射到同一个散列地址处多于一个的对象组成链表，即形成该散列地址对应的溢出表。从散列表的相应位置可以访问其对应的溢出表。

用散列表构建的对象字典逻辑结构如图 1 所示，散列表地址范围为 M，E 代表一个索引项，N 代表一个未使用的散列表项。

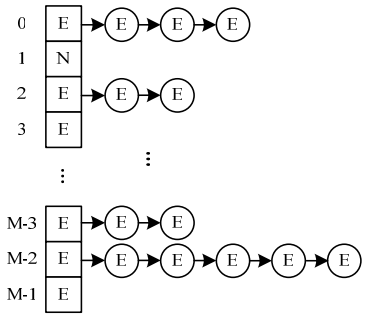

图 1 对象字典逻辑结构

散列地址范围取值时应适中，取值过大将会占用过多存储空间，取值过小则会产生较长的溢出表，降低搜索效率。这里取 *HS_index_range* 为 512。取 779 个样本对象进行测试，如图 2 所示。

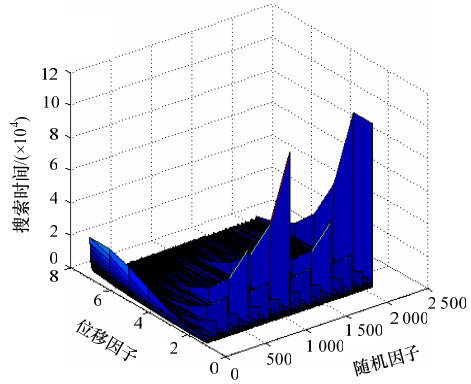

图 2 随机因子，位移因子和搜索时间之间的关系

图 2 中横坐标轴代表随机因子 *ran_num*，纵坐标轴代表位移因子 *shift_num*，竖坐标轴代表搜索时间，搜索时间的计算公式如下：

$$sum = \sum_{i=1}^{512} \sum_{k=0}^{n(i)} k \quad (2)$$

其中，*sum* 为遍历 779 个对象需要的搜索时间；$n(i)$代表样本空间中映射到散列地址 i 的对象个数。*sum* 值越小说明散列表的填充率越高。

经测试可知，当 *ran_num* 取 193, *shift_num* 取 3 时散列表填充率最高，可达 99.8%。

2.3 散列表方式构建对象字典的性能分析

散列表方法是数组和链表搜索的结合。对于散列表的搜索和数组相同，但是对溢出表则采用链表的搜索方式。

数组搜索由于下标的使用，为 $O(1)$算法。链表只能单向搜索，如果链表中的节点为随机分布，那么平均搜索时间为 $O(n)$，其中，n 为链表的长度。

在散列表的填充率为 100%, $m \geqslant N$ 的情况下，可以得到散列表的平均长度：

$$l = (m/N) - 1 \quad (3)$$

其中，m 为对象字典项总数；N 为散列表地址范围。则散列表的平均搜索时间为

$$\bar{t} = \begin{cases} 1 & m < N \\ \dfrac{N}{m} + \dfrac{m-N}{m} \cdot \left(\dfrac{\frac{m}{N} - 1}{2} + 1 \right) & m \geqslant N \end{cases}$$

化简可得

$$\bar{t}=\begin{cases}1 & m<N\\ \dfrac{N^2+m^2}{2mN} & m\geqslant N\end{cases}$$

仍以 1000H~1FFFH 处的对象为例，对数组、链表和散列表 3 种方式实现这部分对象字典作性能比较，如表 2 所示。

表 2　3 种方式性能比较

比较对象	数组	链表	散列表
搜索一个对象的平均比较次数	1	(*n*+1)/2	*n*>512 时为 *n*/1 024+ 256/*n*, *n*≤512 时为 1
占用存储空间个数	4 096	动态分配	512+动态分配

在表 2 中，*n* 代表实际需要建立的对象的个数。散列表的平均比较次数为理想状态下的取值，即对象完全均匀地分布在散列表中。实际出现这种情况的可能性很小。但是即便是出现最差的情况，即所有对象都映射到同一个散列地址处，平均比较次数也和链表方式基本相同，为(*n*-1)/2+1。这种情况出现的可能性也是很小的。

很明显，散列表方式实现的对象字典搜索效率要明显高于链表方式；与数组方式相比则可以节省大量存储空间。

2.4　对象字典待分配空间

如果在建立对象字典时每次只分配一个对象的存储空间，那么将会反复执行分配内存空间的操作。这样必然会带来内存碎片的问题，从而使内存空间的使用效率降低。为了解决这一问题，每次分配多个存储空间，减少动态分配内存空间的次数，从而减少内存碎片。为了维护分配的存储空间，引入对象字典待分配空间 *iODMallocPool*。其数据类型如下：

```
typedef struct _ODIndexMallocPool{
    long ref;
    pODIndex space;
} ODIndexMallocPool, *pODIndexMallocPool
```

其中，*ref* 代表 *iODMallocPool* 中剩余的存储空间个数；*space* 代表 *iODMallocPool* 中第 1 个存储空间的地址。

分配存储空间时，将分配的多个存储空间组成链表，存储空间个数存入 *ref* 中，链表的头结点的地址存入 *space* 中。新建对象需要从 *iODMallocPool* 中取出存储空间时，*ref* 的值减 1，将链表中下一个结点的地址赋给 *space*；删除对象时，将对象占用的存储空间放回 *iODMallocPool* 所维护的链表的头部，*ref* 值加 1。当该空间为空即 *ref* 为 0 时，自动申请一个固定个数的存储空间，然后更新 *ref* 和 *space* 的值。

2.5　对象字典中对象的新建、查找和删除

新建对象时，根据给定的索引和子索引通过哈希函数计算出散列地址。若散列表中此地址为空，则此地址被视为新建对象的存储空间的地址；否则从对象字典待分配空间中获取一个存储空间作为新建对象的存储空间，将其插入该散列地址对应的溢出表表尾。

删除对象时，根据给定的索引和子索引通过哈希函数计算出散列地址。若散列表中该地址为空，则欲删除的对象不存在；否则，比较该地址处存储的索引和子索引及给定的值是否相等。若相等，则将其置空。否则在进入溢出表中查找索引和子索引与给定值相同的结点。找到后将其从散列表中删除，然后将该结点的内存空间放进对象字典待分配空间中以备新建对象时使用。

查找对象时，根据给定的索引和子索引通过哈希函数计算出散列地址。若散列表中此地址为空，则查找不成功；否则，比较该地址处存储的索引和子索引及给定的值是否相等。若相等，查找成功；否则，进入溢出表查找，直至溢出表末尾或找到某一结点存储的索引和子索引与给定值相等为止。

2.6　散列表的动态更新

如前所述，本文处理冲突的方法是链地址法，每一个散列地址都会对应一个溢出表。当产生较长的溢出表时必然会增加访问时间，降低搜索效率。为了进一步提高搜索效率，借鉴计算机高速缓冲存储器的思想，尝试将经常访问的对象字典项放在溢出表的较前位置，以减少常用对象字典项的访问时间。如果访问溢出表中某一结点需要的比较次数大于阈值 *OD_TARGET_DEPTH*，则将该结点移到溢出表表头的位置。从而可以将经常被访问的对象存放在溢出表中靠前的位置。当再次访问这些对象时，可以快速地从溢出表中获得，从而进一步提高了搜索效率。

引入高速缓存思想前后搜索效率比较如图 3 所示。

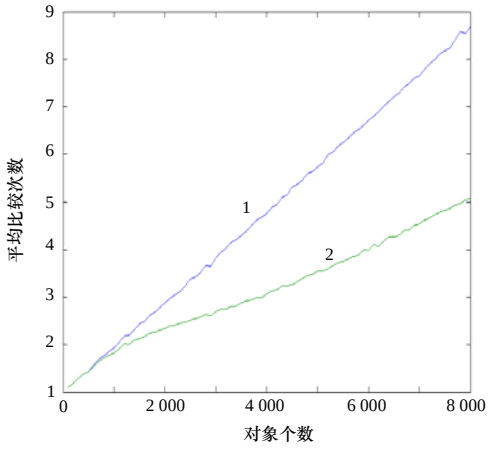

图 3　引入高速缓存思想前后搜索效率的比较

在图 3 中，横坐标表示对象的个数，纵坐标表示搜索常用对象(随机取总的对象个数的 1/4 作为常用对象)的平均比较次数。其中，曲线 1 为没有引入高速缓存思想时所得到的曲线；曲线 2 表示引入了高速缓存思想后得到的曲线。显然，引入高速缓存思想后搜索效率得到了明显提高。

3　结束语

对象字典中定义了所有通信参数和设备参数，是 CANopen 协议的核心。因此，对象字典的性能影响到整个协议栈的性能。本文提出的基于散列表的方式构建的对象字典可以实现动态的更新对象字典。这不但对网络的扩展提供了方便，节省了大量内存空间，而且搜索效率也得到了较大的提高。

参考文献

[1] CiA. CiA Draft Standard 301 CANopen Application Layer and Communication Profile[S]. 2002.

[2] 严蔚敏, 吴伟民. 数据结构[M]. 北京: 清华大学出版社, 1996.

编辑　顾逸斐