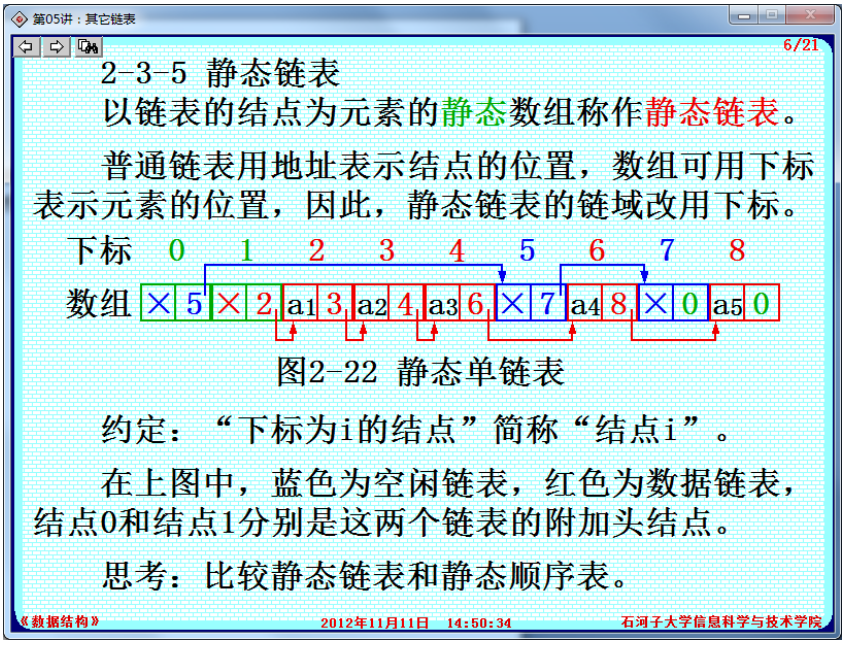
静态链表在优化中的应用

静态链表的基本结构网上到处都是，大学课本里也很详细，不熟悉的同学请看下面我大学时课件抓图，在此感谢楼老师。（说实话，大学那么多课，只有您的课仔细听过 ☺）



数据结构课程设计帮同学做过好几个静态链表的题，没想到毕业两年多来用这东西做过好几个优化，分别如下：

1、DES 密钥缓存

金融交易安全类项目，存储的就是普通的 Key-Value 结构，Key 是密钥名，Value 是 DES 密钥，都为短字符串，原来的方案是开个共享内存做数组，按 Key 排序，折半查找还算说的过去，但插入和删除就是 O(n)了，所以在业务高峰时期就悲剧了……

有些同学可能想“为什么要用共享内存？直接用一个 STL map/unordered_map 就直接搞定。为什么不用现有的 memcached/redis……解决方案……”，记得后来和 coolshell 的陈皓老师 email 交流过，他也有这类疑问；原因大概有这样几个：

- 是一个完完全全的纯 C 项目。
- 有历史原因，必须沿用原来的 Sys V 共享内存方式；另一方面最初用共享内存也是为防止程序挂掉后业务数据不丢失。
- 做过银行电信的同学可能知道，银行不可能信任 memcached/redis……，他们更多的是认小型机/AIX/DB2/Oracle……甚至 MySQL 都信不过，所以更不可能采用时下互联网流行的解决方案。

故只能在原来的方案上优化，而且设想的是要将密钥数据和所用查询机制的索引之类的东西分开存放；大概如下：

1、 开 2 块共享内存 A、B；A 用来存放索引信息，B 用来存放实际的密钥数据。

2、 A 中就是普通的散列表，线性再散列法，存储密钥名对应的密钥在 B 中的静态链表中的**下标值**。

3、 共享内存内不可能用传统的链表结构，所以在 B 中使用双向静态链表存储密钥数据。

A 中散列表存的是数据在 B 中静态链表中的下标值为关键，实现 O(1)访问；如果不把密钥数据和索引数据分开存放，就不用 B 共享内存，直接用 A 存储密钥即可。

这也算是我第一次将静态链表使用于应用中。课本中告诉我们静态链表一般用于没有指针的如BASIC/FORTRAN 之类的语言，它和普通链表一样，插入删除不需要移动元素，无法随机存储等等…… 什么？无法随机访问？如果我们直接拿数据在静态链表中的下标不就可以直接访问了么？

2、通信层和应用层映射

有些同学会在通信层接受连接后生成一个类似可读的client_id之类的值作为通信层和应用层之间交互的映射，映射的方法是通过查 hash map 实现，但是这里的查询是非常频繁的，哪怕是 hash map 也会带来一定的效率损耗。

熟悉 Reactor 机制实现的同学知道底层通信可以使用一个 max_fd 大小的数组作为 fd 到 Event_Handler 结构的映射，在 C++中的写法如下:

```
std::vector<Event_Handler *> fd_map(max_fd(), (Event_Handler *)0);
```

epoll_wait 返回的 fd 集合可以直接以 O(1)的效率映射到对应的 Event_Handler，fd 和 client_id 都存在 Event_Handler 中。

好，现在解决了 fd→Event_Handler→client_id 的映射，那反过来 client_id→Event_Handler→fd 呢？

简单的生成 client_id 的方法可能是一个 int 从 0 递增，通过 client_id 去 hash map 查询 Event_Handler；可能为如下结构：

```
std::unordered_map<int, Event_Handler *> fd_client_id_map(max_fd());
```

但即便是 hash map 也比数组下标访问有一定效率损失；说道这里可能大家也知道了，如果以静态链表存放 Event_Handler *，client_id 就是 Event_Handler *在一个静态链表中的下标，那么就可以直接通过下标取得 Event_Handler *，也就以 O(1)打通了 client_id→Event_Handler→fd 这条路。

Event_Handler *存储在一个静态链表中,它所在的下标作为其 client_id,这个静态链表用 STL vector 实现，可做成动态增加、减少大小；不用做成固定大小，

还有其他几个和具体应用相关的静态链表的使用。

总结：

后来仔细想了下，其实**凡是需要动态生成一个 id，并且需要用此 id 做到另一个东西的映射都可以如上方式使用静态链表，实现真正的 O(1)。**