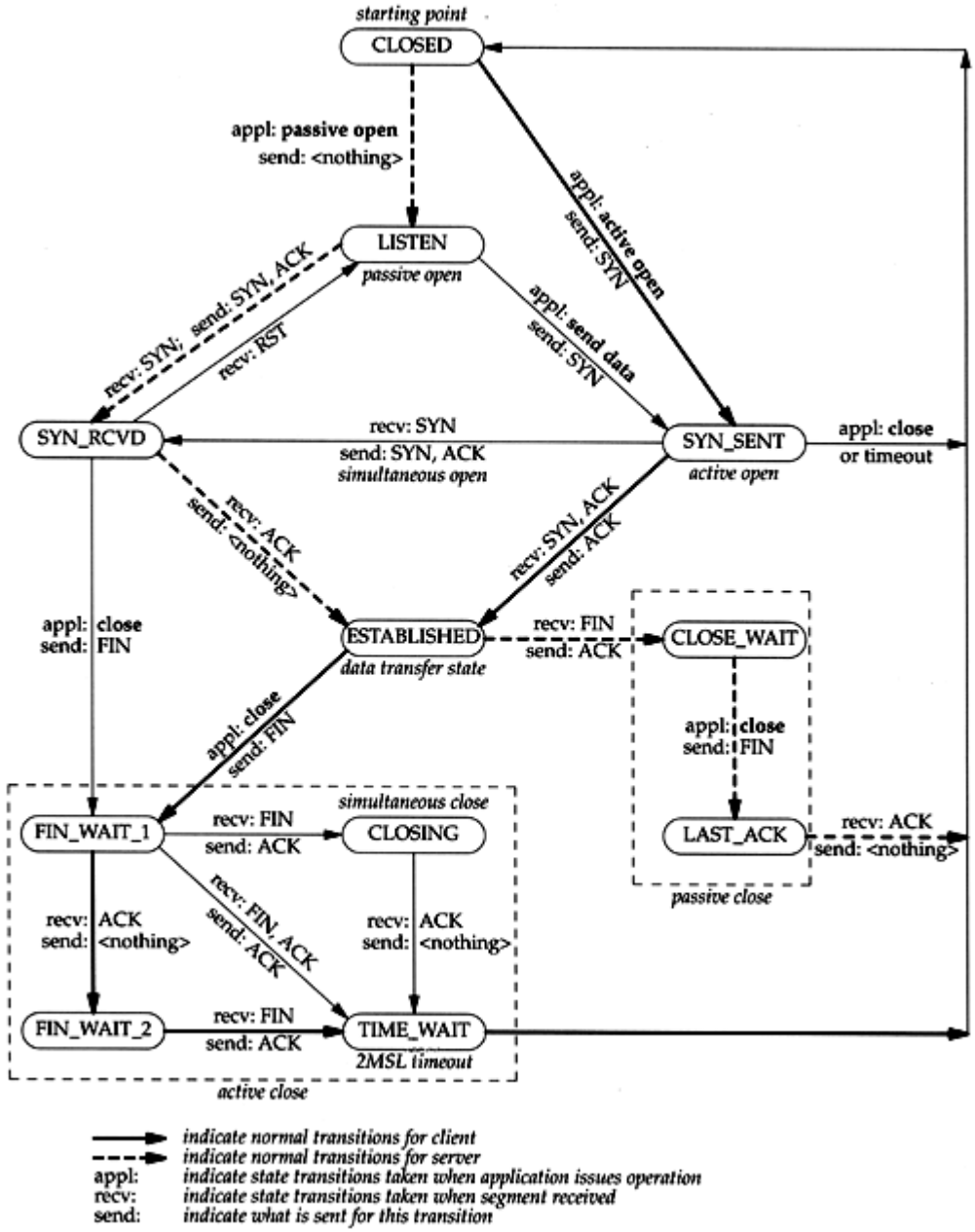
TCP 的状态转移图



为了管理处于不同状态的 tcp_sock，把不同状态的 tcp_sock 分别放入不同的哈希表里。用一个全局变量 tcp_hashinfo 对这些哈希表进行管理。

Inet_hashinfo

Tcp_hashinfo 是 inet_hashinfo 类型，inet_hashinfo 的定义如下：

```
struct inet_hashinfo {
```

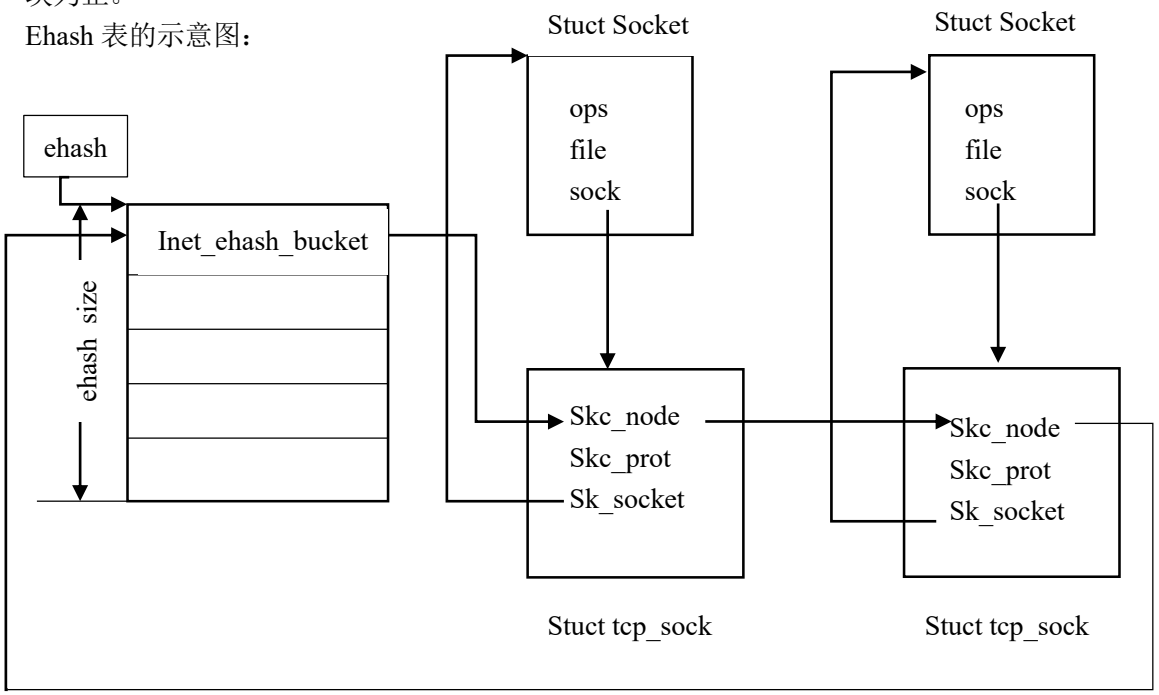
```
    struct inet_ehash_bucket      *ehash;   /* Ehash 是用来管理处于除 LISTEN 状态之外的
tcp_sock 的哈希表 */
    struct inet_bind_hashbucket      *bhash;   /* Bhash 是用来管理已经绑定了端口的
tcp_sock 的哈希表*/

    int                  bhash_size; /* bhash 表的大小*/
    unsigned int             ehash_size;   /*ehash 表的大小*/

    struct hlist_head        listening_hash[INET_LHTABLE_SIZE];   / *Listening_hash 是用来
管理处理 Listen 状态的 tcp_sock 的哈希表,大小为 32 */

    /* All the above members are written once at bootup and
     * never written again _or_ are predominantly read-access.
     *
     * Now align to a new cache line as all the following members
     * are often dirty.
     */
    rwlock_t             lhash_lock ____cacheline_aligned;
    atomic_t             lhash_users;
    wait_queue_head_t            lhash_wait;
    struct kmem_cache                *bind_bucket_cachep;
};
```

其中 ehash 是 inet_ehash_bucket 类型的指针，inet_ehash_bucket 的结构如下：

```
struct inet_ehash_bucket {
    rwlock_t    lock;
    struct hlist_head chain;
};
```

当成功创建一个传输控制块后，就会添加到 tcp_infor 的 ehash 表中，直到释放这个传输控制块为止。

Ehash 表的示意图：



那么如何通过使用这个哈希表呢？对于键值对（key, value），先得到 key，然后得到 value 为 tcp_info->ehash[key]。
首先，我们来计算 key。给定一个 struct sock *sk, 转化为 inet_sock * inet_sk, 我们可以得到一个四元组(laddr, lport, faddr, fport)分别代表本地 IP， 本地端口，远地 IP, 远地端口。通过对这个四元组使用哈希算法(调用 inet_ehashfn 函数)，得到一个 int 值 h = hashfn(laddr, lport, faddr, fport)。通过这个 h, 得到我们要的 value = tcp_info->ehash[h & (ehash_size) -1]。这就是一个 inet_ehash_bucket 的首地址，然后得到链表头 listhead = value->chain。
往这个链表中加入一个节点是 hlist_add_head(&sk->sk_node, list)。
这样就完成了一个往哈希表中添加一个节点的过程。

Bhash 是 inet_bind_hashbucket 类型的指针，inet_bind_hashbucket 的结构如下:

```
struct inet_bind_hashbucket {
        spinlock_t              lock;
        struct hlist_head       chain;
};
```

对于 bhash 表，存的是还没有建立起连接只绑定了端口的连接控制块，只用端口就可以得到哈希表的索引 key = Inet_bhashfn(lport, bhash_size)，哈希算法很简单，就是取 lport & (bhash_size - 1)。只需要本地端口和 bhash 表的 size 就可以。然后得到 value = tcp_into->bhash[key]， 是一个 inet_bind_hashbuccket 的首地址，然后通过 value->chain 就可以得到链表的头结点。

注：用链表是解决哈希冲突的问题，也就是说不同的 port 会被哈希到同一个 bucket 中，因此需要遍历同一个桶的链表，依次查看链表中节点的 port 是否与所给的 port 相同。

可以看出上面两个结构体非常相似，唯一不同的就在于锁的类型，rwlck_t 是对 spinlock_t 自旋锁的封装，实现了读写自旋锁，允许多个读，但只能一个写。当某个处理器上的内核执行线程申请自旋锁时，如果锁可用，则获得锁，然后执行临界区操作，最后释放锁；如果锁已被占用，线程并不会转入睡眠状态，而是忙等待该锁，一旦锁被释放，则第一个感知此信息的线程将获得锁。

```
typedef struct {
        spinlock_t lock;
        volatile int counter;
#ifdef CONFIG_PREEMPT
        unsigned int break_lock;
#endif
} rwlock_t;
```