并发：基于锁的并发数据结构

邰颖
南京大学
智能科学与技术学院

基于锁的并发数据结构

- 向数据结构添加锁使该结构变得**线程安全**。
  - 如何添加锁决定了数据结构的**正确性**和**性能**。

示例：无锁的计数器

- 计数器是最简单的一种数据结构，无法直接扩展到多线程环境

```
1   typedef struct __counter_t {
2           int value;
3   } counter_t;
4
5   void init(counter_t *c) {
6           c->value = 0;
7   }
8
9   void increment(counter_t *c) {
10          c->value++;
11  }
12
13  void decrement(counter_t *c) {
14          c->value--;
15  }
16
17  int get(counter_t *c) {
18          return c->value;
19  }
```

// 初始化计数器的值为0

//增加计数器的值

//减少计数器的值

竞态条件

//获取当前计数器的值

示例：基于锁的并发计数器

•增加一个单独的锁（Add a single lock），让这段代码线程安全

- 当调用操作数据结构的函数时，需要获取该锁

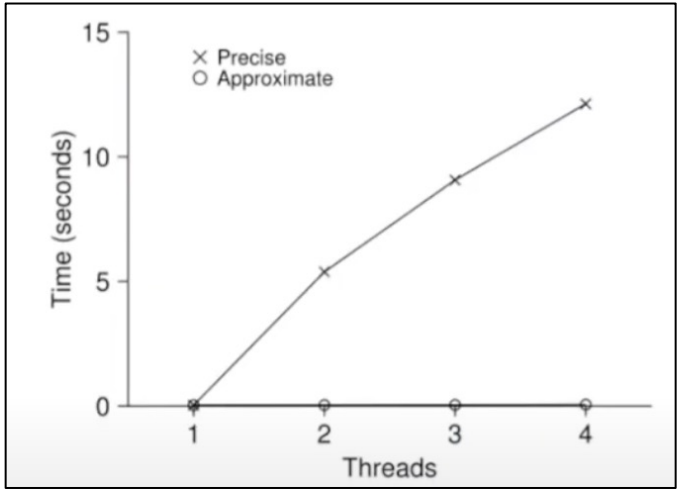
```
1   typedef struct __counter_t {
2           int value;
3           pthread_lock_t lock;
4   } counter_t;
5
6   void init(counter_t *c) {
7           c->value = 0;
8           Pthread_mutex_init(&c->lock, NULL);   // 初始化锁
9   }
10
11  void increment(counter_t *c) {
12          Pthread_mutex_lock(&c->lock);
13          c->value++;                           // 安全地增加计数值
14          Pthread_mutex_unlock(&c->lock);
15  }
16
```

示例：基于锁的并发计数器（续）

```
(Cont.)
17      void decrement(counter_t *c) {
18              Pthread_mutex_lock(&c->lock);
19              c->value--;                          // 安全地减少计数值
20              Pthread_mutex_unlock(&c->lock);
21      }
22
23      int get(counter_t *c) {
24              Pthread_mutex_lock(&c->lock);
25              int rc = c->value;                   // 安全地读取计数值
26              Pthread_mutex_unlock(&c->lock);
27              return rc;
28      }
```

传统计数器方法的性能开销

- 每个线程都更新同一个共享计数器
  - 每个线程更新计数器一百万次
  - 测试环境为：配备4个Intel 2.7GHz i5 CPU的iMac电脑



- ✓ **同步的计数器扩展性不好**：单线程只需要很短的时间（大约0.03s），多个线程并发，性能显著下降
- ✓ 线程越多，性能越差

传统计数器的**可扩展性差**

完美的扩展性

- 我们期望看到的是**完美的扩展性**。
- 即使执行了更多的工作，这些工作也是**并行进行**的。
- 完成任务所花费的总时间**不会增加**。

可扩展的计数器：近似计数器

- 近似计数器：**牺牲少量精确性换取高并发性能：**
  - •使用多个**局部物理计数器**（每个CPU核心一个）来表示一个单一的**逻辑计数器**。
  - •同时维护一个单一的**全局计数器**。
  - •每个计数器均需一个对应的锁：
    - 每个局部计数器有一个锁，全局计数器也有单独一个锁。

- **示例：以一个有4个CPU核心的机器为例：**
  - •拥有4个局部计数器（local counters）。
  - •1个全局计数器（global counter）。

近似计数器的基本思想

- 当一个运行在某个CPU核心上的线程希望增加计数器的值时：
  - 它只需增加该核心上对应的**局部计数器**。
  - 每个CPU核心都拥有各自独立的局部计数器：
    - 不同CPU核心上的线程可以**无竞争地**（without contention）更新局部计数器
    - 因此，这种方式的计数器更新具有良好的**可扩展性（scalable）**
  - 局部计数器的值会定期转移到全局计数器（当某线程需要读取计数器总值时）：
    - 先获得全局锁。
    - 然后用局部计数器的值更新全局计数器。
    - 最后将局部计数器重置为0。

近似计数器的基本思想（续）

- 局部计数器到全局计数器之间的转移频率由阈值 **S**（近似因子，Approximation Factor）决定：

  - **S** 越小：
    - 计数器的行为越接近**不具备可扩展性的传统计数器**。

  - **S** 越大：
    - 计数器的**扩展性（scalable）越好**。

    - 但全局计数值与实际值（actual count）之间的差距也可能更大。

近似计数器示例

- 跟踪近似计数器（Tracing the Approximate Counters）
  - 阈值 S 设置为 5
  - 假设系统有4个CPU核心，每个核心上都运行有线程
  - 每个线程都会更新各自对应的本地计数器（记作 L_1 到 L_4）

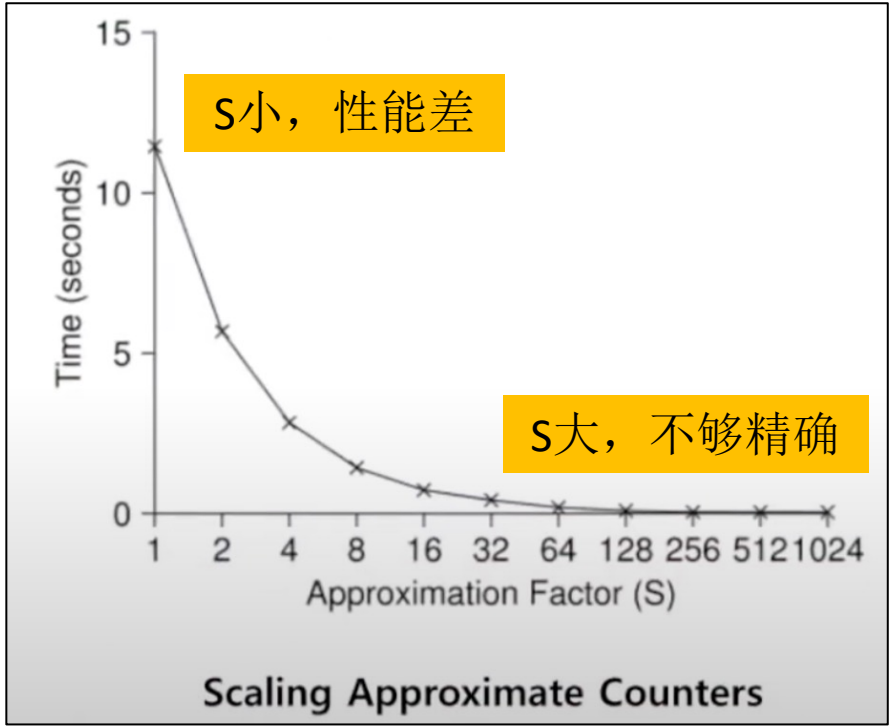
Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

阈值S的重要性

- 在配备4个CPU核心的系统上，每个线程对计数器进行1百万次（1,000,000次）更新：

  - **S较小时**：性能表现**较差**，但全局计数值非常**准确**。

  - **S较大时**：性能表现**非常好**，但全局计数值存在一定**滞后性**，即计数结果不够精确。



近似计数器的实现

```c
typedef struct __counter_t {
    int global;                      // 全局计数器
    pthread_mutex_t glock;           // 全局计数器锁
    int local[NUMCPUS];              // 本地计数器（每个CPU核心一个）
    pthread_mutex_t llock[NUMCPUS];  // 本地计数器对应的锁
    int threshold;                   // 更新频率（阈值）
} counter_t;

// 初始化函数：记录阈值，初始化所有锁和计数器
void init(counter_t *c, int threshold) {
    c->threshold = threshold;        // 设置阈值

    c->global = 0;                   // 初始化全局计数器为0
    pthread_mutex_init(&c->glock, NULL);  // 初始化全局锁

    int i;
    for (i = 0; i < NUMCPUS; i++) {
        c->local[i] = 0;             // 初始化每个CPU核心的本地计数器为0
        pthread_mutex_init(&c->llock[i], NULL); // 初始化每个本地计数器锁
    }
}
```

近似计数器的实现（续）

- 通常只需要获取局部锁并更新局部计数器，一旦局部计数达到阈值，则获取全局锁并将局部值转移到全局

```c
void update(counter_t *c, int threadID, int amt) {
    pthread_mutex_lock(&c->llock[threadID]);  // 获取本地锁
    c->local[threadID] += amt;  // 假设 amt > 0

    if (c->local[threadID] >= c->threshold) {  // 本地计数器达到阈值，转移到全局
        pthread_mutex_lock(&c->glock);  // 获取全局锁
        c->global += c->local[threadID];  // 更新全局计数器
        pthread_mutex_unlock(&c->glock);  // 释放全局锁
        c->local[threadID] = 0;  // 重置本地计数器
    }

    pthread_mutex_unlock(&c->llock[threadID]);  // 释放本地锁
}

// 获取：只返回全局计数值（可能并不完全精确）
int get(counter_t *c) {
    pthread_mutex_lock(&c->glock);  // 获取全局锁
    int val = c->global;  // 获取全局计数值
    pthread_mutex_unlock(&c->glock);  // 释放全局锁
    return val;  // 仅返回近似值
}
```

并发链表：初始化

```c
// 基本节点结构
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

// 基本链表结构（每个链表一个）
typedef struct __list_t {
    node_t *head;
    pthread_mutex_t lock;  // 链表的锁
} list_t;

// 初始化链表：记录阈值，初始化锁，全局计数器和所有本地计数器
void List_Init(list_t *L) {
    L->head = NULL;  // 初始化链表头为空
    pthread_mutex_init(&L->lock, NULL);  // 初始化链表的锁
}
```

并发链表（续）：插入

```
int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);  // 获取链表锁

    node_t *new = malloc(sizeof(node_t));  // 为新节点分配内存
    if (new == NULL) {  // 如果分配失败
        perror("malloc");
        pthread_mutex_unlock(&L->lock);  // 释放锁
        return -1;  // 返回失败
    }

    new->key = key;  // 设置新节点的key
    new->next = L->head;  // 将新节点插入到链表头部
    L->head = new;  // 更新链表头

    pthread_mutex_unlock(&L->lock);  // 释放锁
    return 0;  // 返回成功
}
```

并发链表（续）：查找

```
int List_Lookup(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);  // 获取链表的锁
    node_t *curr = L->head;  // 从链表头开始查找
    while (curr) {
        if (curr->key == key) {  // 找到匹配的key
            pthread_mutex_unlock(&L->lock);  // 解锁
            return 0;  // 成功，返回0
        }
        curr = curr->next;  // 继续查找下一个节点
    }
    pthread_mutex_unlock(&L->lock);  // 解锁
    return -1;  // 未找到，返回-1
}
```

并发链表（续）

•代码在进行插入操作时**获取锁**。

•代码在退出时**释放锁**。

- 如果malloc()失败，代码必须在返回失败前**释放锁**。
- 这种**异常控制流**已经被证明容易出错，可能因为没有及时释放锁导致程序死锁
- 解决方案：**锁的获取和释放仅作用于插入代码中的实际临界区**。

并发链表：插入重写版

```c
void List_Init(list_t *L) {
    L->head = NULL;  // 初始化链表头为空
    pthread_mutex_init(&L->lock, NULL);  // 初始化链表锁
}

void List_Insert(list_t *L, int key) {
    // 不需要同步
    node_t *new = malloc(sizeof(node_t));  // 为新节点分配内存
    if (new == NULL) {  // 如果分配失败
        perror("malloc");
        return;
    }
    new->key = key;  // 设置新节点的key

    // 只锁定临界区
    pthread_mutex_lock(&L->lock);  // 获取锁
    new->next = L->head;  // 将新节点的next指向当前链表头
    L->head = new;  // 更新链表头为新节点
    pthread_mutex_unlock(&L->lock);  // 释放锁
}
```

并发链表：查找重写版

```
int List_Lookup(list_t *L, int key) {
    int rv = -1;  // 初始化返回值为-1（失败）
    pthread_mutex_lock(&L->lock);  // 获取链表锁
    node_t *curr = L->head;  // 从链表头开始遍历
    while (curr) {  // 遍历链表
        if (curr->key == key) {  // 找到匹配的key
            rv = 0;  // 设置返回值为0（成功）
            break;  // 找到后跳出循环
        }
        curr = curr->next;  // 向下一个节点移动
    }
    pthread_mutex_unlock(&L->lock);  // 释放锁
    return rv;  // 返回结果：0表示成功，-1表示失败
}
```

并发链表：扩展链表

- **锁耦合（lock coupling, or Hand-over-hand locking）**
  - •为每个链表节点添加锁，而不是为整个链表添加一个锁。
  - •在遍历链表时：
    - 首先获取下一个节点的锁。
    - 然后释放当前节点的锁。
  - •启用链表操作的高并发度。
    - •然而，实际应用中，**获取和释放每个节点的锁的开销**在遍历链表时是**不划算的**。

并发队列：Michael和Scott并发队列

- 队列中有两个锁：
  - 一个用于队列的**头部**（head）
  - 一个用于队列的**尾部**（tail）
  - 这两个锁的目的是：使**入队（enqueue）和出队（dequeue）**操作能够并发进行

- **添加一个虚拟节点（dummy node）**
  - 在队列初始化代码中分配。
  - 使头部操作和尾部操作能够分开进行。

并发队列（续）：队列初始化

```c
typedef struct __node_t {
    int value;  // 节点值
    struct __node_t *next;  // 下一个节点指针
} node_t;

typedef struct __queue_t {
    node_t *head;  // 队列头指针
    node_t *tail;  // 队列尾指针
    pthread_mutex_t headLock;  // 头部锁
    pthread_mutex_t tailLock;  // 尾部锁
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));  // 分配内存为新节点
    tmp->next = NULL;  // 初始化新节点的next指针为空
    q->head = q->tail = tmp;  // 设置队列的头尾为同一个节点
    pthread_mutex_init(&q->headLock, NULL);  // 初始化头部锁
    pthread_mutex_init(&q->tailLock, NULL);  // 初始化尾部锁
}
```

// 节点结构

// 队列结构

// 这就是 dummy node

- tmp 是 dummy 节点；
- q->head 和 q->tail 最初都指向这个 dummy node；
- 入队操作从 tail->next 插入；
- 出队操作从 head->next 读取。

并发队列（续）：入队

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));          // 动态分配一个新节点
    assert(tmp != NULL);

    tmp->value = value;
    tmp->next = NULL;

    pthread_mutex_lock(&q->tailLock);              // 使用互斥锁对尾部进行保护
    q->tail->next = tmp;                           // 当前尾节点的 next 指向新节点
    q->tail = tmp;                                 // 将 tail 指向新节点，入队完成
    pthread_mutex_unlock(&q->tailLock);
}
```

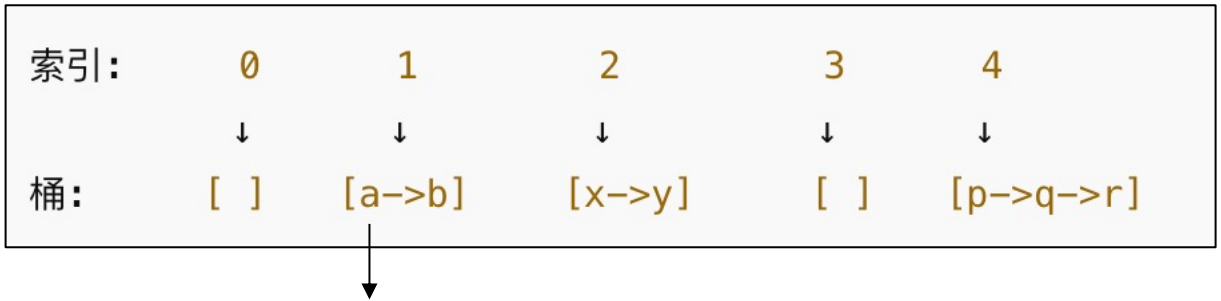
并发队列（续）：出队

```
int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);  // 获取头部锁
    node_t *tmp = q->head;  // 保存队列头节点
    node_t *newHead = tmp->next;  // 获取新的队列头节点
    if (newHead == NULL) {  // 如果队列为空
        pthread_mutex_unlock(&q->headLock);  // 释放头部锁
        return -1;  // 返回-1，表示队列为空
    }
    *value = newHead->value;  // 获取队列头节点的值
    q->head = newHead;  // 更新队列头为新的头节点
    pthread_mutex_unlock(&q->headLock);  // 释放头部锁
    free(tmp);  // 释放旧的头节点内存
    return 0;  // 返回0，表示出队成功
}
```

// 检查队列是否为空

并发哈希表（散列表）

- 关注一个简单的哈希表
  - •哈希表不进行动态扩展
  - •使用并发链表构建
  - •**每个哈希桶使用一个锁**，每个桶由一个链表表示



并发哈希表

```c
#define BUCKETS (101)  // 定义哈希表的桶数为101

typedef struct __hash_t {
    list_t lists[BUCKETS];  // 使用链表数组来表示哈希表的桶
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);  // 初始化每个哈希桶中的链表
    }
}

int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;  // 通过哈希函数确定桶的位置
    return List_Insert(&H->lists[bucket], key);  // 将元素插入到相应的桶中
}

int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;  // 通过哈希函数确定桶的位置
    return List_Lookup(&H->lists[bucket], key);  // 查找相应桶中的元素
}
```

// lists[i] 表示第 i 个桶，桶本身是一个链表

// 遍历数组中的每个桶，调用链表初始化函数

并发哈希表的性能

- 四个线程，每个线程分别执行10,000～50,000次并发更新
  - 在具有四颗Intel 2.7GHz i5 CPU的iMac上测试

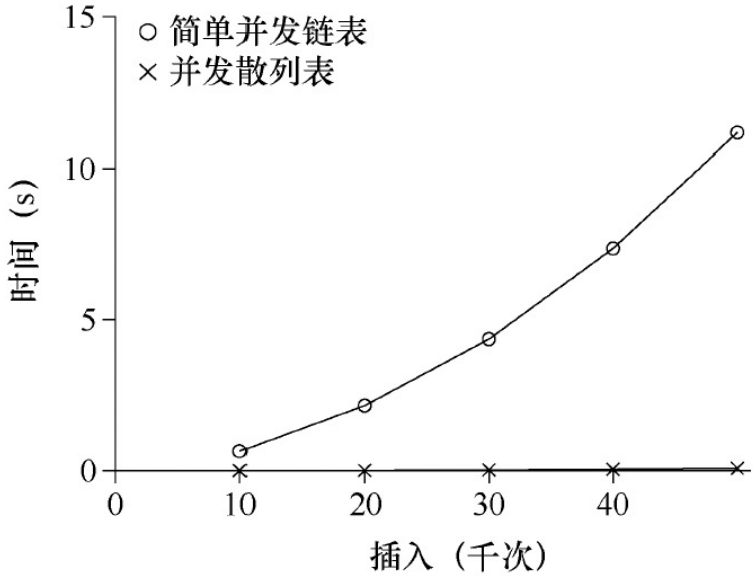


图 29.10　扩展散列表

简单的并发哈希表**扩展性表现优异**

小结

- 介绍了一些代表性的并发数据结构
  - 并发计数器
  - 并发链表
  - 并发队列
  - 并发哈希表