第6章-武技-常见组件后继

时间过得真快, 武技部分修炼完毕, C 初学者进阶的修真之旅的岁月也就过去一半了. 武技本是练气期选手在模拟的妖魔实战中的宝贵经验. 原始而奏效, 飞沙走石, 滔天气浪. 诉述故事也许很短, 但未来的传说由自己在临摹和踩坑中肚子谱写 ~ 此刻一同穿梭那场景, 感受一刹那间神魂震动, 山河破碎, 天地不仁, 人鬼颠鸾

```
~
~ 带好你的剑,
~ 还记得那年的华山剑法?
~ ---:o- 上路吧,
```

6.1 传说中的线程池

线程池这个很老的旧话题, 讨论的不多, 深入还挺难. 也就在一些基础库或编程语言设计中才能见到这类有点精妙复杂的模型技巧. 这里随大流简述简单且控制性强的一种线程池实现. 先引入一个概念"惊群". 举个简单例子. 春天来了, 公园出现了很多麻雀. 而你恰巧有一个玉米渣. 扔出去, 立马无数麻雀过来争抢. 而最终会有一只或多只麻雀会得到. 而那些没有抢到的麻雀会很累 ... .. . 编程中惊群, 同样是个很老的话题. 服务器框架开发中很有机会遇到. 有兴趣的可以自行搜索, 多数介绍的解决方案质量非常高. 这里只讨论线程池中惊群现象. 采用的认可度最高的 POSIX 跨平台的线程库 pthread 来演示和克服.

```
//
// pthread_cond_signal - 通过条件变量激活正在等待的线程
// cond     : 条件变量
// return   : 0 is success, -1 is error, 见 errno
//
extern int __cdecl pthread_cond_signal (pthread_cond_t * cond);

//
// pthread_cond_wait - 线程等待激活
// cond     : 条件变量
// mutex    : 互斥锁
// return   : 0 is success, -1 is error, 见 errno
//
extern int __cdecl pthread_cond_wait (pthread_cond_t * cond,
                                      pthread_mutex_t * mutex);
```

上面 pthread_cond_signal 接口就是线程池中出现惊群现象的原因所在. 他会激活一个或多个 pthread_cond_wait 等待状态线程. 同样我们解决惊群的方法很直接, 定向激活, 每个实际运行线程对象

都有自己的条件变量. 这样每次只要激活需要激活的线程变量就可以了. 惊群现象就自然巧妙的避过去了.

6.1.1 线程池设计

```c
#pragma once

#include "thread.h"

//
// 简易的线程池的库
//
typedef struct threads * threads_t;

//
// threads_create - 创建线程池对象
// return   : 创建的线程池对象, NULL 表示失败
//
extern threads_t threads_create(void);

//
// threads_delete - 删除线程池对象, 同步操作
// pool     : 线程池对象
// return   : void
//
extern void threads_delete(threads_t pool);

//
// threads_insert - 线程池中添加待处理的任务
// pool     : 线程池对象
// frun     : node_f 运行的执行体
// arg      : frun 的参数
// return   : void
//
extern void threads_insert(threads_t pool, void * frun, void * arg);
```

不完全类型 threads_t 就是我们定义的线程池对象, 并有创建删除添加行为. 其中使用函数参数 void * frun 是用于去掉警告. 设计的如果不好, 那还不如不设计, 同样功能越少出错的概率越小越有针对性.

6.1.2 线程池实现

线程池解决的问题是避免创建过多线程对象, 加重操作系统切换的负担. 从而换了一种方式, 将多个线程业务转成多个任务被固定的线程来回处理. 这个思路是牺牲优先级来换取性能. 先说说线程池容易实现的部分, 任务结构的设计.

```c
#include "threads.h"

// struct job 任务链表 结构 和 构造
struct job {
    struct job * next;      // 指向下一个任务结点
    node_f frun;            // 任务结点执行的函数体
    void * arg;
};

inline struct job * job_new(node_f frun, void * arg) {
    struct job * job = malloc(sizeof(struct job));
    job->next = NULL;
    job->frun = frun;
    job->arg = arg;
    return job;
}
```

job_new 创建一个 job 任务结构, 很直白. job::frun(job::arg) 去执行任务. 继续设计线程池结构, 线程对象和线程池对象结构如下:

```c
// struct thread 线程结构体，每个线程一个信号量，定点触发
struct thread {
    struct thread * next;   // 下一个线程对象
    pthread_cond_t cond;    // 线程条件变量
    volatile bool wait;     // true 表示当前线程被挂起
    pthread_t id;           // 当前线程 id
};

// 定义线程池(线程集)定义
struct threads {
    int size;               // 线程池大小，线程体最大数量
    int curr;               // 当前线程池中总的线程数
    int idle;               // 当前线程池中空闲的线程数
    struct thread * thrs;   // 线程结构体对象集
    pthread_mutex_t mutx;   // 线程互斥量
    volatile bool cancel;   // true 表示当前线程池正在 delete
    struct job * head;      // 线程任务链表的链头，队列结构
    struct job * tail;      // 线程任务队列的表尾，后进后出
};

// threads_add - 线程池中添加线程
static void threads_add(struct threads * pool, pthread_t id) {
    struct thread * thrd = malloc(sizeof(struct thread));
    thrd->next = pool->thrs;
    thrd->cond = PTHREAD_COND_INITIALIZER;
    thrd->wait = false;
    thrd->id = id;

    pool->thrs = thrd;
    ++pool->curr;
}

// threads_del - 根据 cond 内存地址删除 pool->thrs 中指定数据
static void threads_del(struct threads * pool, pthread_cond_t * cond) {
    struct thread * head = pool->thrs, * prev = NULL;
    while (head) {
        // 找见了，否则开始记录前置位置
        if (cond == &head->cond) {
            if (prev)
                prev->next = head->next;
            else
                pool->thrs = head->next;
            return free(head);
        }
        prev = head;
        head = head->next;
    }
}

// threads_get - 找到线程 id 对应的条件变量地址
```

```c
static struct thread * threads_get(struct threads * pool, pthread_t id) {
    struct thread * head = pool->thrs;
    while (head) {
        if (pthread_equal(id, head->id))
            break;
        head = head->next;
    }
    return head;
}

// threads_cond - 找到空闲的线程，并返回其信号量
static pthread_cond_t * threads_cond(struct threads * pool) {
    struct thread * head = pool->thrs;
    while (head) {
        if (head->wait)
            return &head->cond;
        head = head->next;
    }
    return NULL;
}
```

通过 struct thread 可以看出线程运行对象中, 都有个 pthread_cont_t 条件变量. 这就是定向激活的关键. struct threads::cancel 用于标识当前线程池是否在销毁阶段. 来避免使用 pthread_cancel + pthread_cleanup_push 和 pthread_cleanup_pop 这类有风险的设计. 由上两个结构衍生了几个辅助行为 threads_del threads_get threads_cond 等. 对于 struct threads 结构中 struct job * head, * tail; 是个待处理的任务队列. struct thread * thrs; 是线程对象的链表. 线程池对象中共用 struct threads 中 mutex 一个互斥量, 方便写代码. 希望还记得前面章节数据结构部分内容, 链表是 C 结构中基础的内丹, 所有代码都是或多或少围绕他这个结构. 要在勤磨练中熟悉提高, 对于刚学习的人. 上面代码其实和业务代码没啥区别, 创建删除添加查找等. 前戏营造的估计够了, 现在开搞其他接口实现.

```c
// THREADS_INT - 开启的线程数是 2 * CPU
#define THREADS_INT        (8)

//
// threads_create - 创建线程池对象
// return   : 创建的线程池对象, NULL 表示失败
//
inline threads_t
threads_create(void) {
    struct threads * pool = calloc(1, sizeof(struct threads));
    pool->size = THREADS_INT;
    pthread_mutex_lock(&pool->mutx, NULL);
    return pool;
}
```

创建接口的实现代码中, calloc 相比 malloc 多调用了 bzero 的相关置零清空操作. 配套还有一个删除释放资源函数. 设计意图允许创建多个线程池对象, 因为有了创建和删除成对操作. 请看下面优雅的删除销毁操作:

```
//
// threads_delete - 尝试异步销毁线程池对象，有些行为是未定义的
// pool     : 线程池对象
// return   : void
//
void
threads_delete(threads_t pool) {
    if (!pool || pool->cancel)
        return;

    // 已经在销毁过程中，直接返回
    pthread_mutex_lock(&pool->mutx);
    if (pool->cancel) {
        pthread_mutex_unlock(&pool->mutx);
        return;
    }

    // 标识当前线程池正在销毁过程中，并随后释放任务列表
    pool->cancel = true;
    struct job * head = pool->head;
    while (head) {
        struct job * next = head->next;
        free(head);
        head = next;
    }
    pool->head = pool->tail = NULL;
    pthread_mutex_unlock(&pool->mutx);

    // 再来销毁每个线程
    struct thread * thrs = pool->thrs;
    while (thrs) {
        struct thread * next = thrs->next;
        // 激活每个线程让其主动退出
        pthread_cond_signal(&thrs->cond);
        pthread_join(thrs->id, NULL);
        thrs = next;
    }

    pthread_mutex_destroy(&pool->mutx);
    // 销毁自己
    free(pool);
}
```

用到很多 pthread api. 不熟悉的多研究, 多做笔记. 不懂的时候说明又是提升功力的机会哈! 对于删除函数, 先监测销毁标识, 后竞争唯一互斥量, 竞争到后就开始释放过程. 先清除任务队列并置空, 随后解锁. 再去准备销毁每个线程, 激活他并等待他退出. 最后销毁自己. 优雅结束了线程池的生成周期. 如果不知道是不是真爱, 那就追求优雅 ~ 如果是真爱, 那么什么都不想要 ~ 随后步入核心部分只有两个函数, 一个是线程轮询处理任务的函数, 另一个是构建线程池函数.

```c
// thread_consumer - 消费线程
static void thread_consumer(struct threads * pool) {
    pthread_t id = pthread_self();
    pthread_mutex_t * mutx = &pool->mutx;

    pthread_mutex_lock(mutx);

    struct thread * thrd = threads_get(pool, id);
    assert(thrd);
    pthread_cond_t * cond = &thrd->cond;

    // 使劲循环的主体，开始消费 or 沉睡
    while (!pool->cancel) {
        if (pool->head) {
            struct job * job = pool->head;
            pool->head = job->next;
            // 队列尾置空监测
            if (pool->tail == job)
                pool->tail = NULL;

            // 解锁，允许其他消费者线程加锁或生产者添加新任务
            pthread_mutex_unlock(mutx);

            // 回调函数，后面再去删除任务
            job->frun(job->arg);
            free(job);

            // 新的一轮开始，需要重新加锁
            pthread_mutex_lock(mutx);
            continue;
        }

        // job 已经为 empty ，开启线程等待
        thrd->wait = true;
        ++pool->idle;

        // 开启等待，直到线程被激活
        int status = pthread_cond_wait(cond, mutx);
        if (status < 0) {
            pthread_detach(id);
            PERR("pthread_cond_wait error status = %d.", status);
            break;
        }
        thrd->wait = false;
        --pool->idle;
    }

    // 到这里程序出现异常，线程退出中，先减少当前线程
    --pool->curr;
    // 去掉这个线程链表 pool->thrs 中对应的数据
```

```
    threads_del(pool, cond);

    // 所有线程共用同一把任务锁
    pthread_mutex_unlock(mutx);
}

//
// threads_insert - 线程池中添加待处理的任务
// pool     : 线程池对象
// frun     : node_f 运行的执行体
// arg      : frun 的参数
// return   : void
//
void
threads_insert(threads_t pool, void * frun, void * arg) {
    if (!frun || !pool || pool->cancel)
        return;

    struct job * job = job_new(frun, arg);
    pthread_mutex_t * mutx = &pool->mutx;

    pthread_mutex_lock(mutx);

    // 线程池中任务队列的插入任务
    if (!pool->head)
        pool->head = job;
    else
        pool->tail->next = job;
    pool->tail = job;

    // 构建线程，构建完毕直接获取
    if (pool->idle > 0) {
        pthread_cond_t * cond = threads_cond(pool);
        // 先释放锁后发送信号激活线程，速度快，缺点丧失线程执行优先级
        pthread_mutex_unlock(mutx);
        // 发送给空闲的线程，这个信号量一定存在
        pthread_cond_signal(cond);
        return;
    }

    if (pool->curr < pool->size) { // 没有，那就新建线程去处理
        pthread_t id;
        if (pthread_create(&id, NULL, (start_f)thread_consumer, pool))
            CERR("pthread_create error curr = %d.", pool->curr);
        else // 添加开启线程的信息
            threads_add(pool, id);
    }

    pthread_mutex_unlock(mutx);
}
```

对于消费者线程 thread_consumer 函数运行起来后, 只有内部出现异常 status == -1 时候会进入自销毁 pthread_detach 操作. 外部 pool->cancel == true 的时候会让其退出, 走 pthread_join 关联接收. 再次想起以前扯的一句闲篇, 关于提升技术好办法

- 1' 多望书
- 2' 多写代码, 搜搜, 问问
- 3' 多看好代码, 临摹源码
- 4' 多创造, 改进, 实战

等该明白的都明白了, 多数会变得那就这样吧. 期待这样的场景重复, 到这里线程池是结束了, 不妨为其写段测试代码 ~

```
#include "times.h"
#include "threads.h"

// old 全局计时器, 存在锁问题
static int old;

// ppt 简单的线程打印函数
static inline void ppt(const char * str) {
    printf("%d => %s\n", ++old, str);
}

// doc 另一个线程测试函数
static inline void doc(void * arg) {
    printf("p = %d, 技术不决定项目的成败! 我老大哭了\n", ++old);
}

void test_threads(void) {
    // 创建线程池
    threads_t pool = threads_create();

    // 添加任务到线程池中
    for (int i = 0; i < BUFSIZ; ++i) {
        threads_insert(pool, ppt, "你为你负责的项目拼命过吗. 流过泪吗");
        threads_insert(pool, doc, NULL);
    }

    // 等待 5s 再结束吧
    usleep(5 * 1000000);

    // 清除当前线程池资源, 实战上线程池是常驻内存, 不要清除.
    threads_delete(pool);
}
```

线程模型有点廉颇老矣, 尚能饭否的味道. 在现代服务业务处理上面, 切换代价, 资源数量(线程栈大小, 线程数量)消耗大. 所以 goroutine 这类模型很吃香. 但如果你足够自信通过设置 CPU 硬亲和性, 有时候会

获得更高性能. 总而言之咱们费了老大劲写了个线程池, 99% 业务基本都不会用到(实战中进程线程协程模型常混搭). 密集型业务目前修真界都流行少量线程加轮询消息队列的方式处理, 下一个主角该登场了 ~

6.2 消息轮序器

服务开发中, 消息轮询器基本上就是整个服务器调度处理的核心! 所有待处理的业务统一封装 push 进去, 单独线程异步 loop 去处理, 周而复始. 等同于守护门派安定的无上大阵. 下面就带大家写个超炫迈的封魔大阵, 收获门派一世繁华 ~ 接口设计部分 loop.h

```
#pragma once

#include "q.h"
#include "thread.h"

typedef struct loop * loop_t;

//
// loop_delete - 轮询对象销毁
// p         : 轮询对象
// return    : void
//
extern void loop_delete(loop_t p);

//
// loop_push  - 消息压入轮询器
// p         : 轮询对象
// m         : 消息
// return    : void
//
extern void loop_push(loop_t p, void * m);

//
// loop_create - 创建轮询对象
// frun      : node_f 消息处理行为
// fdie      : node_f 消息销毁行为
// return    : 轮询对象
//
extern loop_t loop_create(void * frun, void * fdie);
```

函数 loop_create 创建一个消息轮序器并启动, 需要注册两个函数 frun 和 fdie, 前者用于处理每个 push 进来的消息, fdie 用于用户 push 进来消息的善后工作销毁清除操作. 这个库实现的非常精简. 直贴代码, 比较有价值. 多感受其中的妙用, 入戏, 小就不能满足你了吗.

```c
#include "loop.h"

// loop 轮询器结构
struct loop {
    pthread_t id;           // 运行的线程id
    pthread_mutex_t lock;   // 消息切换锁
    pthread_cond_t cond;    // 等待条件变量

    volatile bool wait;     // true 标识正在等待
    volatile bool loop;     // true 线程正在运行

    struct q wq;            // 写消息

    struct q rq;            // 读消息
    node_f frun;            // 消息处理行为
    node_f fdie;            // 消息销毁行为
};

// loop_run 消息处理行为
inline void loop_run(loop_t p, void * m) {
    // 开始处理消息
    p->frun(m);
    p->fdie(m);
}

//
// loop_delete - 轮询对象销毁
// p        : 轮询对象
// return   : void
//
void
loop_delete(loop_t p) {
    if (p == NULL) return;

    //
    // delete 执行必须在 push 之后, C 代码是在刀剑上跳舞 ~
    //
    p->loop = false;
    // 尝试激活信号量
    pthread_cond_broadcast(&p->cond);
    // 等待线程结束, 然后退出
    pthread_end(p->id);

    // 队列内存清理
    q_delete(&p->wq, p->fdie);
    q_delete(&p->rq, p->fdie);
    free(p);
}

//
```

```
// loop_push - 消息压入轮询器
// p        : 轮询对象
// m        : 消息
// return   : void
//
void
loop_push(loop_t p, void * m) {
    assert(p != NULL && m != NULL);

    pthread_mutex_lock(&p->lock);

    q_push(&p->wq, m);

    if (p->wait) {
        p->wait = false;
        pthread_cond_signal(&p->cond);
    }

    pthread_mutex_unlock(&p->lock);
}

// loop_loop 轮询器执行的循环体
static void loop_loop(loop_t p) {
    while (p->loop) {
        void * m = q_pop(&p->rq);
        if (m != NULL) {
            loop_run(p, m);
            continue;
        }

        pthread_mutex_lock(&p->lock);
        // 没有数据开始阻塞
        if (q_empty(&p->wq)) {
            p->wait = true;
            pthread_cond_wait(&p->cond, &p->lock);
        }

        // read q <- write q
        q_swap(p->rq, p->wq);

        pthread_mutex_unlock(&p->lock);
    }
}

//
// loop_create - 创建轮询对象
// frun     : node_f 消息处理行为
// fdie     : node_f 消息销毁行为
// return   : 轮询对象
//
loop_t
```

```
loop_create(void * frun, void * fdie) {
    assert(frun != NULL);

    loop_t p = malloc(sizeof(struct loop));
    if (p != NULL) {
        RETNUL("malloc panic %zu", sizeof(struct loop));
    }

    // 初始化 POSIX 互斥锁和条件变量
    if (pthread_mutex_init(&p->lock, NULL)) {
        free(p);
        RETNUL("pthread_mutex_init panic error");
    }
    if (pthread_cond_init(&p->cond, NULL)) {
        pthread_cond_destroy(&p->cond);
        free(p);
        RETNUL("pthread_cond_init panic error");
    }

    if (q_init(&p->rq) == false) {
        pthread_cond_destroy(&p->cond);
        pthread_mutex_destroy(&p->lock);
        free(p);
        return NULL;
    }
    if (q_init(&p->wq) == false) {
        free(p->rq.data);
        pthread_cond_destroy(&p->cond);
        pthread_mutex_destroy(&p->lock);
        free(p);
        return NULL;
    }
    p->frun = frun;
    p->fdie = fdie;
    p->wait = p->loop = true;

    if (pthread_run(&p->id, loop_loop, p)) {
        free(p->wq.data);
        free(p->rq.data);
        pthread_cond_destroy(&p->cond);
        pthread_mutex_destroy(&p->lock);
        free(p);
        RETNUL("pthread_run panic error");
    }

    return p;
}
```

对于 struct loop::fdie 也支持 NULL 行为操作. 如果 C 编译器层面语法糖支持的好些, 那就爽了. 整体思路是乒乓交换 ~ 感悟至今, 进出妖魔战场更加频繁, 修炼也越发坚深, 然而心境中域外天魔也逐渐在另一个次元逼近而来. 他会拷问你的内心, 你为何修炼编程? 随之进入弥天幻境, 太多太多人在幻境的路中间 ~ 不曾解脱 ~ 不愿走过那条大道, 去, 元婴终身无望 ~

6.3 阅读理解 pthread_mutex_lock

本文阅读理解抽自 glibc 库中 pt-mutext-lock.c 部分. 非常清晰讲述互斥锁大致原理. 首先要知道互斥锁提供那些功能, PT_MTX_NORMAL, PT_MTX_RECURSIVE, PT_MTX_ERRORCHECK, PTHREAD_MUTEX_ROBUST 业务语义. 然后是 lll_lock, lll_robust_lock 借助 atomic 和 syscall __NR_futex 等标准和系统能力. 经过之前几章训练感兴趣同学, 完全有能力看懂这些用户线程级别代码. 系统开发宝藏就是**最新的 glibc 库, 然后是 man 手册**. 有一说一这些成名库, 起的变量名确实**言简意赅, 一看就懂**.

```
/* pthread_mutex_lock.  Hurd version.
   Copyright (C) 2016-2021 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library;  if not, see
   <https://www.gnu.org/licenses/>.  */

#include <pthread.h>
#include <stdlib.h>
#include <assert.h>
#include <pt-internal.h>
#include "pt-mutex.h"
#include <hurdlock.h>

int
__pthread_mutex_lock (pthread_mutex_t *mtxp)
{
  struct __pthread *self;
  int flags = mtxp->__flags & GSYNC_SHARED;
  int ret = 0;

  switch (MTX_TYPE (mtxp))
    {
    case PT_MTX_NORMAL:
      lll_lock (mtxp->__lock, flags);
      break;

    case PT_MTX_RECURSIVE:
      self = _pthread_self ();
      if (mtx_owned_p (mtxp, self, flags))
        {
          if (__glibc_unlikely (mtxp->__cnt + 1 == 0))
            return EAGAIN;

          ++mtxp->__cnt;
          return ret;
        }

      lll_lock (mtxp->__lock, flags);
```

```
      mtx_set_owner (mtxp, self, flags);
      mtxp->__cnt = 1;
      break;

    case PT_MTX_ERRORCHECK:
      self = _pthread_self ();
      if (mtx_owned_p (mtxp, self, flags))
        return EDEADLK;

      lll_lock (mtxp->__lock, flags);
      mtx_set_owner (mtxp, self, flags);
      break;

    case PT_MTX_NORMAL | PTHREAD_MUTEX_ROBUST:
    case PT_MTX_RECURSIVE | PTHREAD_MUTEX_ROBUST:
    case PT_MTX_ERRORCHECK | PTHREAD_MUTEX_ROBUST:
      self = _pthread_self ();
      ROBUST_LOCK (self, mtxp, lll_robust_lock, flags);
      break;

    default:
      ret = EINVAL;
      break;
    }

  return ret;
}

hidden_def (__pthread_mutex_lock)
strong_alias (__pthread_mutex_lock, _pthread_mutex_lock)
weak_alias (__pthread_mutex_lock, pthread_mutex_lock)
```

年轻人的故事

从前，有位乡下青年，读了点书，嫌乡村的生活单调，决定要去城里闯世界。临走时，他向村中的村长请教，村长给了他三个字的忠告：“不要怕”。并讲好等他回来时还有另三个字相赠。

30 年后，饱经风霜的青年带着满头白发决定还是回乡村生活。回来时，得知当年的村长已死，心中怅然若失，不知村长另外要赠的三字是什么。村长的儿子转交给他一个信封，说是长者临死前嘱咐交给他的。信只有三个字：“不要悔”。

6.4 展望

好快好快, 修炼之路已经走过小一半. 从华山剑法练起, 到现在的一步两步三步. 此刻自己可以出门踏草原, 风和日丽. 遇到那些业务中的小妖精, 分分钟可以干掉了吧. loop 日月轮在实战中会用的最多. 对于定时器, 多数内嵌到主线程轮询模块(update) 中. 恭喜同行, 同行历练求索, 在血与歌中感受生的洗礼. 尝悟心中的道. 此景想起宋代大文豪的一首喜悦 ❀

元日(宋-王安石)
爆竹声中一岁除，春风送暖入屠苏。
千门万户曈曈日，总把新桃换旧符。