

Chfs

前言

Chfs是大三上CSE的课程大作业，主要内容是写一个inode-based文件系统，然后实现一些功能，比如Crash Consistency、RPC&Lock Server，之后再使用Raft协议将它扩展成分布式文件系统，再实现MapReduce功能统计单词数。总之课程大作业还是延续了IPADS开的课的一贯硬核特性，让我喜提无数个code rush的不眠之夜😓

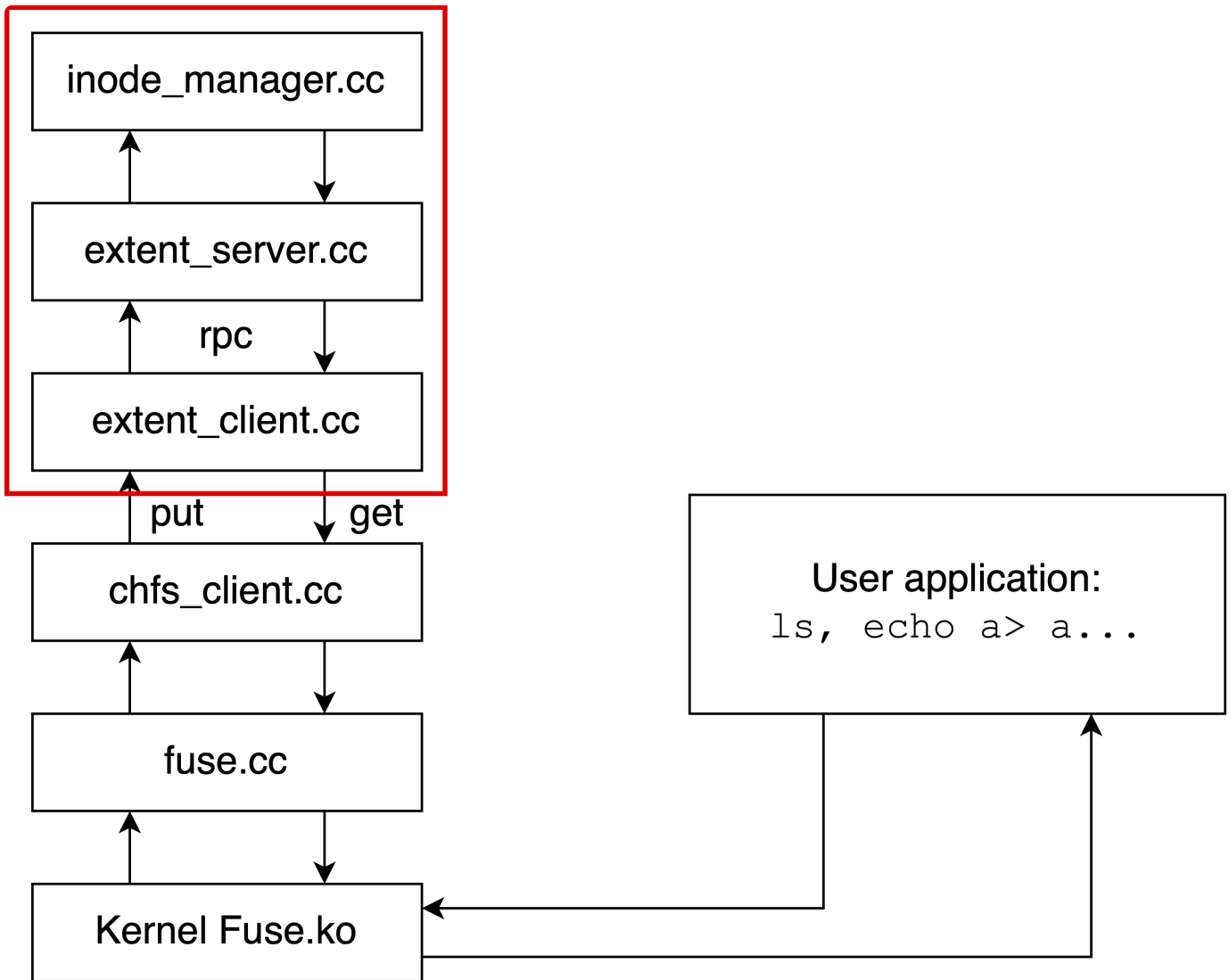
这几个Lab中最有意思的是Lab3 Raft，即实现Raft协议去将原来的inode-based文件系统扩展成分布式文件系统。Lab3结结实实让我体验了一些什么叫做分布式系统和并发编程，bug的隐蔽性直接上了两个数量级（有时候run 十次才出一次错误，薛定谔的bug），究极折磨...不过自己动手实现Raft这种经典的分布式协议还是相当有成就感且锻炼人的，即使我当时疯狂借鉴他人代码，在写完之后我也感觉自己对于分布式系统有了一个更为底层的认识（现在就是后悔当时做了copy cat）

现在复习一遍项目，也顺便把ACID、CAP、Paxos/Raft这些东西捡起来吧，弥补去年的遗憾，也为自己即将到来的面试、往后的职业生涯争取更多胜算。

再次祝愿offer多多，也祝所有看到这篇文章的人offer多多🙏🙏🙏

Lab1 Basic File System

Lab1为实现inode-based文件系统的基本功能，如create、read、write、mkdir，架构如下：



其中FUSE是一个用户自定义文件系统(Implementing filesystems in user space)

用户在前台对FUSE文件系统挂载目录下的文件进行操作，触发系统调用，系统调用会调用用户态文件系统的相关操作完成操作，并将结果通过内核返回给用户

下面的链接是有关FUSE文件的简单介绍：



5分钟搞懂用户空间文件系统FUSE工作原理

fuse是什么？ FUSE, implementing filesystems in user space, 在用户空间实现文件系统。简单讲， ...
知乎专栏

`Kernel Fuse.ko` : kernel中的FUSE模块

`fuse.cc` : 用户实现的FUSE用户态接口，接收来自kernel的请求，将请求转化成chfs调用

`chfs_client.cc` : Chfs的相关文件系统接口

`extent_client.cc` : 块存储的提供者(Block Provider)

`extent_server.cc` : 块存储具体操作的执行者(CREATE/GETATTR/PUT/GET/REMOVE)

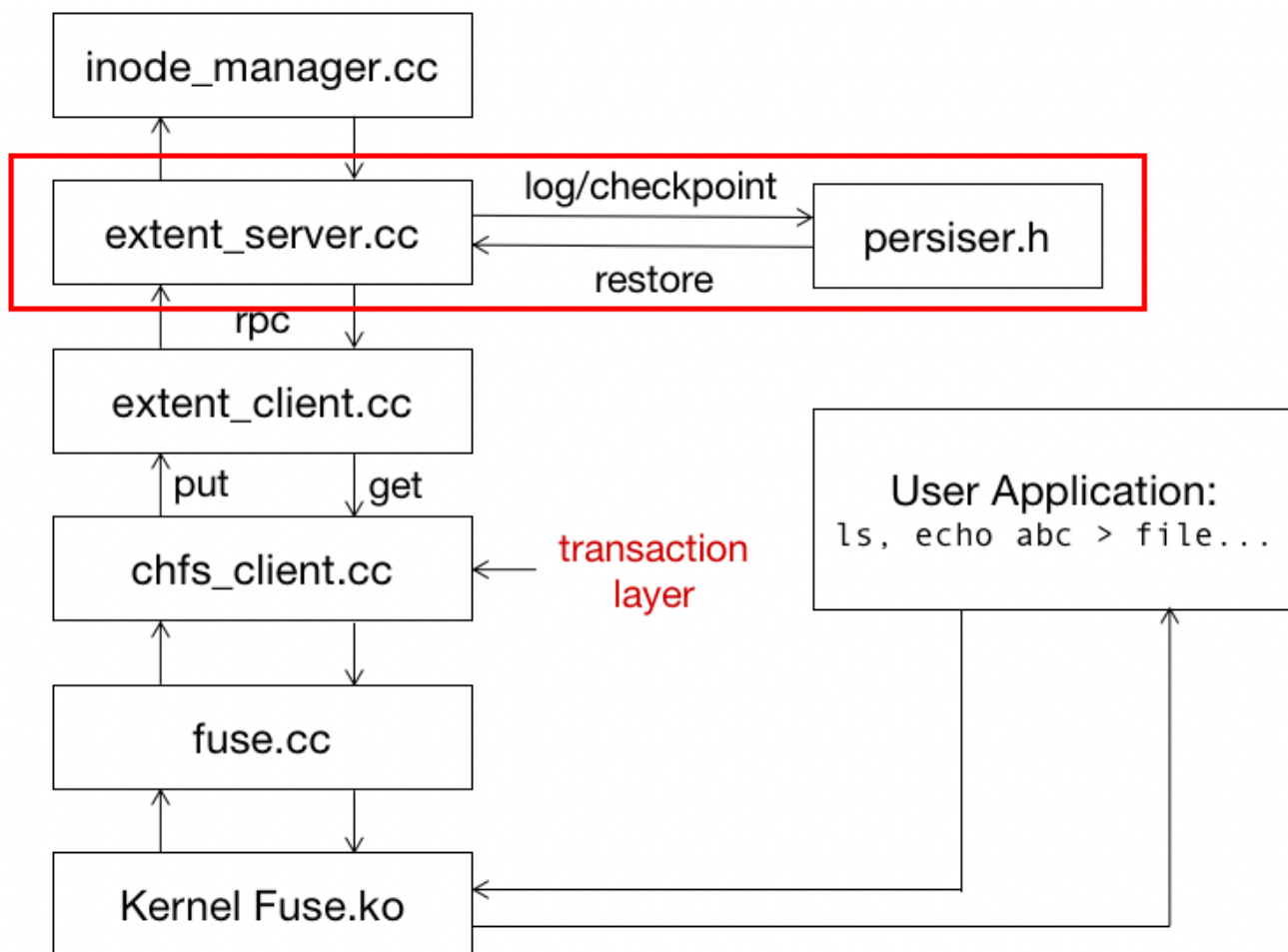
`inode_manager.cc` : 实现了inode层的`alloc_inode`、`free_inode`、`read_file`、`write_file`、`remove_file`、`get_attr`供 `extent_server.cc` 调用

Lab2 Atomicity

Lab2基于Lab1, 将Lab1中的inode-based文件系统扩展成transactional文件系统, 使得其能够在Crash之后恢复文件系统的内容, 并保证Consistency

log/checkpoint/restore功能在新的persistence模块中实现, transaction layer为 `chfs_client`

新的架构图如下图所示:



这里的log使用的是redo-only log, 因为我们只是需要在crash的情况下保证consistency, 在运行时并不会发生transaction abort进而需要回滚的情况(这种情况下需要undo-redo log)。配合上checkpoint, 我们便能够实现Crash Consistency了。

下面我们分析一下redo-only log、undo-only log和undo-redo log

Redo-only Log

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] - amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" + new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

Question: What is the commit point of this transaction? (transfer)

- The line after `log.append(commit_log).sync()`

规则：在Apply所有更改之前需要先写log，写完log之后commit，然后再apply更改

在上图中，我们将转账后的账户余额缓存到了 `new_a` 和 `new_b` 中，然后写入日志 `commit_log`，并在写完日志后commit，然后将余额缓存 `new_a` 和 `new_b` 写入到 `record[a]` 和 `record[b]` 之中。



同时，redo-only log的每一条日志叫 `Log entry`，代表一整个transaction的log

在Systems Crash之后，恢复状态的规则如下：

1. Travel from start to end.
2. Re-apply the updates recorded in a complete log entry.

Redo Log的优点：

1. The commit is extremely efficient: only one file append operations(w/ updated data)

Redo Log的缺点：

1. **Wastes of disk I/O**: all disk operations must happen at the commit point.
2. **All updates must be buffered in the memory until the transaction commits.**(前置条件)

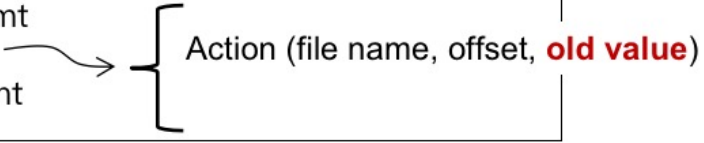
3. The log file is continuously growing while most its updates are already flushed to the disk.

Undo-only Log

解决先前redo-only log的缺陷，让 `uncommitted values` 可以直接被写入磁盘，以：

1. 在commit point之前释放内存空间
2. 最大化利用disk I/O

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    log.append(...).sync()
    records[a] = records[a] - amt
    log.append(...).sync()
    records[b] = records[b] + amt
```



规则：在更新数据之前，将undo log record写入到日志文件中，同时数据直接写入磁盘(加sync操作)

(Should contain sufficient information to undo uncommitted trasactions)

要注意，Undo-only Log也需要commit，用于判断是否需要回滚，以上的例子中没有给出来

事实上，Undo-only Log很少被使用，因为：

1. 在execution过程中比undo-redo log要慢很多，因为强制数据刷盘，随机读写性能开销大
2. 在recovery过程中也没比Undo-redo Log快多少

Undo-redo Log

我们是否需要redo entry取决于对record的更改是否直接写入磁盘（e.g. sync、OS调度策略）

事实上是需要的，原因：

1. 可能会有Page Cache，对record的更改可能并不会直接落盘，undo log的正确性依赖于对record的更改是否直接落盘
2. redo entry的写入方式是顺序写入，相比于undo log强制record刷盘的随机写入要快很多，时间性能要更好

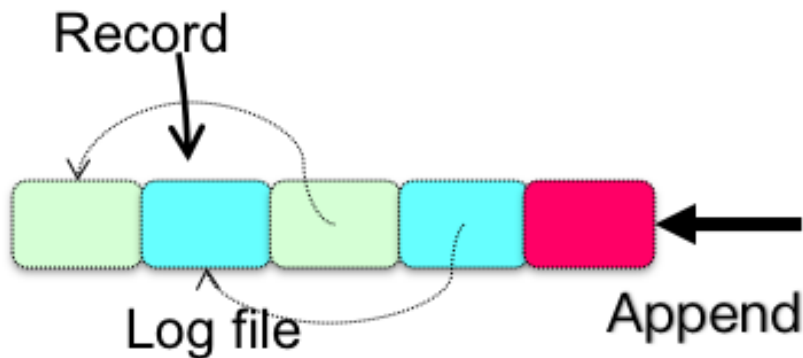
因此我们可以对日志结构进行优化，加入redo entry（`new values`，拆成了好几部分）

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    log.append(...).sync()
    records[a] = records[a] - amt
    log.append(...).sync()
    records[b] = records[b] + amt
```

→ { Action (file name, offset, **old & new values**)

log.append("TX {id} commit").sync()

规则：在更新数据之前，将redo-undo log record写入到日志文件中



Undo-redo Log的每一条日志叫做 `log record`，只包含transaction中一部分的日志内容

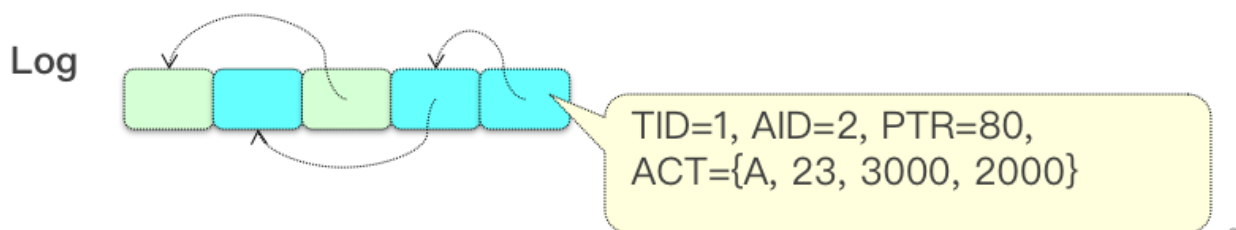
相比于redo-only log的 `log entry`，它有以下特点：

1. Containing the updates of a single action.
2. Log records from different transaction(TX) may possibly interleave. (因此需要一个前向的指针，指向同一个transaction的前一个 `log record`)

Log Record的结构如下图所示：

Each log record consists of

1. Transaction ID
2. Action ID
3. Pointer to previous record in this transaction
4. Action (file name, offset, old & new value)
5. ...



6

Undo-redo log在System Crash之后恢复状态的方法：

1. Travel from end to start.
2. Mark all transaction's log record w/o CMT log and append ABORT log.
3. UNDO ABORT logs from end to start.
4. REDO CMT logs from start to end.

相比于redo-only log, undo-redo log的优势在于：

1. 无前置条件, redo-only log要求所有的数据在提交之前都在内存中, 对内存资源占用大, 且浪费I/O资源

相比于redo-only log, undo-redo log的劣势在于：

1. 磁盘写次数多, redo-only log只需要一次
2. Recover的时候要扫log好几次, 而redo-only log只需要一次

Checkpoint

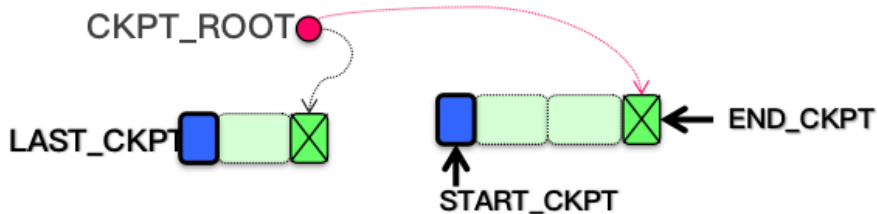
当日志文件过大时, 我们可以使用checkpoint技术来缩小log文件的大小

7

How to checkpoint?

actions

1. Wait till no transactions are in progress
2. Write a **CKPT** record to log
 - Contains a list of all transaction in process and pointers to their most recent log records
3. Save all files
4. Atomically save checkpoint by updating checkpoint root to new CKPT record



76

Lab3 Raft

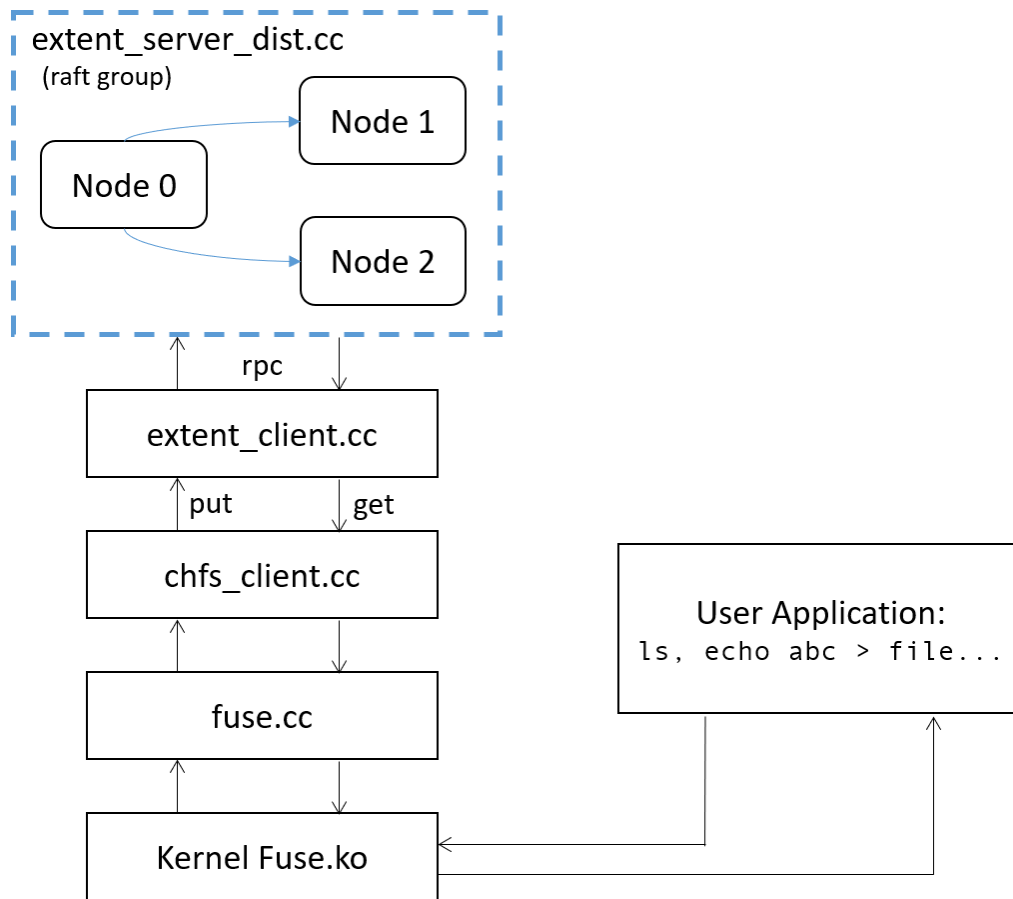
Lab3基于Lab1, 需要将文件系统扩展为基于Raft协议的高可用、高容错、强一致性分布式文件系统

使用这个文件系统的用户只需考虑实现单节点的状态机(e.g. a single node extent server)

这个Lab分为四个部分:

1. Part1: Leader election & Heart beat mechanism
2. Part2: Log replication protocol
3. Part3: Persist logs on the non-volatile storage to tolerate machine crashes.
4. Part4: Snapshot mechanism of Raft
5. Part5: Extend filesystem implemented in lab1 to distributed filesystem based on Raft Library.

具体的架构如图所示:



Code Structure

1. raft_command

`raft_command` 是基类，里面的函数都是虚函数（部分是纯虚函数），用于序列化反序列化log。

▼ raft_command基类

C++

```
1 class raft_command {
2 public:
3     virtual ~raft_command();
4
5     // These interfaces will be used to persistent the command.
6     virtual int size() const = 0;
7     virtual void serialize(char* buf, int size) const = 0;
8     virtual void deserialize(const char* buf, int size) = 0;
9 };
```

2. raft_state_machine

`raft_state_machine` 也是基类，代表Raft中的状态机，最主要的功能有apply log和snapshot

```
1 class raft_state_machine {  
2 public:  
3     virtual ~raft_state_machine();  
4  
5     // Apply a log to the state machine.  
6     virtual void apply_log(const raft_command &cmd) = 0;  
7  
8     // Generate a snapshot of the current state.  
9     virtual std::vector<char> snapshot() = 0;  
10    // Apply the snapshot to the state machine.  
11    virtual void apply_snapshot(const std::vector<char>&) = 0;  
12 };
```

3. raft

`raft` 类表示一个Raft server，即分布式文件系统中的节点。

该类使用了两个类模版参数，`state_machine` 和 `command`，以和replicated state machine解耦，这样用户就可以实现自己的 `state_machine` 和 `command`。

```

1  template<typename state_machine, typename command>
2  class raft {
3  public:
4      raft(
5          rpcs* rpc_server,
6          std::vector<rpcc*> rpc_clients,
7          int idx,
8          raft_storage<command> *storage,
9          state_machine *state
10     );
11     ~raft();
12
13     // start the raft node.
14     // Please make sure all of the rpc request handlers have been register
15     ed before this method.
16     void start();
17
18     // stop the raft node.
19     // Please make sure all of the background threads are joined in this m
20     ethod.
21     // Notice: you should check whether is server should be stopped by cal
22     ling is_stopped().
23     // Once it returns true, you should break all of your long-run
24     ning loops in the background threads.
25     void stop();
26
27     // send a new command to the raft nodes.
28     // This method returns true if this raft node is the leader that succe
29     ssfully appends the log.
30     // If this node is not the leader, returns false.
31     bool new_command(command cmd, int &term, int &index);
32
33     // returns whether this node is the leader.
34     bool is_leader(int &term);
35
36     // save a snapshot of all the applied log.
37     bool save_snapshot();
38 }

```

4. raft_storage

`raft_storage` 是用来持久化Raft log和metadata的类，接收类模版参数 `command`

```

1  template<typename command>
2  class raft_storage {
3  public:
4      raft_storage(const std::string &file_dir);
5  }

```

实现细节

我们实现的Raft协议是Asynchronous的，所有任务线程都是并行的，Raft Server之间的消息传递通过RPC进行

后台任务线程一直运行，定期执行(通过random timer设置时间间隔):

1. `run_background_election` : 每个Raft Server中的Leader Election任务线程，在一段时间内如果没有收到来自Leader的heartbeat或者是其它RPC Call，则进入到Leader Election阶段
2. `run_background_ping` : Leader的heartbeat任务线程，定期向Followers发送信息维持 authority
3. `run_background_commit` : Leader的commit任务线程，定期向Followers发送Leader的logs
4. `run_background_apply` : 每个Raft Server中的Log Replication任务线程，定期将log持久化到硬盘中

绑定后台任务线程

```

1  template<typename state_machine, typename command>
2  void raft<state_machine, command>::start() {
3      RAFT_LOG("start");
4      this->background_election = new std::thread(&raft::run_background_election, this);
5      this->background_ping = new std::thread(&raft::run_background_ping, this);
6      this->background_commit = new std::thread(&raft::run_background_commit, this);
7      this->background_apply = new std::thread(&raft::run_background_apply, this);
8      ...
9  }

```

此外RPC调用是Async的，这是因为Lab提供了RPC Call/Handle提供了先进先出的任务队列 `ThrPool` 类，以及为每一个RPC Call注册好了Handler函数

```

1  class ThrPool {
2
3
4      public:
5          struct job_t {
6              void *(*f)(void *); //function point
7              void *a; //function arguments
8          };
9
10         ThrPool(int sz, bool blocking=true);
11         ~ThrPool();
12         template<class C, class A> bool addObjJob(C *o, void (C::*m)(A),
13         A a);
14         template<class C, class A0, class A1> bool addObjJob(C *o, void
15         (C::*m)(A0, A1), A0 a0, A1 a1);
16         template<class C, class A0, class A1, class A2> bool addObjJob(C *
17         o, void (C::*m)(A0, A1, A2), A0 a0, A1 a1, A2 a2);
18         template<class C, class A0, class A1, class A2, class A3> bool add
19         ObjJob(C *o, void (C::*m)(A0, A1, A2, A3), A0 a0, A1 a1, A2 a2, A3 a3);
20
21         bool takeJob(job_t *j);
22
23         void destroy();
24     private:
25         pthread_attr_t attr_;
26         int nthreads_;
27         bool blockadd_;
28         bool stopped;
29
30         fifo<job_t> jobq_;
31         std::vector<pthread_t> th_;
32
33         bool addJob(void *(*f)(void *), void *a);
34 };

```

Async RPC Call的方式:

```

1  thread_pool->addObjJob(this, &raft::your_method, arg1, arg2);

```

Raft协议

High Level Approach: problem decomposition

1. Leader Election
 - a. Select one server as the leader.
 - b. Detect crashes, choose new leader.
2. Log Replication(normal operation)
 - a. Leader accepts commands from client, append to its log.
 - b. Leader replicates its log to other servers (overwrite inconsistencies)
3. Safety
 - a. Keep logs consistent
 - b. Only servers with **up-to-date** logs can become the leader.

Three roles in Raft

1. Leader: Handles all client interactions, log replication (At most 1 at the same time).
2. Follower: Passive (Only responds to incoming RPCs)
3. Candidate: Used to elect a new leader.

Lab4 MapReduce

很简单的单词统计功能，为数不多能够轻松写出来的Lab...

具体细节就不进行赘述了。