转载：http://blog.csdn.net/ywy463726588/article/details/42742355



[ceph源码分析之线程介绍](http://blog.csdn.net/ywy463726588/article/details/42742355)

Ceph是一款优秀的分布式存储软件，通过学习它的源码，我们可以学到很多编程技巧，ceph中关于多线程的编程，是一个很好的学习对象。

Ceph中线程的在src/common/Thread.h中定义

**[cpp]** [view plain](http://blog.csdn.net/ywy463726588/article/details/42742355) [copy](http://blog.csdn.net/ywy463726588/article/details/42742355)

1. **class** Thread {
2. **private**:
3. pthread\_t thread\_id;
4. **public**:
5. Thread(**const** Thread& other);
6. **const** Thread& operator=(**const** Thread& other);
7. Thread();
8. **virtual** ~Thread();
9. **protected**:
10. **virtual** **void** \*entry() = 0;//子类需要实现此函数，作为线程入口
11. **private**:
12. **static** **void** \*\_entry\_func(**void** \*arg);
13. **public**:
14. **const** pthread\_t &get\_thread\_id();
15. **bool** is\_started();
16. **bool** am\_self();
17. **int** kill(**int** signal);//杀掉自己
18. **int** try\_create(**size\_t** stacksize);
19. **void** create(**size\_t** stacksize = 0);//创建线程，里面传入了entry函数
20. **int** join(**void** \*\*prval = 0);//阻塞自己等待线程结束，回收资源
21. **int** detach();//父子线程分离
22. };

C开发者都知道线程编程接口，一个线程在创建时调用pthread\_create函数来传入entry函数，杀死线程调用pthread\_kill函数，当线程被杀死之后，必须调用pthread\_join函数来进行线程资源的回收，如果不调用此函数，就会出现类似zombie process。如果要想让系统自己回收线程资源，就要将线程与父线程分离即调用pthread\_detach.通过接口对比，我们发现src/common/Thread.h中定义的class thread，实际上是对线程代码ceph自己实现的一层封装。

Ceph中所有要用的线程必须继承Thread类，通过查找发现如下一些线程：

**[cpp]** [view plain](http://blog.csdn.net/ywy463726588/article/details/42742355) [copy](http://blog.csdn.net/ywy463726588/article/details/42742355)

1. Accepter.h (src\msg):**class** Accepter : **public** Thread     //用来socket bind的线程,      accepter线程入口函数里定义了poll的网络通讯结构，用来放入管道
2. Admin\_socket.h (src\common):**class** AdminSocket : **public** Thread
3. Ceph\_context.cc (src\common):**class** CephContextServiceThread : **public** Thread
4. DispatchQueue.h (src\msg):  **class** DispatchThread : **public** Thread   //用来进行消息分发的线程，  在simpleMessenger中有dispatch\_queue成员变量,
5. FileJournal.h (src\os):  **class** Writer : **public** Thread     //用来进行写数据到journal中的线程
6. FileJournal.h (src\os):  **class** WriteFinisher : **public** Thread   //当用aio异步模式写数据到journal完成后，此线程用来接管其他剩余操作
7. FileStore.h (src\os):  **struct** SyncThread : **public** Thread    //用来同步数据执行同步的线程，主要是将已经完成的journal的序列号写入到文件中
8. Finisher.h (src\common):  **struct** FinisherThread : **public** Thread   //公用的finisher线程，用来查看某些特定的操作是否结束，结束后进行后续处理工作
9. MDLog.h (src\mds):  **class** ReplayThread : **public** Thread
10. OSD.h (src\osd):  **struct** T\_Heartbeat : **public** Thread   //维系osd进程之间互相心跳连接的线程
11. OutputDataSocket.h (src\common):**class** OutputDataSocket : **public** Thread
12. Pipe.h (src\msg):    **class** Reader : **public** Thread   //用来处理所有对socket的读操作，由acepter线程将socket accept以后打入到SimpleMessenger::dispatch\_queue中交由此线程处理
13. Pipe.h (src\msg):    **class** Writer : **public** Thread   //用来处理所有对socket的写操作，由acepter线程将socket accept以后打入到SimpleMessenger::dispatch\_queue中交由此线程处理
14. Pipe.h (src\msg):    **class** DelayedDelivery: **public** Thread    //用来处理所有对socket的延时操作
15. Signal\_handler.cc (src\global):**struct** SignalHandler : **public** Thread
16. SimpleMessenger.h (src\msg):  **class** ReaperThread : **public** Thread //用来进行消息通信的主要线程 reaper是用来在通讯完成时拆除管道，其中成员有accepter线程（用来bind，accept socket文件放入管道），还有dispatch\_queue线程
17. Throttle.cc (src\test\common):  **class** Thread\_get : **public** Thread
18. Timer.cc (src\common):**class** SafeTimerThread : **public** Thread
19. WorkQueue.h (src\common):  **struct** WorkThread : **public** Thread

可以将这些线程分为四类线程

第一类是普通类线程：

使用此类线程类直接申明继承自Thread，重写一个entry函数，在进程启动最初时，调用了create函数创建了线程，同时使用它的人必须自己定义消息队列。上面大部分线程都是此类，比如FileJournal::write\_thread就是一个FileJournal::Writer类对象，它自己定义了消息队列FileJournal::writeq

第二类是SafeTimerThread类线程:

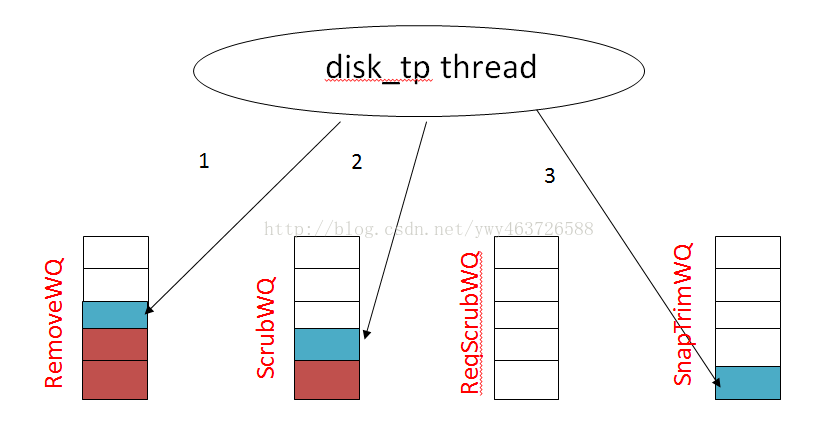
此类线程使用者可以直接申明一个SafeTimer成员变量，因为SafeTimer中已经封装了SafeTimerThread类和一个消息队列（成员是Context回调类），并完成了entry函数的逻辑流程。使用者使用方法，就是设置回调函数，通过SafeTimer::add\_event\_after函数将钩子埋入，等待规定时间到达后执行。

第三类是FinisherThread类线程:

此类线程使用者可以直接申明一个Finisher成员变量，因为Finsher中已经封装了FinisherThread类和一个消息队列（成员是Context回调类），并完成entry函数的逻辑流程。使用者使用方法，就是设置回调函数，通过Finisher::queue函数将钩子埋入，等待某类操作完成后执行。

第四类是ThreadPool内部线程：

这类线程由于是具体工作类线程，所以他们一般都是以线程池形式一下创建多个。ThreadPool类内部有多个线程set<WorkThread\*>和多个消息队列vector<WorkQueue\_\*>组成。工作流程就是线程不断的轮询从队列中拿去数据进行操作（如下图）。



先说ThreadPool中的线程是WorkThread类型，它的入口函数是ThreadPool::worker函数，此函数内定义了WorkThread类线程的操作逻辑。基本流程就是轮询所有WorkQueue\_，当发现某种类型WorkQueue\_中有数据时拿出，然后依次调用该WorkQueue\_自己定义的函数\_void\_process和\_void\_process\_finish等函数来顺序执行操作。

**[cpp]** [view plain](http://blog.csdn.net/ywy463726588/article/details/42742355) [copy](http://blog.csdn.net/ywy463726588/article/details/42742355)

1. **void** ThreadPool::worker(WorkThread \*wt)
2. {
3. ……
4. **while** (!\_stop) {
5. ……
6. **if** (!\_pause && !work\_queues.empty()) {
7. WorkQueue\_\* wq;
8. **int** tries = work\_queues.size();
9. **bool** did = **false**;
10. **while**(tries--) {
11. last\_work\_queue++;
12. last\_work\_queue %= work\_queues.size();
13. wq = work\_queues[last\_work\_queue];
14. **void** \*item = wq->\_void\_dequeue();
15. **if** (item) {
16. ……
17. wq->\_void\_process(item,tp\_handle);
18. ……
19. wq->\_void\_process\_finish(item);
20. ……
21. }
22. ……
23. }
24. ……
25. }
26. ……
27. }

再说一说ThreadPool中的WorkQueue\_,这是一种抽象的类，只定义了一个队列应该有的一些特定的函数，这些函数几乎都是虚函数，目的是为了调用到自己三个子类BatchWorkQueue,WorkQueueVal,WorkQueue自己定义的函数。而在三个子类中对应函数\_void\_process,\_void\_process\_finish中又分别调用了使用者自己继承它们而自己实现的具体操作函数如\_process,\_process\_finish。

**[cpp]** [view plain](http://blog.csdn.net/ywy463726588/article/details/42742355) [copy](http://blog.csdn.net/ywy463726588/article/details/42742355)

1. **struct** WorkQueue\_ {
2. string name;
3. **time\_t** timeout\_interval, suicide\_interval;
4. WorkQueue\_(string n, **time\_t** ti, **time\_t** sti)
5. : name(n), timeout\_interval(ti), suicide\_interval(sti)
6. { }
7. **virtual** ~WorkQueue\_() {}
8. **virtual** **void** \_clear() = 0;
9. **virtual** **bool** \_empty() = 0;
10. **virtual** **void** \*\_void\_dequeue() = 0;
11. **virtual** **void** \_void\_process(**void** \*item, TPHandle &handle) = 0;
12. **virtual** **void** \_void\_process\_finish(**void** \*) = 0;
13. };

最后说一下使用者如何使用：

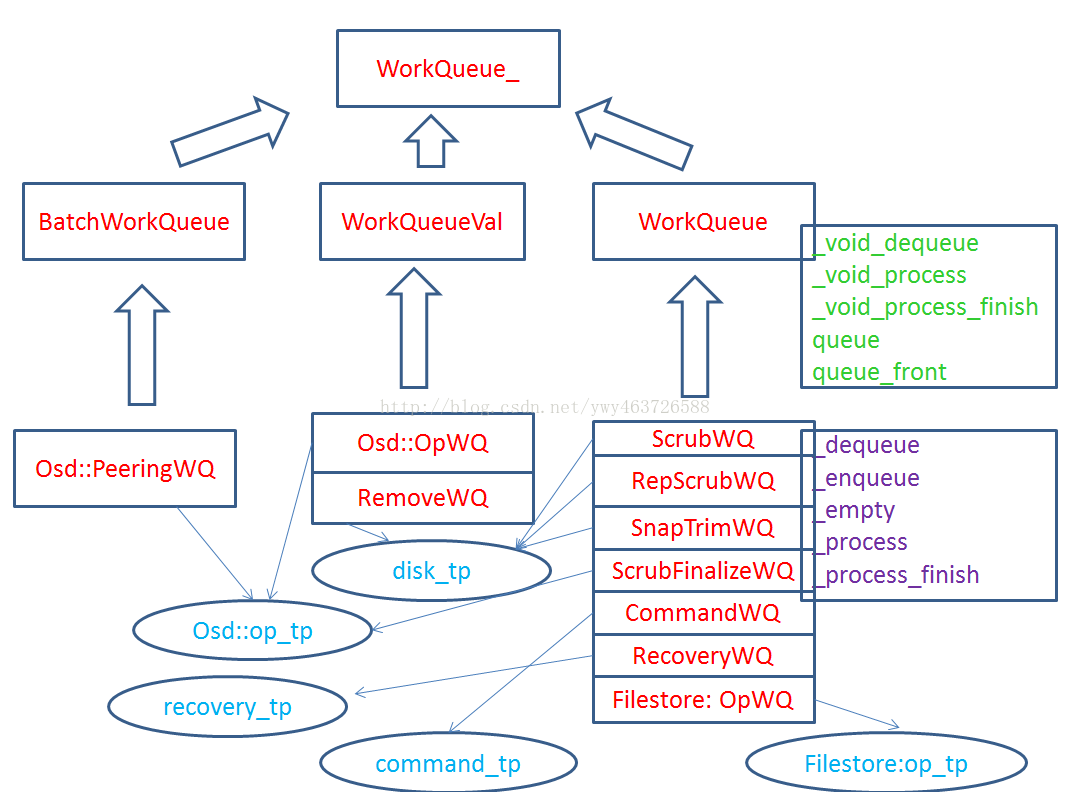
1.声明线程池成员ThreadPool \*\_tp

2.声明队列类型ThreadPool::WorkQueue\_\*\_wq

3.重写WorkQueue中对应函数\_void\_process,\_void\_process\_finish

4.调用\*\_tp.add\_work\_queue(\*\_wq)将队列传入

下图是WorkQueue\_拥有三大继承：



从上图中我们可以看到ceph中目前本人知道的线程池有五种，OSD中四个，FileStore中一个

op\_tp      处理client来的请求

disk\_tp    处理scrub操作

recovery\_tp处理recovery\_tp操作

command\_tp 处理命令行来的操作

FileStore::op\_tp   处理底层数据操作

可以将上一篇介绍写流程底层操作那篇文章中的线程进行分类，可以发现拥有:

一个普通线程:FileJournal::write\_thread

两个finisher线程:Journal::Finisher.finisher\_thread,

FileStore::ondisk\_finisher.finisher\_thread,

FileStore::op\_finisher.finisher\_thread

两个线程池线程: OSD::op\_tp,

FileStore::op\_tp

