Message queue

Benefits

- **Enabling asynchronous processing:**

    - Defer processing of time-consuming tasks without blocking our clients. Anything that is slow or unpredictable is a candidate for asynchronous processing. Example include
        - Interact with remote servers
        - Low-value processing in the critical path
        - Resource intensive work
        - Independent processing of high- and low- priority jobs
    - Message queues enable your application to operate in an asynchronous way, but it only adds value if your application is not built in an asynchronous way to begin with. If you developed in an environment like Node.js, which is built with asynchronous processing at its core, you will not benefit from a message broker that much. What is good about message brokers is that they allow you to easily introduce asynchronous processing to other platforms, like those that are synchronous by nature (C, Java, Ruby)
- **Easier scalability**:
    - Producers and consumers can be scaled separately. We can add more producers at any time without overloading the system. Messages that cannot be consumed fast enough will just begin to line up in the message queue. We can also scale consumers separately, as now they can be hosted on separate machines and the number of consumers can grow independently of producers.
- **Decoupling**:
    - All that publishers need to know is the format of the message and where to publish it. Consumers can become oblivious as to who publishes messages and why. Consumers can focus solely on processing messages from the queue. Such a high level decoupling enables consumers and producers to be developed indepdently. They can even be developed by different teams using different technologies.
- **Evening out traffic spikes**:
    - You should be able to keep accepting requests at high rates even at times of increased traffic. Even if your publishing generates messages much faster than consumers can keep up with, you can keep enqueueing messages, and publishers do not have to be affected by a temporary capacity problem on the consumer side.

- **Isolating failures and self-healing**:
  - The fact that consumers' availability does not affect producers allows us to stop message processing at any time. This means that we can perform maintainance and deployments on back-end servers at any time. We can simply restart, remove, or add servers without affecting producer's availability, which simplifies deployments and server management. Instead of breaking the entire application whenever a back-end server goes offline, all that we experience is reduced throughput, but there is no reduction of availability. Reduced throughput of asynchronous tasks is usually invisible to the user, so there is no consumer impact.

Components

- Message producer
  - Locate the message queue and send a valid message to it
- Message broker - where messages are sent and buffered for consumers.
  - Be available at all times for producers and to accept their messages.
  - Buffering messages and allowing consumers to consume related messages.
- Message consumer
  - Receive and process message from the message queue.
  - The two most common ways of implement consumers are a "cron-like" and a "daemon-like" approach.
    - Connects periodically to the queue and checks the status of the queue. If there are messages, it consumes them and stops when the queue is empty or after consuming a certain amount of messages. This model is common in scripting languages where you do not have a persistenly running application container, such as PHP, Ruby, or Perl. Cron-like is also referred to as a pull model because the consumers pulls messages from the queue. It can also be used if messages are added to the queue rarely or if network connectivity is unreliable. For example, a mobile application may try to pull the queue from time to time, assuming that connection may be lost at any point in time.
    - A daemon-like consumer runs constantly in an infinite loop, and it usually has a permanent connection to the message broker.

Instead of checking the status of the queue periodically, it simply blocks on the socket read operation. This means that the consumer is waiting idly until messages are pushed by the message broker in the connection. This model is more common in languages with persistent application containers, such as Java, C#, and Node.js. This is also referred to as a push model because messages are pushed by the message broker onto the consumer as fast as the consumer can keep processing them.

Routing methods

- Direct worker queue method
  - Consumers and producers only have to know the name of the queue.
  - Well suited for the distribution of time-consuming tasks such as sending out e-mails, processing videos, resizing images, or uploading content to third-party web services.
- Publish/Subscribe method
  - Producers publish message to a topic, not a queue. Messages arriving to a topic are then cloned for each consumer that has a declared subscription to that topic.
- Custom routing rules
  - A consumer can decide in a more flexible way what messages should be routed to its queue.
  - Logging and alerting are good examples of custom routing based on pattern matching.

Protocols

- AMQP: A standardized protocol accepted by OASIS. Aims at enterprise integration and interoperability.
- STOMP: A minimalist protocol.
  - Simplicity is one of its main advantages. It supports fewer than a dozen operations, so implementation and debugging of libraries are much easier. It also means that the protocol layer does not add much performance overhead.
  - But interoperability can be limited because there is no standard way of

doing certain things. A good example of impaired is message prefetch count. Prefetch is a great way of increasing throughput because messages are received in batches instead of one message at a time. Although both RabbitMQ and ActiveMQ support this feature, they both implement it using different custom STOMP headers.

- JMS
    - A good feature set and is popular
    - Your ability to integrate with non-JVM-based languages will be very limited.

Metrics to decide which message broker to use

- Number of messages published per second
- Average message size
- Number of messages consumed per second (this can be much higher than publishing rate, as multiple consumers may be subscribed to receive copies of the same message)
- Number of concurrent publishers
- Number of concurrent consumers
- If message persistence is needed (no message loss during message broker crash)
- If message acknowledgement is need (no message loss during consumer crash)

Challenges

- No message ordering: Messages are processed in parallel and there is no synchronization between consumers. Each consumer works on a single message at a time and has no knowledge of other consumers running in parallel to it. Since your consumers are running in parallel and any of them can become slow or even crash at any point in time, it is difficult to prevent messages from being occasionally delivered out of order.
    - Solutions:
        - Limit the number of consumers to a single thread per queue
        - Build the system to assume that messages can arrive in random order

        - Use a messaging broker that supports partial message ordering guarantee.
    - It is best to depend on the message broker to deliver messages in the right order by using partial message guarantee (ActiveMQ) or topic partitioning (Kafka). If your broker does not support such functionality, you will need to ensure that your application can handle messages being processed in an unpredictable order.
        - Partial message ordering is a clever mechanism provided by ActiveMQ called message groups. Messages can be published with a special label called a message group ID. The group ID is defined by the application developer. Then all messages belonging to the same group are guaranteed to be consumed in the same order they were produced. Whenever a message with a new group ID gets published, the message broker maps the new group Id to one of the existing consumers. From then on, all the messages belonging to the same group are delivered to the same consumer. This may cause other consumers to wait idly without messages as the message broker routes messages based on the mapping rather than random distribution.
    - Message ordering is a serious issue to consider when architecting a message-based application, and RabbitMQ, ActiveMQ and Amazon SQS messaging platform cannot guarantee global message ordering with parallel workers. In fact, Amazon SQS is known for unpredictable ordering messages because their infrastructure is heavily distributed and ordering of messages is not supported.
- Message requeueing
    - By allowing messages to be delivered to your consumers more than once, you make your system more robust and reduce constraints put on the message queue and its workers. For this approach to work, you need to make all of your consumers idempotent.
        - But it is not an easy thing to do. Sending emails is, by nature, not an idempotent operation. Adding an extra layer of tracking and persistence could help, but it would add a lot of complexity and may not be able to handle all of the faiulres.
        - Idempotent consumers may be more sensitive to messages being processed out of order. If we have two messages, one to set the product's price to $55 and another one to set the price of the

    same product to $60, we could end up with different results based on their processing order.

* Race conditions become more likely
* Risk of increased complexity
    * When integrating applications using a message broker, you must be very diligent in documenting dependencies and the overarching message flow. Without good documentation of the message routes and visibility of how the message flow through the system, you may increase the complexity and make it much harder for developers to understand how the system works.

RocketMQ

Definition

* A broker contains a master node and a slave node
    * Broker 1 has topic 1 to 5
    * Broker 2 has topic 6 to 10
* NameNode cluster contains the mapping from Topic=>Broker
* Scenario
    1. Consumer group tells name node cluster which topic it subscribe to
    2. Broker pulls from name node cluster about the heartbeat message (whether I am alive / topic mapping on the broker)
    3. Producer group pushes events to the broker
    4. Broker push events to consumer group

Time series data

* For time series data, RocketMQ must be configured in a standalone mode. There is no HA solution available.
    * Broker 1 and 2 all have the same topic.
    * Consumer is talking to broker 1. Broker 1 has Message 1-10. Broker 2 has message 1-9.
    * When broker 1 dies, if switched to broker 2 then message 10 will be

    lost.
  - It works in non-time series scenarios but not in time-series scenarios.
- RocketMQ high availability ??? To be read:
  1. RocketMQ architecture https://rocketmq.apache.org/docs/rmq-arc/
  2. RocketMQ deployment https://rocketmq.apache.org/docs/rmq-deployment/
  3. RocketMQ high availability http://www.iocoder.cn/RocketMQ/high-availability/

Storage model

- Each consumer consumes an index list
- IndexList
  - Each index contains
    - OffSet
    - Size
    - TagsCode: checksum
- MessageBodyList

Delay message queue

Use cases

- In payment system, if a user has not paid within 30 minutes after ordering. Then this order should be expired and the inventory needs to be reset.
- A user scheduled a smart device to perform a specific task at a certain time. When the time comes, the instruction will be pushed to the user's device from the server.
- Control packet lifetime in networks such as Netty.

Data structures

PriorityQueue

DelayQueue implementation in JDK

- Internal structure: DelayQueue is a specialized PriorityQueue that orders elements based on their delay time.
- Characteristics: When the consumer wants to take an element from the queue, they can take it only when the delay for that particular element has expired.
- Pros:
  - Not introduce other dependencies
- Cons:
  - It is only a data structure implementation and all queue elements will be stored within JVM memory. It would require large amounts of efforts to build a scalable delay queue implementation on top of it.

Delayed interface

- Algorithm: When the consumer tries to take an element from the queue, the DelayQueue will execute getDelay() to find out if that element is allowed to be returned from the queue. If the getDelay() method will return zero or a negative number, it means that it could be retrieved from the queue.
- Data structure:

```
public class DelayQueue<E extends Delayed>
                                        extends AbstractQueue<E>
                                        implements
BlockingQueue<E>
```

```
// Each element we want to put into the DelayQueue needs to
implement the Delayed interface
public class DelayObject implements Delayed {
    private String data;
    private long startTime;

    public DelayObject(String data, long delayInMilliseconds) {
        this.data = data;
        this.startTime = System.currentTimeMillis() +
delayInMilliseconds;
    }

    // It will return the remaining delay associated with the
item in the top of the PriorityQueue in the given time unit.
        @Override
        public long getDelay(TimeUnit unit) {
            long diff = startTime - System.currentTimeMillis();
            return unit.convert(diff, TimeUnit.MILLISECONDS);
        }

        // The elements in the DelayQueue will be sorted
according to the expiration time. The item that will expire
first is kept at the head of the queue and the element with the
highest expiration time is kept at the tail of the queue:
        @Override
        public int compareTo(Delayed o) {
            return Ints.saturatedCast(
              this.startTime - ((DelayObject) o).startTime);
        }
}
```

Test with Producer/Consumer pattern

```
// DelayedQueue is a blocking queue. When delayedQueue.take()
method is called, it will only return when there is an item to
be returned.
public class DelayQueueProducer implements Runnable
{
    private BlockingQueue<DelayObject> queue;
    private Integer numberOfElementsToProduce;
    private Integer delayOfEachProducedMessageMilliseconds;

    // standard constructor

    @Override
    public void run()
    {
        for (int i = 0; i < numberOfElementsToProduce; i++)
        {
            DelayObject object
              = new DelayObject(
                UUID.randomUUID().toString(),
delayOfEachProducedMessageMilliseconds);
            System.out.println("Put object: " + object);
            try
            {
                queue.put(object);
                Thread.sleep(500);
            }
            catch (InterruptedException ie)
            {
                ie.printStackTrace();
            }
        }
    }
}

public class DelayQueueConsumer implements Runnable
{
    private BlockingQueue<DelayObject> queue;
    private Integer numberOfElementsToTake;
    public AtomicInteger numberOfConsumedElements = new
AtomicInteger();

    // standard constructors

    @Override
    public void run() {
        for (int i = 0; i < numberOfElementsToTake; i++)
        {
            try
            {
                DelayObject object = queue.take();
                numberOfConsumedElements.incrementAndGet();
                System.out.println("Consumer take: " + object);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Reference

- https://www.baeldung.com/java-delay-queue

Timing wheel

Simple wheel

- Keep a large timing wheel
- A curser in the timing wheel moves one location every time unit (just like a seconds hand in the clock)
- If the timer interval is within a rotation from the current curser position then put the timer in the corresponding location
- Requires exponential amount of memory

Hashed wheel (sorted)

- Sorted Lists in each bucket
- The list in each bucket can be insertion sorted
- Hence START_TIMER takes O(n) time in the worst case
- If n < WheelSize then average O(1)

Hashed wheel (unsorted)

- Unsorted list in each bucket
- List can be kept unsorted to avoid worst case O(n) latency for START_TIMER
- However worst case PER_TICK_BOOKKEEPING = O(n)
- Again, if n < WheelSize then average O(1)

Hierarchical wheels

- START_TIMER = O(m) where m is the number of wheels. The bucket value on each wheel needs to be calculated
- STOP_TIMER = O(1)
- PER_TICK_BOOKKEEPING = O(1) on avg.

Reference

- A hashed timer implementation https://github.com/ifesdjeen/hashed-wheel-timer
- http://www.cloudwall.io/hashed-wheel-timers
- Implementation in Netty: https://www.jianshu.com/p/f009666ef55c

Implemenations

Timer + Database

- Initial solution: Creates a table within a database, uses a timer thread to scan the table periodically.
- Cons:
  - If the volume of data is large and there is a high frequency of insertion rate, then it won't be efficient to lookup and update records.
  - There is a difference between when task is scheduled to be executed and when the task should be executed.
- How to optimize:
  - Shard the table according to task id to boost the lookup efficiency.

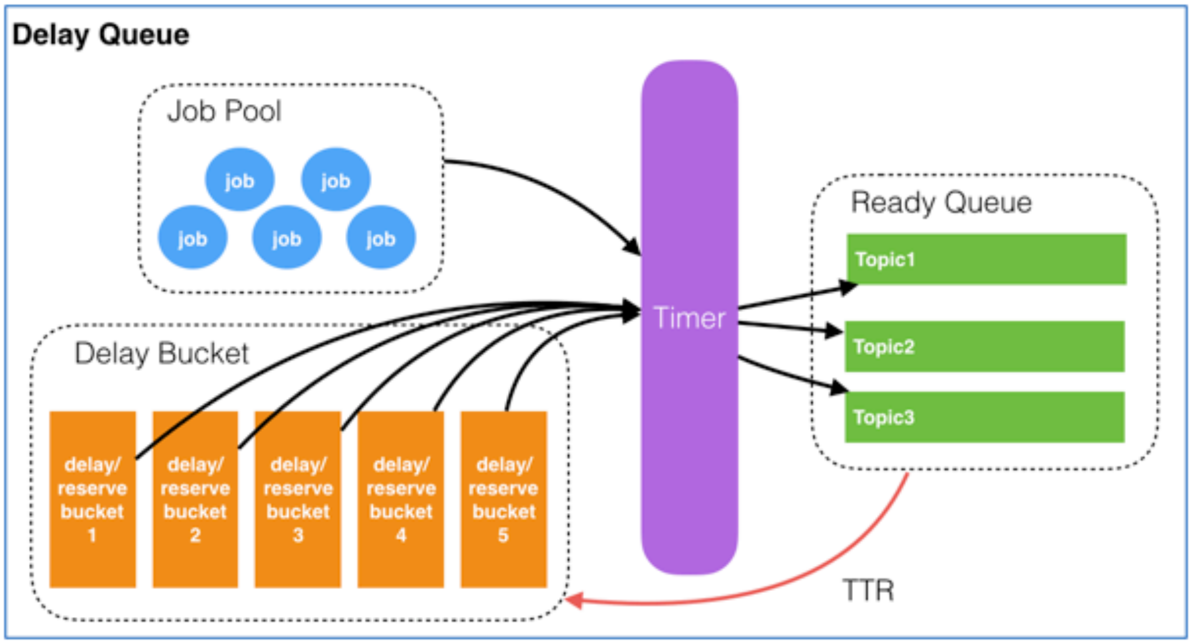
```
INT taskId
TIME expired
INT maxRetryAllowed
INT job status (0: newly created; 1: started; 2: failed; 3:
succeeded)
```

Redis + MySQL

Algorithm

```
redis> ZADD delayqueue <future_timestamp> "messsage"
redis> MULTI
redis> ZRANGEBYSCORE delayqueue 0 <current_timestamp>
redis> ZREMRANGEBYSCORE delayqueue 0 <current_timestamp>
redis> EXEC
```

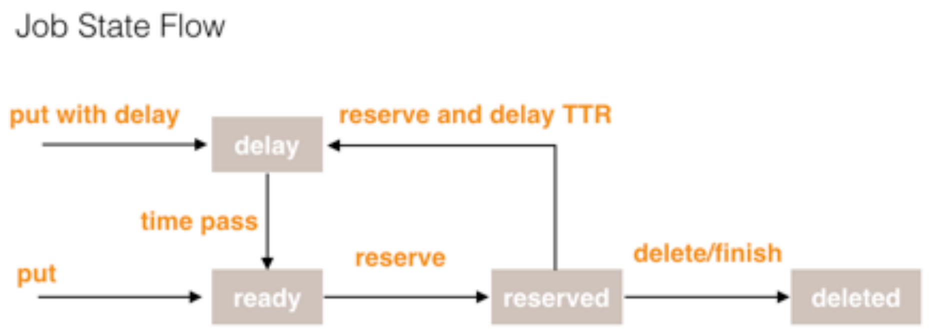
Components



- JobPool: Store all metadata about jobs
  - Stores as key value pairs. Key is job id and value is job struct.
  - Job struct contains the following:
    a. topic: job category. Needed because each category will has its own callback function.
    b. id: job unique identifier
    c. delayTime: time to delay before executing the task
    d. ttr: timeout duration for this job to be executed
    e. body: job content
    f. callback: http url for calling a specific function
- Timer: Scan delay bucket and put expired jobs into ready queue
- Delay queue: A list of ordered queues which store all delayed/reserved jobs (only stores job Id)
- Ready queue: A list of ordered queues which store jobs in Ready state.
  - Topic: The same category of job collections
- Response queue: Stores the responses
- Database: Stores the message content
- Dispatcher: It will poll the delay queue and move items to the corresponding topic within ready queues if the tasks are ready.
- Worker: Workers use BLPOP on the ready queue and process the message. Once done, the response could be put in a response queue and send to consumer.

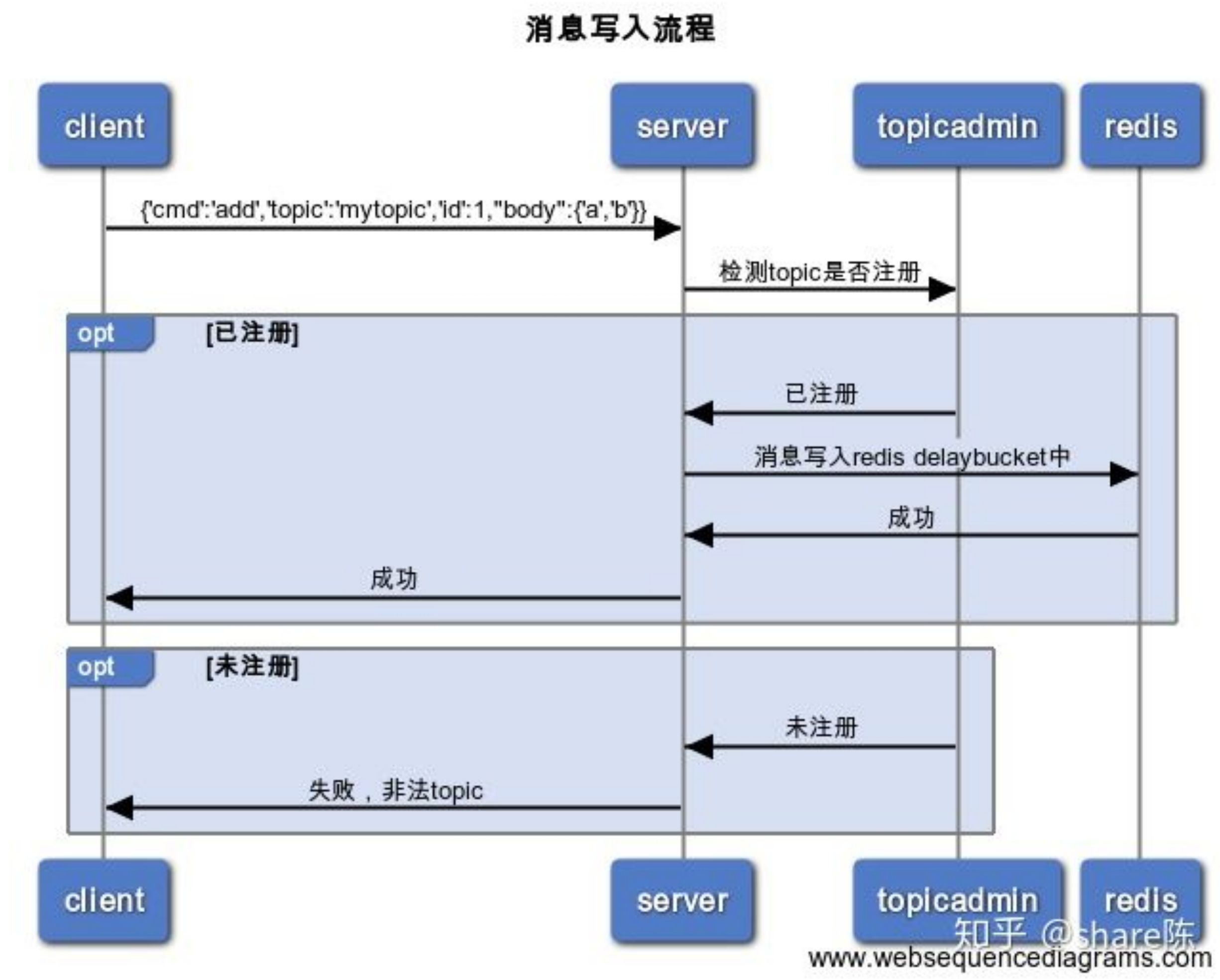
Flow chart (In Chinese)

Job state flow



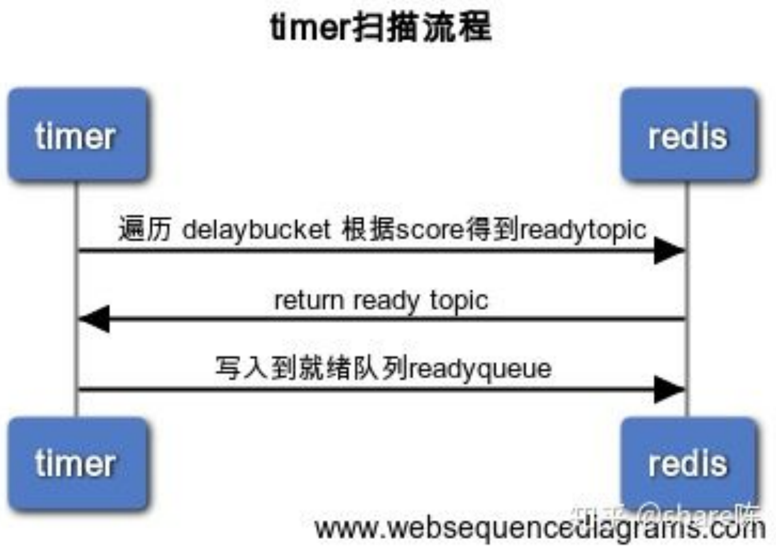
- Ready: The job is ready to be consumed.
- Delay: The job needs to wait for the proper clock cycle.
- Reserved: The job has been read by the consumer, but has not got an acknowledgement (delete/finish)
- Deleted: Consumer has acknowledged and finished.

Produce delay task



- What is topic admin ???
-

Execute delay task



Timer mechanism (Signaling)

- Busy waiting
  - Def: Setting the signal values in some shared object variable. Thread A may set the boolean member variable hasDataToProcess to true from inside a synchronized block, and thread B may read the hasDataToProcess member variable, also inside a synchronized block.
  - Example: Thread B is constantly checking signal from thread A which causes hasDataToProcess() to return true on a loop. This is called busy waiting

```
// class definition
public class MySignal
{
  protected boolean hasDataToProcess = false;

  public synchronized boolean hasDataToProcess()
  {
    return this.hasDataToProcess;
  }

  public synchronized void setHasDataToProcess(boolean hasData)
  {
    this.hasDataToProcess = hasData;
  }
}

...

// main program
protected MySignal sharedSignal = ...

// Thread B is busy waiting for thread a to set

while(!sharedSignal.hasDataToProcess())
{
  //do nothing... busy waiting
}
```

- Wait notify
  - Pros:
    - Reduce the CPU load caused by waiting thread in busy waiting

mode.

- Cons:
    - Missed signals: if you call notify() before wait() it is lost.
    - it can be sometimes unclear if notify() and wait() are called on the same object.
    - There is nothing in wait/notify which requires a state change, yet this is required in most cases.
    - Spurious wakeups: wait() can return spuriously

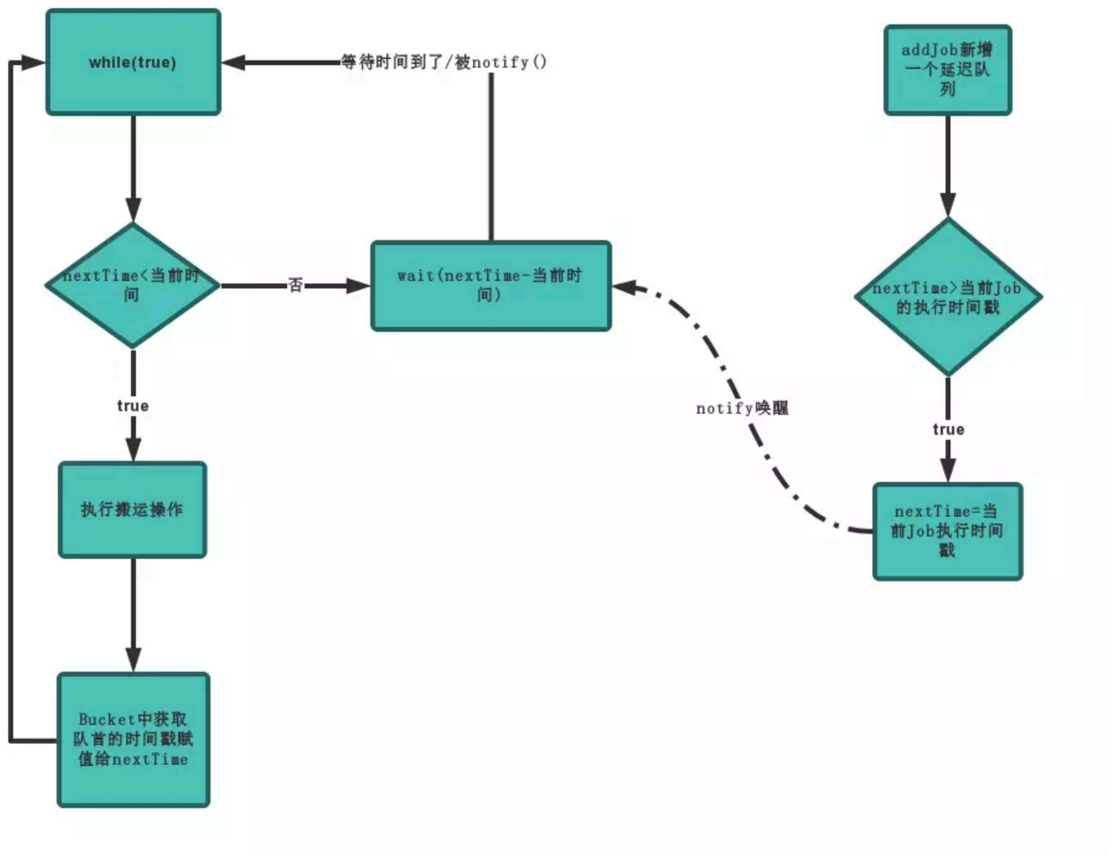
```
// Clients: Insert delayed tasks to delayQueues (Redis sorted
set)
InsertDelayTasks(String msg)
{
    // score = current time + delay time
        redis.zdd(delayTaskSortedSets,score,msg)

        // the number of elements in delayTaskSortedSets
        len = zcount(delayTaskSortedSets, 0, -1)

        // notify polling thread if there exists delayed tasks
to be executed
        synchronized(delayTaskSortedSets)
        {
                if(len > 0)
                {
                    delayTaskSortedSets.notify()
                }
        }
}

// DelayQueue server polling thread: Scan delayQueues and put
expired tasks to ready queue
GetDelayMsg()
{
        while(True)
        {
                // Wait until the number of elements inside
delayTasksSortedTask is bigger than 0
            synchronized(delayTaskSortedSets)
            {
                        while (0 ==
zcount(delayTaskSortedSets,0, -1))
```

```
                        {
                                delayTaskSortedSets.wait()
                        }
                }

                // Peek the top element from delayTasksSortedSet
                msg = redis.zcard(delayTaskSortedSets,0,1)
                waittime = score - curtime

                if(waittime > 0)
                {
                        // Still need to wait
                        synchronized(delayTaskSortedSets)
                        {

delayTaskSortedSets.wait(waittime)
                        }
                }
                else
                {
                        // Add to an element to ReadyQueue
                    readyQueue.put(delayTaskSortedSets, msg)
                        redis.zrem(msg);
                }
        }
}

// ReadyQueue server processing thread: Process ReadyQueue
elements
ProcessReady()
{
    while(True)
    {
                msg = blockingReadyQueue.take()
                MQ.insert(msg)
        }

        mq.inset(msg)
}
```

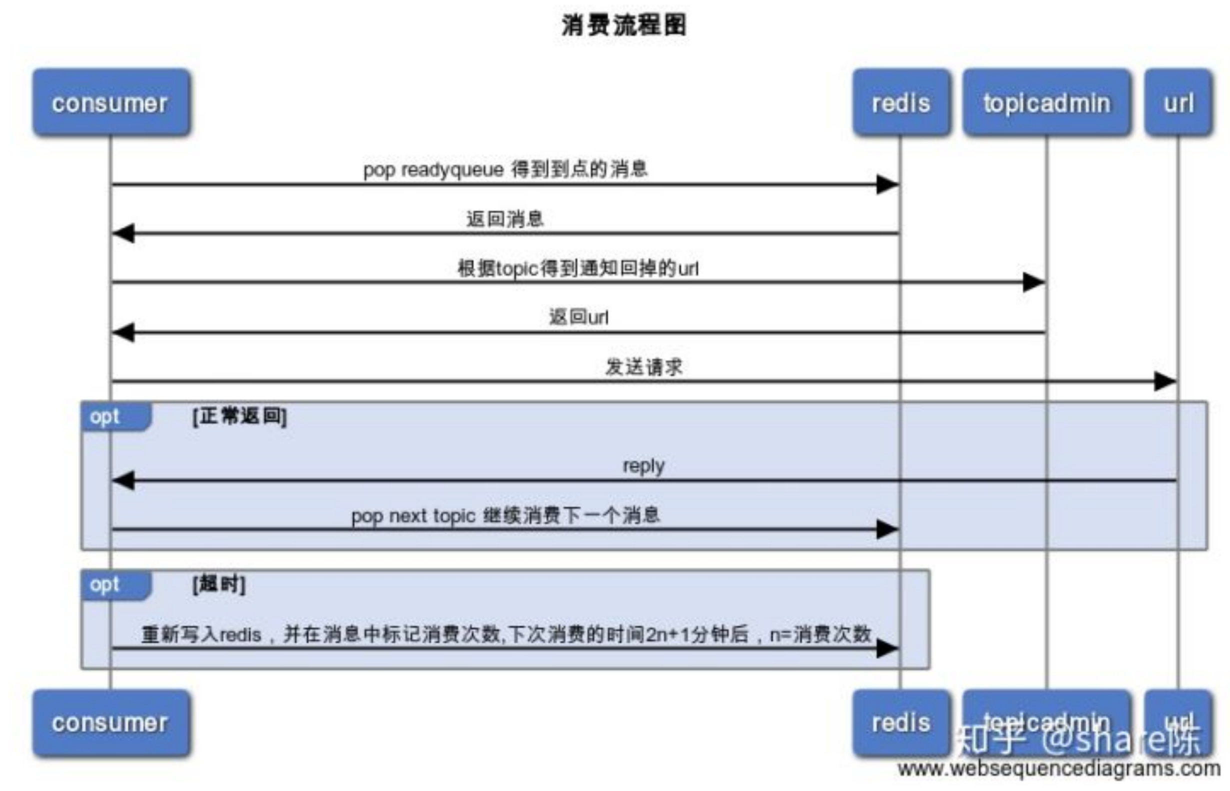
- Wait notify + Regular schedule
    - Motivation: When there are multiple consumers for delay queue, each one of them will possess a different timestamp. Suppose consumer A will move the next delay task within 1 minute and all other consumers

will only start moving after 1 hour. If consumer A dies and does not restart, then it will at least 1 hour for the task to be moved to ready queue. A regular scanning of delay queue will compensate this defficiency.

- When will nextTime be updated:
  - Scenario for starting: When delayQueue polling thread gets started, nextTime = 0 ; Since it must be smaller than the current timestamp, a peeking operation will be performed on top of delayQueue.
    - If there is an item in the delayQueue, nextTime = delayTime from the message;
    - Otherwise, nextTime = Long.MaxValue
  - Scenario for execution: While loop will always be executed on a regular basis
    - If nextTime is bigger than current time, then wait(nextTime - currentTime)
    - Otherwise, the top of the delay queue will be polled out to the ready queue.
  - Scenario for new job being added: Compare delayTime of new job with nextTime
    - If nextTime is bigger than delayTime, nextTime = delayTime; notify all delayQueue polling threads.
    - Otherwise, wait(nextTime - currentTime)



Consume delay task


消费流程图
consumer
redis
topicadmin
url
pop readyqueue 得到到点的消息
返回消息
根据topic得到通知回掉的url
返回url
发送请求
opt
[正常返回]
reply
pop next topic 继续消费下一个消息
opt
[超时]
重新写入redis，并在消息中标记消费次数,下次消费的时间2n+1分钟后，n=消费次数
知乎 @share陈
www.websequencediagrams.com

- Workers use BLPOP on the topics

Consume multiple jobs at once ???

TCP long polling ???

Retention ???

- Assumption: QPS 1000, maximum retention period 7 days,

How to scale?

Fault tolerant

- For a message in ready queue, if server has not received acknowledgement within certain period (e.g. 5min), the message will be put inside Ready queue again.
- There needs to be a leader among server nodes. Otherwise message might be put into ready queue repeatedly.
- How to guarantee that there is no message left during BLPOP and server restart?
  - Kill the Redis blpop client when shutting down the server.
  - https://hacpai.com/article/1565796946371

Redisson ???

ScheduledExecutorService ???

Beanstalk

- Cons
    - Not convenient when deleting a msg.
    - Developed based on C language, not Java and PHP.

References

- https://github.blog/2009-11-03-introducing-resque/
- http://tutorials.jenkov.com/java-concurrency/thread-signaling.html
- https://hacpai.com/article/1565796946371
- https://stackoverflow.com/questions/10868552/scalable-delayed-task-execution-with-redis
- https://juejin.im/post/5b5e52ecf265da0f716c3203
- https://tech.youzan.com/queuing_delay/
- http://www.throwable.club/2019/09/01/redis-delay-task-second/