NAME

Thread::Queue - Thread-safe queues

VERSION

This document describes Thread::Queue version 3.12

SYNOPSIS

```
use strict;
use warnings;

use threads;
use Thread::Queue;

my $q = Thread::Queue->new();    # A new empty queue

# Worker thread
my $thr = threads->create(
    sub {
        # Thread will loop until no more work
        while (defined(my $item = $q->dequeue())) {
            # Do work on $item
            ...
        }
    }
);

# Send work to the thread
$q->enqueue($item1, ...);
# Signal that there is no more work to be sent
$q->end();
# Join up with the thread when it finishes
$thr->join();


...

# Count of items in the queue
my $left = $q->pending();

# Non-blocking dequeue
if (defined(my $item = $q->dequeue_nb())) {
    # Work on $item
}

# Blocking dequeue with 5-second timeout
if (defined(my $item = $q->dequeue_timed(5))) {
    # Work on $item
}

# Set a size for a queue
$q->limit = 5;

# Get the second item in the queue without dequeuing anything
```

```
my $item = $q->peek(1);

# Insert two items into the queue just behind the head
$q->insert(1, $item1, $item2);

# Extract the last two items on the queue
my ($item1, $item2) = $q->extract(-2, 2);
```

DESCRIPTION

This module provides thread-safe FIFO queues that can be accessed safely by any number of threads.

Any data types supported by *threads::shared* can be passed via queues:

Ordinary scalars

Array refs

Hash refs

Scalar refs

Objects based on the above

Ordinary scalars are added to queues as they are.

If not already thread-shared, the other complex data types will be cloned (recursively, if needed, and including any `bless`ings and read-only settings) into thread-shared structures before being placed onto a queue.

For example, the following would cause *Thread::Queue* to create a empty, shared array reference via `&shared([])`, copy the elements 'foo', 'bar' and 'baz' from `@ary` into it, and then place that shared reference onto the queue:

```
my @ary = qw/foo bar baz/;
$q->enqueue(\@ary);
```

However, for the following, the items are already shared, so their references are added directly to the queue, and no cloning takes place:

```
my @ary :shared = qw/foo bar baz/;
$q->enqueue(\@ary);

my $obj = &shared({});
$$obj{'foo'} = 'bar';
$$obj{'qux'} = 99;
bless($obj, 'My::Class');
$q->enqueue($obj);
```

See *LIMITATIONS* for caveats related to passing objects via queues.

QUEUE CREATION

->new()

Creates a new empty queue.

->new(LIST)

Creates a new queue pre-populated with the provided list of items.

BASIC METHODS

The following methods deal with queues on a FIFO basis.

->enqueue(LIST)

Adds a list of items onto the end of the queue.

->dequeue()

->dequeue(COUNT)

Removes the requested number of items (default is 1) from the head of the queue, and returns them. If the queue contains fewer than the requested number of items, then the thread will be blocked until the requisite number of items are available (i.e., until other threads `enqueue` more items).

->dequeue_nb()

->dequeue_nb(COUNT)

Removes the requested number of items (default is 1) from the head of the queue, and returns them. If the queue contains fewer than the requested number of items, then it immediately (i.e., non-blocking) returns whatever items there are on the queue. If the queue is empty, then `undef` is returned.

->dequeue_timed(TIMEOUT)

->dequeue_timed(TIMEOUT, COUNT)

Removes the requested number of items (default is 1) from the head of the queue, and returns them. If the queue contains fewer than the requested number of items, then the thread will be blocked until the requisite number of items are available, or until the timeout is reached. If the timeout is reached, it returns whatever items there are on the queue, or `undef` if the queue is empty.

The timeout may be a number of seconds relative to the current time (e.g., 5 seconds from when the call is made), or may be an absolute timeout in *epoch* seconds the same as would be used with *cond_timedwait()*. Fractional seconds (e.g., 2.5 seconds) are also supported (to the extent of the underlying implementation).

If `TIMEOUT` is missing, `undef`, or less than or equal to 0, then this call behaves the same as `dequeue_nb`.

->pending()

Returns the number of items still in the queue. Returns `undef` if the queue has been ended (see below), and there are no more items in the queue.

->limit

Sets the size of the queue. If set, calls to `enqueue()` will block until the number of pending items in the queue drops below the `limit`. The `limit` does not prevent enqueuing items beyond that count:

```
my $q = Thread::Queue->new(1, 2);
$q->limit = 4;
$q->enqueue(3, 4, 5);   # Does not block
$q->enqueue(6);         # Blocks until at least 2 items are
                        # dequeued
my $size = $q->limit;   # Returns the current limit (may return
                        # 'undef')
$q->limit = 0;          # Queue size is now unlimited
```

Calling any of the dequeue methods with `COUNT` greater than a queue's `limit` will generate an error.

->end()

Declares that no more items will be added to the queue.

All threads blocking on `dequeue()` calls will be unblocked with any remaining items in the queue and/or `undef` being returned. Any subsequent calls to `dequeue()` will behave like `dequeue_nb()`.

Once ended, no more items may be placed in the queue.

ADVANCED METHODS

The following methods can be used to manipulate items anywhere in a queue.

To prevent the contents of a queue from being modified by another thread while it is being examined and/or changed, *lock* the queue inside a local block:

```
{
    lock($q);   # Keep other threads from changing the queue's contents
    my $item = $q->peek();
    if ($item ...) {
        ...
    }
}
# Queue is now unlocked
```

->peek()

->peek(INDEX)

Returns an item from the queue without dequeuing anything. Defaults to the the head of queue (at index position 0) if no index is specified. Negative index values are supported as with *arrays* (i.e., -1 is the end of the queue, -2 is next to last, and so on).

If no items exists at the specified index (i.e., the queue is empty, or the index is beyond the number of items on the queue), then `undef` is returned.

Remember, the returned item is not removed from the queue, so manipulating a `peek`ed at reference affects the item on the queue.

->insert(INDEX, LIST)

Adds the list of items to the queue at the specified index position (0 is the head of the list). Any existing items at and beyond that position are pushed back past the newly added items:

```
$q->enqueue(1, 2, 3, 4);
$q->insert(1, qw/foo bar/);
# Queue now contains:  1, foo, bar, 2, 3, 4
```

Specifying an index position greater than the number of items in the queue just adds the list to the end.

Negative index positions are supported:

```
$q->enqueue(1, 2, 3, 4);
$q->insert(-2, qw/foo bar/);
# Queue now contains:  1, 2, foo, bar, 3, 4
```

Specifying a negative index position greater than the number of items in the queue adds the list to the head of the queue.

->extract()

->extract(INDEX)

->extract(INDEX, COUNT)

Removes and returns the specified number of items (defaults to 1) from the specified index position in the queue (0 is the head of the queue). When called with no arguments, `extract` operates the same as `dequeue_nb`.

This method is non-blocking, and will return only as many items as are available to fulfill the request:

```
$q->enqueue(1, 2, 3, 4);
my $item  = $q->extract(2)     # Returns 3
                               # Queue now contains:  1, 2, 4
my @items = $q->extract(1, 3)  # Returns (2, 4)
                               # Queue now contains:  1
```

Specifying an index position greater than the number of items in the queue results in `undef` or an empty list being returned.

```
$q->enqueue('foo');
my $nada = $q->extract(3)      # Returns undef
my @nada = $q->extract(1, 3)   # Returns ()
```

Negative index positions are supported. Specifying a negative index position greater than the number of items in the queue may return items from the head of the queue (similar to `dequeue_nb`) if the count overlaps the head of the queue from the specified position (i.e. if queue size + index + count is greater than zero):

```
$q->enqueue(qw/foo bar baz/);
my @nada = $q->extract(-6, 2);  # Returns ()      - (3+(-6)+2) <= 0
my @some = $q->extract(-6, 4);  # Returns (foo)   - (3+(-6)+4) > 0
                                # Queue now contains:  bar, baz
my @rest = $q->extract(-3, 4);  # Returns (bar, baz) -
                                #                  (2+(-3)+4) > 0
```

NOTES

Queues created by *Thread::Queue* can be used in both threaded and non-threaded applications.

LIMITATIONS

Passing objects on queues may not work if the objects' classes do not support sharing. See *"BUGS AND LIMITATIONS" in threads::shared* for more.

Passing array/hash refs that contain objects may not work for Perl prior to 5.10.0.

SEE ALSO

Thread::Queue on MetaCPAN: *https://metacpan.org/release/Thread-Queue*

Code repository for CPAN distribution: *https://github.com/Dual-Life/Thread-Queue*

threads, *threads::shared*

Sample code in the *examples* directory of this distribution on CPAN.

MAINTAINER

Jerry D. Hedden, <jdhedden AT cpan DOT org>

LICENSE