NAME

"Future::Phrasebook" − coding examples for "Future" and "Future::Utils"

This documentation−only module provides a phrasebook−like approach to giving examples on how to use Future and Future::Utils to structure Future−driven asynchronous or concurrent logic. As with any inter−dialect phrasebook it is structured into pairs of examples; each given first in a traditional call/return Perl style, and second in a style using Futures. In each case, the generic function or functions in the example are named in "ALL_CAPITALS()" to make them stand out.

In the examples showing use of Futures, any function that is expected to return a "Future" instance is named with a leading "F_" prefix. Each example is also constructed so as to yield an overall future in a variable called "$f", which represents the entire operation.

SEQUENCING

The simplest example of a sequencing operation is simply running one piece of code, then immediately running a second. In call/return code we can just place one after the other.

```
FIRST();
SECOND();
```

Using a Future it is necessary to await the result of the first `Future` before calling the second.

```
my $f = F_FIRST()
   ->then( sub { F_SECOND(); } );
```

Here, the anonymous closure is invoked once the `Future` returned by `F_FIRST()` succeeds. Because `then` invokes the code block only if the first Future succeeds, it shortcircuits around failures similar to the way that `die()` shortcircuits around thrown exceptions. A `Future` representing the entire combination is returned by the method.

Because the `then` method itself returns a `Future` representing the overall operation, it can itself be further chained.

```
FIRST();
SECOND();
THIRD();
```

```
my $f = F_FIRST()
   ->then( sub { F_SECOND(); } )
   ->then( sub { F_THIRD(); } );
```

See below for examples of ways to handle exceptions.

Passing Results

Often the result of one function can be passed as an argument to another function.

```
OUTER( INNER() );
```

The result of the first `Future` is passed into the code block given to the `then` method.

```
my $f = F_INNER()
   ->then( sub { F_OUTER( @_ ) } );
```

CONDITIONALS

It may be that the result of one function call is used to determine whether or not another operation is taken.

```
if( COND() == $value ) {
   ACTION();
}
```

Because the `then_with_f` code block is given the first future in addition to its results it can decide whether to call the second function to return a new future, or simply return the one it was given.

```
my $f = F_COND()
   ->then_with_f( sub {
      my ( $f_cond, $result ) = @_;
      if( $result == $value ) {
         return F_ACTION();
      }
      else {
         return $f_cond;
      }
   });
```

EXCEPTION HANDLING

In regular call/return style code, if any function throws an exception, the remainder of the block is not executed, the containing `try` or `eval` is aborted, and control is passed to the corresponding `catch` or line after the `eval`.

```
try {
   FIRST();
}
catch {
   my $e = $_;
   ERROR( $e );
};
```

The `else` method on a `Future` can be used here. It behaves similar to `then`, but is only invoked if the initial `Future` fails; not if it succeeds.

```
my $f = F_FIRST()
   ->else( sub { F_ERROR( @_ ); } );
```

Alternatively, the second argument to the `then` method can be applied, which is invoked only on case of failure.

```
my $f = F_FIRST()
   ->then( undef, sub { F_ERROR( @_ ); } );
```

Often it may be the case that the failure-handling code is in fact immediate, and doesn't return a `Future`. In that case, the `else` code block can return an immediate `Future` instance.

```
my $f = F_FIRST()
   ->else( sub {
      ERROR( @_ );
      return Future->done;
   });
```

Sometimes the failure handling code simply needs to be aware of the failure, but rethrow it further up.

```
try {
   FIRST();
}
catch {
   my $e = $_;
   ERROR( $e );
   die $e;
};
```

In this case, while the `else` block could return a new `Future` failed with the same exception, the `else_with_f` block is passed the failed `Future` itself in addition to the failure details so it can just return that.

```
my $f = F_FIRST()
   ->else_with_f( sub {
      my ( $f1, @failure ) = @_;
      ERROR( @failure );
      return $f1;
   });
```

The `followed_by` method is similar again, though it invokes the code block regardless of the success or failure of the initial `Future`. It can be used to create `finally` semantics. By returning the `Future` instance that it was passed, the `followed_by` code ensures it doesn't affect the result of the operation.

```
try {
   FIRST();
}
catch {
   ERROR( $_ );
}
finally {
   CLEANUP();
};
```

```
my $f = F_FIRST()
   ->else( sub {
      ERROR( @_ );
      return Future->done;
   })
   ->followed_by( sub {
      CLEANUP();
      return shift;
   });
```

ITERATION

To repeat a single block of code multiple times, a `while` block is often used.

```
while( COND() ) {
   FUNC();
}
```

The `Future::Utils::repeat` function can be used to repeatedly iterate a given `Future`-returning block of code until its ending condition is satisfied.

```
use Future::Utils qw( repeat );
my $f = repeat {
   F_FUNC();
} while => sub { COND() };
```

Unlike the statement nature of perl's `while` block, this `repeat Future` can yield a value; the value returned by `$f->get` is the result of the final trial of the code block.

Here, the condition function it expected to return its result immediately. If the repeat condition function itself returns a `Future`, it can be combined along with the loop body. The trial `Future` returned by the code block is passed to the `while` condition function.

```
my $f = repeat {
   F_FUNC()
      ->followed_by( sub { F_COND(); } );
} while => sub { shift->get };
```

The condition can be negated by using `until` instead

```
until( HALTING_COND() ) {
   FUNC();
}
```

```
my $f = repeat {
   F_FUNC();
} until => sub { HALTING_COND() };
```

Iterating with Exceptions

Technically, this loop isn't quite the same as the equivalent `while` loop in plain Perl, because the `while` loop will also stop executing if the code within it throws an exception. This can be handled in `repeat` by testing for a failed `Future` in the `until` condition.

```
while(1) {
   TRIAL();
}
```

```
my $f = repeat {
   F_TRIAL();
} until => sub { shift->failure };
```

When a repeat loop is required to retry a failure, the `try_repeat` function should be used. Currently this function behaves equivalently to `repeat`, except that it will not print a warning if it is asked to retry after a failure, whereas this behaviour is now deprecated for the regular `repeat` function so that yields a warning.

```
my $f = try_repeat {
   F_TRIAL();
} while => sub { shift->failure };
```

Another variation is the `try_repeat_until_success` function, which provides a convenient shortcut to calling `try_repeat` with a condition that makes another attempt each time the previous one fails; stopping once it achieves a successful result.

```
while(1) {
   eval { TRIAL(); 1 } and last;
}
```

```
my $f = try_repeat_until_success {
   F_TRIAL();
};
```

Iterating over a List

A variation on the idea of the `while` loop is the `foreach` loop; a loop that executes once for each item in a given list, with a variable set to one value from that list each time.

```
foreach my $thing ( @THINGS ) {
   INSPECT( $thing );
}
```

This can be performed with `Future` using the `foreach` parameter to the `repeat` function. When this is in effect, the block of code is passed each item of the given list as the first parameter.

```
my $f = repeat {
   my $thing = shift;
   F_INSPECT( $thing );
} foreach => \@THINGS;
```

Recursing over a Tree

A regular call/return function can use recursion to walk over a tree-shaped structure, where each item yields a list of child items.

```
sub WALK
{
   my ( $item ) = @_;
   ...
   WALK($_) foreach CHILDREN($item);
}
```

This recursive structure can be turned into a `while()`–based repeat loop by using an array to store the remaining items to walk into, instead of using the perl stack directly:

```
sub WALK
{
   my @more = ( $root );
   while( @more ) {
      my $item = shift @more;
      ...
      unshift @more, CHILDREN($item)
   }
}
```

This arrangement then allows us to use `fmap_void` to walk this structure using Futures, possibly concurrently. A lexical array variable is captured that holds the stack of remaining items, which is captured by the item code so it can `unshift` more into it, while also being used as the actual `fmap` control array.

```
my @more = ( $root );

my $f = fmap_void {
   my $item = shift;
   ...->on_done( sub {
      unshift @more, @CHILDREN;
   })
} foreach => \@more;
```

By choosing to either `unshift` or `push` more items onto this list, the tree can be walked in either depth-first or breadth-first order.

SHORT-CIRCUITING

Sometimes a result is determined that should be returned through several levels of control structure. Regular Perl code has such keywords as `return` to return a value from a function immediately, or `last` for immediately stopping execution of a loop.

```
sub func {
   foreach my $item ( @LIST ) {
      if( COND($item) ) {
         return $item;
      }
   }
   return MAKE_NEW_ITEM();
}
```

The `Future::Utils::call_with_escape` function allows this general form of control flow, by calling a block of code that is expected to return a future, and itself returning a future. Under normal circumstances the result of this future propagates through to the one returned by `call_with_escape`.

However, the code is also passed in a future value, called here the ‘‘escape future’’. If the code captures this future and completes it (either by calling `done` or `fail`), then the overall returned future immediately completes with that result instead, and the future returned by the code block is cancelled.

```
my $f = call_with_escape {
   my $escape_f = shift;
```

```
   ( repeat {
      my $item = shift;
      COND($item)->then( sub {
         my ( $result ) = @_;
         if( $result ) {
            $escape_f->done( $item );
         }
         return Future->done;
      })
   } foreach => \@ITEMS )->then( sub {
      MAKE_NEW_ITEM();
   });
};
```

Here, if `$escape_f` is completed by the condition test, the future chain returned by the code (that is, the `then` chain of the `repeat` block followed by `MAKE_NEW_ITEM()`) will be cancelled, and `$f` itself will receive this result.

CONCURRENCY

This final section of the phrasebook demonstrates a number of abilities that are simple to do with `Future` but can't easily be done with regular call/return style programming, because they all involve an element of concurrency. In these examples the comparison with regular call/return code will be somewhat less accurate because of the inherent ability for the `Future`–using version to behave concurrently.

Waiting on Multiple Functions

The `Future->wait_all` constructor creates a `Future` that waits for all of the component futures to complete. This can be used to form a sequence with concurrency.

```
{ FIRST_A(); FIRST_B() }
SECOND();
```

```
my $f = Future->wait_all( FIRST_A(), FIRST_B() )
   ->then( sub { SECOND() } );
```

Unlike in the call/return case, this can perform the work of `FIRST_A()` and `FIRST_B()` concurrently, only proceeding to `SECOND()` when both are ready.

The result of the `wait_all` `Future` is the list of its component `Future`s. This can be used to obtain the results.

```
SECOND( FIRST_A(), FIRST_B() );
```

```
my $f = Future->wait_all( FIRST_A(), FIRST_B() )
   ->then( sub {
      my ( $f_a, $f_b ) = @_
      SECOND( $f_a->get, $f_b->get );
   } );
```

Because the `get` method will re-raise an exception caused by a failure of either of the `FIRST` functions, the second stage will fail if any of the initial Futures failed.

As this is likely to be the desired behaviour most of the time, this kind of control flow can be written slightly neater using `Future->needs_all` instead.

```
my $f = Future->needs_all( FIRST_A(), FIRST_B() )
   ->then( sub { SECOND( @_ ) } );
```

The `get` method of a `needs_all` convergent Future returns a concatenated list of the results of all its component Futures, as the only way it will succeed is if all the components do.

Waiting on Multiple Calls of One Function

Because the `wait_all` and `needs_all` constructors take an entire list of `Future` instances, they can be conveniently used with `map` to wait on the result of calling a function concurrently once per item in a list.

```
my @RESULT = map { FUNC( $_ ) } @ITEMS;
PROCESS( @RESULT );
```

Again, the `needs_all` version allows more convenient access to the list of results.

```
my $f = Future->needs_all( map { F_FUNC( $_ ) } @ITEMS )
   ->then( sub {
      my @RESULT = @_;
      F_PROCESS( @RESULT )
   } );
```

This form of the code starts every item's future concurrently, then waits for all of them. If the list of `@ITEMS` is potentially large, this may cause a problem due to too many items running at once. Instead, the `Future::Utils::fmap` family of functions can be used to bound the concurrency, keeping at most some given number of items running, starting new ones as existing ones complete.

```
my $f = fmap {
   my $item = shift;
   F_FUNC( $item )
} foreach => \@ITEMS;
```

By itself, this will not actually act concurrently as it will only keep one Future outstanding at a time. The `concurrent` flag lets it keep a larger number "in flight" at any one time:

```
my $f = fmap {
   my $item = shift;
   F_FUNC( $item )
} foreach => \@ITEMS, concurrent => 10;
```

The `fmap` and `fmap_scalar` functions return a Future that will eventually give the collected results of the individual item futures, thus making them similar to perl's `map` operator.

Sometimes, no result is required, and the items are run in a loop simply for some side-effect of the body.

```
foreach my $item ( @ITEMS ) {
   FUNC( $item );
}
```

To avoid having to collect a potentially-large set of results only to throw them away, the `fmap_void` function variant of the `fmap` family yields a Future that completes with no result after all the items are complete.

```
my $f = fmap_void {
   my $item = shift;
   F_FIRST( $item )
} foreach => \@ITEMS, concurrent => 10;
```

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>