NAME

"Future::Utils" – utility functions for working with "Future" objects

SYNOPSIS

```
use Future::Utils qw( call_with_escape );

my $result_f = call_with_escape {
   my $escape_f = shift;
   my $f = ...
      $escape_f->done( "immediate result" );
      ...
};
```

```
use Future::Utils qw( repeat try_repeat try_repeat_until_success );

my $eventual_f = repeat {
   my $trial_f = ...
   return $trial_f;
} while => sub { my $f = shift; return want_more($f) };

my $eventual_f = repeat {
   ...
   return $trial_f;
} until => sub { my $f = shift; return acceptable($f) };

my $eventual_f = repeat {
   my $item = shift;
   ...
   return $trial_f;
} foreach => \@items;

my $eventual_f = try_repeat {
   my $trial_f = ...
   return $trial_f;
} while => sub { ... };

my $eventual_f = try_repeat_until_success {
   ...
   return $trial_f;
};

my $eventual_f = try_repeat_until_success {
   my $item = shift;
   ...
   return $trial_f;
} foreach => \@items;
```

```
use Future::Utils qw( fmap_concat fmap_scalar fmap_void );

my $result_f = fmap_concat {
   my $item = shift;
   ...
   return $item_f;
} foreach => \@items, concurrent => 4;

my $result_f = fmap_scalar {
```

```
      my $item = shift;
      ...
      return $item_f;
   } foreach => \@items, concurrent => 8;

   my $done_f = fmap_void {
      my $item = shift;
      ...
      return $item_f;
   } foreach => \@items, concurrent => 10;
```

Unless otherwise noted, the following functions require at least version *0.08*.

INVOKING A BLOCK OF CODE

call

```
$f = call { CODE }
```

Since version 0.22.

The `call` function invokes a block of code that returns a future, and simply returns the future it returned. The code is wrapped in an `eval {}` block, so that if it throws an exception this is turned into an immediate failed `Future`. If the code does not return a `Future`, then an immediate failed `Future` instead.

(This is equivalent to using `Future->call`, but is duplicated here for completeness).

call_with_escape

```
$f = call_with_escape { CODE }
```

Since version 0.22.

The `call_with_escape` function invokes a block of code that returns a future, and passes in a separate future (called here an "escape future"). Normally this is equivalent to the simple `call` function. However, if the code captures this future and completes it by calling `done` or `fail` on it, the future returned by `call_with_escape` immediately completes with this result, and the future returned by the code itself is cancelled.

This can be used to implement short-circuit return from an iterating loop or complex sequence of code, or immediate fail that bypasses failure handling logic in the code itself, or several other code patterns.

```
$f = $code->( $escape_f )
```

(This can be considered similar to `call-with-escape-continuation` as found in some Scheme implementations).

REPEATING A BLOCK OF CODE

The `repeat` function provides a way to repeatedly call a block of code that returns a Future (called here a "trial future") until some ending condition is satisfied. The `repeat` function itself returns a `Future` to represent running the repeating loop until that end condition (called here the "eventual future"). The first time the code block is called, it is passed no arguments, and each subsequent invocation is passed the previous trial future.

The result of the eventual future is the result of the last trial future.

If the eventual future is cancelled, the latest trial future will be cancelled.

If some specific subclass or instance of `Future` is required as the return value, it can be passed as the `return` argument. Otherwise the return value will be constructed by cloning the first non-immediate trial `Future`.

repeat+while

```
$future = repeat { CODE } while => CODE
```

Repeatedly calls the `CODE` block while the `while` condition returns a true value. Each time the trial future completes, the `while` condition is passed the trial future.

```
$trial_f = $code->( $previous_trial_f )
$again = $while->( $trial_f )
```

If the `$code` block dies entirely and throws an exception, this will be caught and considered as an immediately-failed `Future` with the exception as the future's failure. The exception will not be propagated to the caller.

repeat+until

```
$future = repeat { CODE } until => CODE
```

Repeatedly calls the `CODE` block until the `until` condition returns a true value. Each time the trial future completes, the `until` condition is passed the trial future.

```
$trial_f = $code->( $previous_trial_f )
$accept = $until->( $trial_f )
```

repeat+foreach

```
$future = repeat { CODE } foreach => ARRAY, otherwise => CODE
```

Since version 0.13.

Calls the `CODE` block once for each value obtained from the array, passing in the value as the first argument (before the previous trial future). When there are no more items left in the array, the `otherwise` code is invoked once and passed the last trial future, if there was one, or `undef` if the list was originally empty. The result of the eventual future will be the result of the future returned from `otherwise`.

The referenced array may be modified by this operation.

```
$trial_f = $code->( $item, $previous_trial_f )
$final_f = $otherwise->( $last_trial_f )
```

The `otherwise` code is optional; if not supplied then the result of the eventual future will simply be that of the last trial. If there was no trial, because the `foreach` list was already empty, then an immediate successful future with an empty result is returned.

repeat+foreach+while

```
$future = repeat { CODE } foreach => ARRAY, while => CODE, ...
```

Since version 0.13.

repeat+foreach+until

```
$future = repeat { CODE } foreach => ARRAY, until => CODE, ...
```

Since version 0.13.

Combines the effects of `foreach` with `while` or `until`. Calls the `CODE` block once for each value obtained from the array, until the array is exhausted or the given ending condition is satisfied.

If a `while` or `until` condition is combined with `otherwise`, the `otherwise` code will only be run if the array was entirely exhausted. If the operation is terminated early due to the `while` or `until` condition being satisfied, the eventual result will simply be that of the last trial that was executed.

repeat+generate

```
$future = repeat { CODE } generate => CODE, otherwise => CODE
```

Since version 0.13.

Calls the `CODE` block once for each value obtained from the generator code, passing in the value as the first argument (before the previous trial future). When the generator returns an empty list, the `otherwise` code is invoked and passed the last trial future, if there was one, otherwise `undef` if the generator never returned a value. The result of the eventual future will be the result of the future returned from `otherwise`.

```
$trial_f = $code->( $item, $previous_trial_f )
$final_f = $otherwise->( $last_trial_f )
```

```
( $item ) = $generate->()
```

The generator is called in list context but should return only one item per call. Subsequent values will be ignored. When it has no more items to return it should return an empty list.

For backward compatibility this function will allow a `while` or `until` condition that requests a failure be repeated, but it will print a warning if it has to do that. To apply repeating behaviour that can catch and retry failures, use `try_repeat` instead. This old behaviour is now deprecated and will be removed in the next version.

try_repeat

```
$future = try_repeat { CODE } ...
```

Since version 0.18.

A variant of `repeat` that doesn't warn when the trial fails and the condition code asks for it to be repeated.

In some later version the `repeat` function will be changed so that if a trial future fails, then the eventual future will immediately fail as well, making its semantics a little closer to that of a `while {}` loop in Perl. Code that specifically wishes to catch failures in trial futures and retry the block should use `try_repeat` specifically.

try_repeat_until_success

```
$future = try_repeat_until_success { CODE } ...
```

Since version 0.18.

A shortcut to calling `try_repeat` with an ending condition that simply tests for a successful result from a future. May be combined with `foreach` or `generate`.

This function used to be called `repeat_until_success`, and is currently aliased as this name as well.

APPLYING A FUNCTION TO A LIST

The `fmap` family of functions provide a way to call a block of code that returns a Future (called here an "item future") once per item in a given list, or returned by a generator function. The `fmap*` functions themselves return a `Future` to represent the ongoing operation, which completes when every item's future has completed.

While this behaviour can also be implemented using `repeat`, the main reason to use an `fmap` function is that the individual item operations are considered as independent, and thus more than one can be outstanding concurrently. An argument can be passed to the function to indicate how many items to start initially, and thereafter it will keep that many of them running concurrently until all of the items are done, or until any of them fail. If an individual item future fails, the overall result future will be marked as failing with the same failure, and any other pending item futures that are outstanding at the time will be cancelled.

The following named arguments are common to each `fmap*` function:

foreach => ARRAY

Provides the list of items to iterate over, as an `ARRAY` reference.

The referenced array will be modified by this operation, `shift`ing one item from it each time. The can `push` more items to this array as it runs, and they will be included in the iteration.

generate => CODE

Provides the list of items to iterate over, by calling the generator function once for each required item. The function should return a single item, or an empty list to indicate it has no more items.

```
( $item ) = $generate->()
```

This function will be invoked each time any previous item future has completed and may be called again even after it has returned empty.

concurrent => INT

Gives the number of item futures to keep outstanding. By default this value will be 1 (i.e. no concurrency); larger values indicate that multiple item futures will be started at once.

return => Future
: Normally, a new instance is returned by cloning the first non-immediate future returned as an item future. By passing a new instance as the `return` argument, the result will be put into the given instance. This can be used to return subclasses, or specific instances.

In each case, the main code block will be called once for each item in the list, passing in the item as the only argument:

```
$item_f = $code->( $item )
```

The expected return value from each item's future, and the value returned from the result future will differ in each function's case; they are documented below.

For similarity with perl's core `map` function, the item is also available aliased as `$_`.

fmap_concat

```
$future = fmap_concat { CODE } ...
```

Since version 0.14.

This version of `fmap` expects each item future to return a list of zero or more values, and the overall result will be the concatenation of all these results. It acts like a future-based equivalent to Perl's `map` operator.

The results are returned in the order of the original input values, not in the order their futures complete in. Because of the intermediate storage of `ARRAY` references and final flattening operation used to implement this behaviour, this function is slightly less efficient than `fmap_scalar` or `fmap_void` in cases where item futures are expected only ever to return one, or zero values, respectively.

This function is also available under the name of simply `fmap` to emphasise its similarity to perl's `map` keyword.

fmap_scalar

```
$future = fmap_scalar { CODE } ...
```

Since version 0.14.

This version of `fmap` acts more like the `map` functions found in Scheme or Haskell; it expects that each item future returns only one value, and the overall result will be a list containing these, in order of the original input items. If an item future returns more than one value the others will be discarded. If it returns no value, then `undef` will be substituted in its place so that the result list remains in correspondence with the input list.

This function is also available under the shorter name of `fmap1`.

fmap_void

```
$future = fmap_void { CODE } ...
```

Since version 0.14.

This version of `fmap` does not collect any results from its item futures, it simply waits for them all to complete. Its result future will provide no values.

While not a map in the strictest sense, this variant is still useful as a way to control concurrency of a function call iterating over a list of items, obtaining its results by some other means (such as side-effects on captured variables, or some external system).

This function is also available under the shorter name of `fmap0`.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>