NAME

"Future" – represent an operation awaiting completion

SYNOPSIS

```
my $future = Future->new;

perform_some_operation(
   on_complete => sub {
      $future->done( @_ );
   }
);

$future->on_ready( sub {
   say "The operation is complete";
} );
```

DESCRIPTION

A `Future` object represents an operation that is currently in progress, or has recently completed. It can be used in a variety of ways to manage the flow of control, and data, through an asynchronous program.

Some futures represent a single operation and are explicitly marked as ready by calling the `done` or `fail` methods. These are called ''leaf'' futures here, and are returned by the `new` constructor.

Other futures represent a collection of sub-tasks, and are implicitly marked as ready depending on the readiness of their component futures as required. These are called ''convergent'' futures here as they converge control and data-flow back into one place. These are the ones returned by the various `wait_*` and `need_*` constructors.

It is intended that library functions that perform asynchronous operations would use future objects to represent outstanding operations, and allow their calling programs to control or wait for these operations to complete. The implementation and the user of such an interface would typically make use of different methods on the class. The methods below are documented in two sections; those of interest to each side of the interface.

It should be noted however, that this module does not in any way provide an actual mechanism for performing this asynchronous activity; it merely provides a way to create objects that can be used for control and data flow around those operations. It allows such code to be written in a neater, forward-reading manner, and simplifies many common patterns that are often involved in such situations.

See also Future::Utils which contains useful loop-constructing functions, to run a future-returning function repeatedly in a loop.

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

FAILURE CATEGORIES

While not directly required by `Future` or its related modules, a growing convention of `Future`-using code is to encode extra semantics in the arguments given to the `fail` method, to represent different kinds of failure.

The convention is that after the initial message string as the first required argument (intended for display to humans), the second argument is a short lowercase string that relates in some way to the kind of failure that occurred. Following this is a list of details about that kind of failure, whose exact arrangement or structure are determined by the failure category. For example, IO::Async and Net::Async::HTTP use this convention to indicate at what stage a given HTTP request has failed:

```
->fail( $message, http => ... )  # an HTTP-level error during protocol
->fail( $message, connect => ... )  # a TCP-level failure to connect a
                                    # socket
->fail( $message, resolve => ... )  # a resolver (likely DNS) failure
                                    # to resolve a hostname
```

By following this convention, a module remains consistent with other `Future`-based modules, and makes

it easy for program logic to gracefully handle and manage failures by use of the `catch` method.

SUBCLASSING

This class easily supports being subclassed to provide extra behavior, such as giving the `get` method the ability to block and wait for completion. This may be useful to provide `Future` subclasses with event systems, or similar.

Each method that returns a new future object will use the invocant to construct its return value. If the constructor needs to perform per-instance setup it can override the `new` method, and take context from the given instance.

```
sub new
{
   my $proto = shift;
   my $self = $proto->SUPER::new;

   if( ref $proto ) {
      # Prototype was an instance
   }
   else {
      # Prototype was a class
   }

   return $self;
}
```

If an instance overrides the "block_until_ready" method, this will be called by `get` and `failure` if the instance is still pending.

In most cases this should allow future-returning modules to be used as if they were blocking call/return−style modules, by simply appending a `get` call to the function or method calls.

```
my ( $results, $here ) = future_returning_function( @args )->get;
```

DEBUGGING

By the time a `Future` object is destroyed, it ought to have been completed or cancelled. By enabling debug tracing of objects, this fact can be checked. If a future object is destroyed without having been completed or cancelled, a warning message is printed.

```
$ PERL_FUTURE_DEBUG=1 perl -MFuture -E 'my $f = Future->new'
Future=HASH(0xaa61f8) was constructed at -e line 1 and was lost near -e line 0 be
```

Note that due to a limitation of perl's `caller` function within a `DESTROY` destructor method, the exact location of the leak cannot be accurately determined. Often the leak will occur due to falling out of scope by returning from a function; in this case the leak location may be reported as being the line following the line calling that function.

```
$ PERL_FUTURE_DEBUG=1 perl -MFuture
sub foo {
   my $f = Future->new;
}

foo();
print "Finished\n";

Future=HASH(0x14a2220) was constructed at - line 2 and was lost near - line 6 bef
Finished
```

A warning is also printed in debug mode if a `Future` object is destroyed that completed with a failure, but the object believes that failure has not been reported anywhere.

```
$ PERL_FUTURE_DEBUG=1 perl -Mblib -MFuture -E 'my $f = Future->fail("Oops")'
Future=HASH(0xac98f8) was constructed at -e line 1 and was lost near -e line 0 wi
```

Such a failure is considered reported if the `get` or `failure` methods are called on it, or it had at least one `on_ready` or `on_fail` callback, or its failure is propagated to another `Future` instance (by a sequencing or converging method).

Future::AsyncAwait::Awaitable ROLE

Since version 0.43 this module provides the Future::AsyncAwait::Awaitable API. Subclass authors should note that several of the API methods are provided by special optimised internal methods, which may require overriding in your subclass if your internals are different from that of this module.

CONSTRUCTORS

new

```
$future = Future->new

$future = $orig->new
```

Returns a new `Future` instance to represent a leaf future. It will be marked as ready by any of the `done`, `fail`, or `cancel` methods. It can be called either as a class method, or as an instance method. Called on an instance it will construct another in the same class, and is useful for subclassing.

This constructor would primarily be used by implementations of asynchronous interfaces.

done *(class method)*

fail *(class method)*

```
$future = Future->done( @values )

$future = Future->fail( $exception, $category, @details )
```

Since version 0.26.

Shortcut wrappers around creating a new `Future` then immediately marking it as done or failed.

wrap

```
$future = Future->wrap( @values )
```

Since version 0.14.

If given a single argument which is already a `Future` reference, this will be returned unmodified. Otherwise, returns a new `Future` instance that is already complete, and will yield the given values.

This will ensure that an incoming argument is definitely a `Future`, and may be useful in such cases as adapting synchronous code to fit asynchronous libraries driven by `Future`.

call

```
$future = Future->call( \&code, @args )
```

Since version 0.15.

A convenient wrapper for calling a `CODE` reference that is expected to return a future. In normal circumstances is equivalent to

```
$future = $code->( @args )
```

except that if the code throws an exception, it is wrapped in a new immediate fail future. If the return value from the code is not a blessed `Future` reference, an immediate fail future is returned instead to complain about this fact.

METHODS

As there are a lare number of methods on this class, they are documented here in several sections.

INSPECTION METHODS

The following methods query the internal state of a Future instance without modifying it or otherwise causing side-effects.

is_ready

```
$ready = $future->is_ready
```

Returns true on a leaf future if a result has been provided to the `done` method, failed using the `fail` method, or cancelled using the `cancel` method.

Returns true on a convergent future if it is ready to yield a result, depending on its component futures.

is_done

```
$done = $future->is_done
```

Returns true on a future if it is ready and completed successfully. Returns false if it is still pending, failed, or was cancelled.

is_failed

```
$failed = $future->is_failed
```

Since version 0.26.

Returns true on a future if it is ready and it failed. Returns false if it is still pending, completed successfully, or was cancelled.

is_cancelled

```
$cancelled = $future->is_cancelled
```

Returns true if the future has been cancelled by `cancel`.

state

```
$str = $future->state
```

Since version 0.36.

Returns a string describing the state of the future, as one of the three states named above; namely `done`, `failed` or `cancelled`, or `pending` if it is none of these.

IMPLEMENTATION METHODS

These methods would primarily be used by implementations of asynchronous interfaces.

done

```
$future->done( @result )
```

Marks that the leaf future is now ready, and provides a list of values as a result. (The empty list is allowed, and still indicates the future as ready). Cannot be called on a convergent future.

If the future is already cancelled, this request is ignored. If the future is already complete with a result or a failure, an exception is thrown.

fail

```
$future->fail( $exception, $category, @details )
```

Marks that the leaf future has failed, and provides an exception value. This exception will be thrown by the `get` method if called.

The exception must evaluate as a true value; false exceptions are not allowed. A failure category name and other further details may be provided that will be returned by the `failure` method in list context.

If the future is already cancelled, this request is ignored. If the future is already complete with a result or a failure, an exception is thrown.

If passed a Future::Exception instance (i.e. an object previously thrown by the `get`), the additional details will be preserved. This allows the additional details to be transparently preserved by such code as

```
...
catch {
   return Future->fail($@);
}
```

die

```
$future->die( $message, $category, @details )
```

Since version 0.09.

A convenient wrapper around `fail`. If the exception is a non-reference that does not end in a linefeed, its value will be extended by the file and line number of the caller, similar to the logic that `die` uses.

Returns the `$future`.

on_cancel

```
$future->on_cancel( $code )
```

If the future is not yet ready, adds a callback to be invoked if the future is cancelled by the `cancel` method. If the future is already ready the method is ignored.

If the future is later cancelled, the callbacks will be invoked in the reverse order to that in which they were registered.

```
$on_cancel->( $future )
```

If passed another `Future` instance, the passed instance will be cancelled when the original future is cancelled. In this case, the reference is only strongly held while the target future remains pending. If it becomes ready, then there is no point trying to cancel it, and so it is removed from the originating future's cancellation list.

USER METHODS

These methods would primarily be used by users of asynchronous interfaces, on objects returned by such an interface.

on_ready

```
$future->on_ready( $code )
```

If the future is not yet ready, adds a callback to be invoked when the future is ready. If the future is already ready, invokes it immediately.

In either case, the callback will be passed the future object itself. The invoked code can then obtain the list of results by calling the `get` method.

```
$on_ready->( $future )
```

If passed another `Future` instance, the passed instance will have its `done`, `fail` or `cancel` methods invoked when the original future completes successfully, fails, or is cancelled respectively.

Returns the `$future`.

get

```
@result = $future->get

$result = $future->get
```

If the future is ready and completed successfully, returns the list of results that had earlier been given to the `done` method on a leaf future, or the list of component futures it was waiting for on a convergent future. In scalar context it returns just the first result value.

If the future is ready but failed, this method raises as an exception the failure that was given to the `fail` method. If additional details were given to the `fail` method, an exception object is constructed to wrap them of type Future::Exception.

If the future was cancelled an exception is thrown.

If it is not yet ready then ''block_until_ready'' is invoked to wait for a ready state.

block_until_ready

```
$f = $f->block_until_ready
```

Since version 0.40.

Blocks until the future instance is no longer pending.

Returns the invocant future itself, so it is useful for chaining.

Usually, calling code would either force the future using ''get'', or use either `then` chaining or `async/await` syntax to wait for results. This method is useful in cases where the exception-throwing part of `get` is not required, perhaps because other code will be testing the result using ''is_done'' or similar.

```
if( $f->block_until_ready->is_done ) {
   ...
}
```

This method is intended for subclasses to override, but a default implementation for back-compatibility purposes is provided which calls the `await` method. If the future is not yet ready, attempts to wait for an eventual result by using the underlying `await` method, which subclasses should provide. The default implementation will throw an exception if called on a still-pending instance that does not provide an `await` method.

unwrap

```
@values = Future->unwrap( @values )
```

Since version 0.26.

If given a single argument which is a `Future` reference, this method will call `get` on it and return the result. Otherwise, it returns the list of values directly in list context, or the first value in scalar. Since it involves an implicit blocking wait, this method can only be used on immediate futures or subclasses that implement ''block_until_ready''.

This will ensure that an outgoing argument is definitely not a `Future`, and may be useful in such cases as adapting synchronous code to fit asynchronous libraries that return `Future` instances.

on_done

```
$future->on_done( $code )
```

If the future is not yet ready, adds a callback to be invoked when the future is ready, if it completes successfully. If the future completed successfully, invokes it immediately. If it failed or was cancelled, it is not invoked at all.

The callback will be passed the result passed to the `done` method.

```
$on_done->( @result )
```

If passed another `Future` instance, the passed instance will have its `done` method invoked when the original future completes successfully.

Returns the `$future`.

failure

```
$exception = $future->failure

$exception, $category, @details = $future->failure
```

If the future is ready, returns the exception passed to the `fail` method or `undef` if the future completed successfully via the `done` method.

If it is not yet ready then ''block_until_ready'' is invoked to wait for a ready state.

If called in list context, will additionally yield the category name and list of the details provided to the `fail` method.

Because the exception value must be true, this can be used in a simple `if` statement:

```
if( my $exception = $future->failure ) {
   ...
}
else {
   my @result = $future->get;
   ...
}
```

on_fail

```
$future->on_fail( $code )
```

If the future is not yet ready, adds a callback to be invoked when the future is ready, if it fails. If the future has already failed, invokes it immediately. If it completed successfully or was cancelled, it is not invoked at all.

The callback will be passed the exception and other details passed to the `fail` method.

```
$on_fail->( $exception, $category, @details )
```

If passed another `Future` instance, the passed instance will have its `fail` method invoked when the original future fails.

To invoke a `done` method on a future when another one fails, use a CODE reference:

```
$future->on_fail( sub { $f->done( @_ ) } );
```

Returns the `$future`.

cancel

```
$future->cancel
```

Requests that the future be cancelled, immediately marking it as ready. This will invoke all of the code blocks registered by `on_cancel`, in the reverse order. When called on a convergent future, all its component futures are also cancelled. It is not an error to attempt to cancel a future that is already complete or cancelled; it simply has no effect.

Returns the `$future`.

SEQUENCING METHODS

The following methods all return a new future to represent the combination of its invocant followed by another action given by a code reference. The combined activity waits for the first future to be ready, then may invoke the code depending on the success or failure of the first, or may run it regardless. The returned sequence future represents the entire combination of activity.

In some cases the code should return a future; in some it should return an immediate result. If a future is returned, the combined future will then wait for the result of this second one. If the combinined future is cancelled, it will cancel either the first future or the second, depending whether the first had completed. If the code block throws an exception instead of returning a value, the sequence future will fail with that exception as its message and no further values.

As it is always a mistake to call these sequencing methods in void context and lose the reference to the returned future (because exception/error handling would be silently dropped), this method warns in void context.

then

```
$future = $f1->then( \&done_code )
```

Since version 0.13.

Returns a new sequencing `Future` that runs the code if the first succeeds. Once `$f1` succeeds the code reference will be invoked and is passed the list of results. It should return a future, `$f2`. Once `$f2` completes the sequence future will then be marked as complete with whatever result `$f2` gave. If `$f1` fails then the sequence future will immediately fail with the same failure and the code will not be invoked.

```
$f2 = $done_code->( @result )
```

else

```
$future = $f1->else( \&fail_code )
```

Since version 0.13.

Returns a new sequencing `Future` that runs the code if the first fails. Once `$f1` fails the code reference will be invoked and is passed the failure and other details. It should return a future, `$f2`. Once `$f2` completes the sequence future will then be marked as complete with whatever result `$f2` gave. If `$f1` succeeds then the sequence future will immediately succeed with the same result and the code will not be invoked.

```
$f2 = $fail_code->( $exception, $category, @details )
```

then *(2 arguments)*

```
$future = $f1->then( \&done_code, \&fail_code )
```

The `then` method can also be passed the `$fail_code` block as well, giving a combination of `then` and `else` behaviour.

This operation is designed to be compatible with the semantics of other future systems, such as Javascript's Q or Promises/A libraries.

catch

```
$future = $f1->catch(
   name => \&code,
   name => \&code, ...
)
```

Since version 0.33.

Returns a new sequencing `Future` that behaves like an `else` call which dispatches to a choice of several alternative handling functions depending on the kind of failure that occurred. If `$f1` fails with a category name (i.e. the second argument to the `fail` call) which exactly matches one of the string names given, then the corresponding code is invoked, being passed the same arguments as a plain `else` call would take, and is expected to return a `Future` in the same way.

```
$f2 = $code->( $exception, $category, @details )
```

If `$f1` does not fail, fails without a category name at all, or fails with a category name that does not match any given to the `catch` method, then the returned sequence future immediately completes with the same result, and no block of code is invoked.

If passed an odd-sized list, the final argument gives a function to invoke on failure if no other handler matches.

```
$future = $f1->catch(
   name => \&code, ...
   \&fail_code,
)
```

This feature is currently still a work-in-progress. It currently can only cope with category names that are literal strings, which are all distinct. A later version may define other kinds of match (e.g. regexp), may specify some sort of ordering on the arguments, or any of several other semantic extensions. For more detail on the ongoing design, see <https://rt.cpan.org/Ticket/Display.html?id=103545>.

then *(multiple arguments)*

```
$future = $f1->then( \&done_code, @catch_list, \&fail_code )
```

Since version 0.33.

The `then` method can be passed an even-sized list inbetween the `$done_code` and the `$fail_code`, with the same meaning as the `catch` method.

transform

```
$future = $f1->transform( %args )
```

Returns a new sequencing `Future` that wraps the one given as `$f1`. With no arguments this will be a trivial wrapper; `$future` will complete or fail when `$f1` does, and `$f1` will be cancelled when `$future` is.

By passing the following named arguments, the returned `$future` can be made to behave differently to `$f1`:

done => CODE

Provides a function to use to modify the result of a successful completion. When `$f1` completes successfully, the result of its `get` method is passed into this function, and whatever it returns is passed to the `done` method of `$future`

fail => CODE

Provides a function to use to modify the result of a failure. When `$f1` fails, the result of its `failure` method is passed into this function, and whatever it returns is passed to the `fail` method of `$future`.

then_with_f

```
$future = $f1->then_with_f( ... )
```

Since version 0.21.

Returns a new sequencing `Future` that behaves like `then`, but also passes the original future, `$f1`, to any functions it invokes.

```
$f2 = $done_code->( $f1, @result )
$f2 = $catch_code->( $f1, $category, @details )
$f2 = $fail_code->( $f1, $category, @details )
```

This is useful for conditional execution cases where the code block may just return the same result of the original future. In this case it is more efficient to return the original future itself.

then_done
then_fail

```
$future = $f->then_done( @result )

$future = $f->then_fail( $exception, $category, @details )
```

Since version 0.22.

Convenient shortcuts to returning an immediate future from a `then` block, when the result is already known.

else_with_f

```
$future = $f1->else_with_f( \&code )
```

Since version 0.21.

Returns a new sequencing `Future` that runs the code if the first fails. Identical to `else`, except that the code reference will be passed both the original future, `$f1`, and its exception and other details.

```
$f2 = $code->( $f1, $exception, $category, @details )
```

This is useful for conditional execution cases where the code block may just return the same result of the original future. In this case it is more efficient to return the original future itself.

else_done
else_fail

```
$future = $f->else_done( @result )

$future = $f->else_fail( $exception, $category, @details )
```

Since version 0.22.

Convenient shortcuts to returning an immediate future from a `else` block, when the result is already known.

catch_with_f

```
$future = $f1->catch_with_f( ... )
```

Since version 0.33.

Returns a new sequencing `Future` that behaves like `catch`, but also passes the original future, `$f1`, to any functions it invokes.

followed_by

```
$future = $f1->followed_by( \&code )
```

Returns a new sequencing `Future` that runs the code regardless of success or failure. Once `$f1` is ready the code reference will be invoked and is passed one argument, `$f1`. It should return a future, `$f2`. Once `$f2` completes the sequence future will then be marked as complete with whatever result `$f2` gave.

```
$f2 = $code->( $f1 )
```

without_cancel

```
$future = $f1->without_cancel
```

Since version 0.30.

Returns a new sequencing `Future` that will complete with the success or failure of the original future, but if cancelled, will not cancel the original. This may be useful if the original future represents an operation that is being shared among multiple sequences; cancelling one should not prevent the others from running too.

retain

```
$f = $f->retain
```

Since version 0.36.

Creates a reference cycle which causes the future to remain in memory until it completes. Returns the invocant future.

In normal situations, a `Future` instance does not strongly hold a reference to other futures that it is feeding a result into, instead relying on that to be handled by application logic. This is normally fine because some part of the application will retain the top-level Future, which then strongly refers to each of its components down in a tree. However, certain design patterns, such as mixed Future-based and legacy callback-based API styles might end up creating Futures simply to attach callback functions to them. In that situation, without further attention, the Future may get lost due to having no strong references to it. Calling `->retain` on it creates such a reference which ensures it persists until it completes. For example:

```
Future->needs_all( $fA, $fB )
   ->on_done( $on_done )
   ->on_fail( $on_fail )
   ->retain;
```

CONVERGENT FUTURES

The following constructors all take a list of component futures, and return a new future whose readiness somehow depends on the readiness of those components. The first derived class component future will be used as the prototype for constructing the return value, so it respects subclassing correctly, or failing that a plain `Future`.

wait_all

```
$future = Future->wait_all( @subfutures )
```

Returns a new `Future` instance that will indicate it is ready once all of the sub future objects given to it indicate that they are ready, either by success, failure or cancellation. Its result will be a list of its component futures.

When given an empty list this constructor returns a new immediately-done future.

This constructor would primarily be used by users of asynchronous interfaces.

wait_any

```
$future = Future->wait_any( @subfutures )
```

Returns a new `Future` instance that will indicate it is ready once any of the sub future objects given to it indicate that they are ready, either by success or failure. Any remaining component futures that are not yet ready will be cancelled. Its result will be the result of the first component future that was ready; either success or failure. Any component futures that are cancelled are ignored, apart from the final component left; at which point the result will be a failure.

When given an empty list this constructor returns an immediately-failed future.

This constructor would primarily be used by users of asynchronous interfaces.

needs_all

```
$future = Future->needs_all( @subfutures )
```

Returns a new `Future` instance that will indicate it is ready once all of the sub future objects given to it indicate that they have completed successfully, or when any of them indicates that they have failed. If any sub future fails, then this will fail immediately, and the remaining subs not yet ready will be cancelled. Any component futures that are cancelled will cause an immediate failure of the result.

If successful, its result will be a concatenated list of the results of all its component futures, in corresponding order. If it fails, its failure will be that of the first component future that failed. To access each component future's results individually, use `done_futures`.

When given an empty list this constructor returns a new immediately-done future.

This constructor would primarily be used by users of asynchronous interfaces.

needs_any

```
$future = Future->needs_any( @subfutures )
```

Returns a new `Future` instance that will indicate it is ready once any of the sub future objects given to it indicate that they have completed successfully, or when all of them indicate that they have failed. If any sub future succeeds, then this will succeed immediately, and the remaining subs not yet ready will be cancelled. Any component futures that are cancelled are ignored, apart from the final component left; at which point the result will be a failure.

If successful, its result will be that of the first component future that succeeded. If it fails, its failure will be that of the last component future to fail. To access the other failures, use `failed_futures`.

Normally when this future completes successfully, only one of its component futures will be done. If it is constructed with multiple that are already done however, then all of these will be returned from `done_futures`. Users should be careful to still check all the results from `done_futures` in that case.

When given an empty list this constructor returns an immediately-failed future.

This constructor would primarily be used by users of asynchronous interfaces.

METHODS ON CONVERGENT FUTURES

The following methods apply to convergent (i.e. non-leaf) futures, to access the component futures stored by it.

pending_futures

```
@f = $future->pending_futures
```

ready_futures

```
@f = $future->ready_futures
```

done_futures

```
@f = $future->done_futures
```

failed_futures

```
@f = $future->failed_futures
```

cancelled_futures

```
@f = $future->cancelled_futures
```

Return a list of all the pending, ready, done, failed, or cancelled component futures. In scalar context, each will yield the number of such component futures.

TRACING METHODS

set_label

label

```
$future = $future->set_label( $label )

$label = $future->label
```

Since version 0.28.

Chaining mutator and accessor for the label of the `Future`. This should be a plain string value, whose value will be stored by the future instance for use in debugging messages or other tooling, or similar purposes.

btime

rtime

```
[ $sec, $usec ] = $future->btime

[ $sec, $usec ] = $future->rtime
```

Since version 0.28.

Accessors that return the tracing timestamps from the instance. These give the time the instance was constructed ("birth" time, `btime`) and the time the result was determined (the "ready" time, `rtime`). Each result is returned as a two-element ARRAY ref, containing the epoch time in seconds and microseconds, as given by `Time::HiRes::gettimeofday`.

In order for these times to be captured, they have to be enabled by setting `$Future::TIMES` to a true value. This is initialised true at the time the module is loaded if either `PERL_FUTURE_DEBUG` or `PERL_FUTURE_TIMES` are set in the environment.

elapsed

```
$sec = $future->elapsed
```

Since version 0.28.

If both tracing timestamps are defined, returns the number of seconds of elapsed time between them as a floating-point number. If not, returns `undef`.

wrap_cb

```
$cb = $future->wrap_cb( $operation_name, $cb )
```

Since version 0.31.

Note: This method is experimental and may be changed or removed in a later version.

This method is invoked internally by various methods that are about to save a callback CODE reference supplied by the user, to be invoked later. The default implementation simply returns the callback argument as-is; the method is provided to allow users to provide extra behaviour. This can be done by applying a method modifier of the `around` kind, so in effect add a chain of wrappers. Each wrapper can then perform its own wrapping logic of the callback. `$operation_name` is a string giving the reason for which the callback is being saved; currently one of `on_ready`, `on_done`, `on_fail` or `sequence`; the latter being used for all the sequence-returning methods.

This method is intentionally invoked only for CODE references that are being saved on a pending `Future` instance to be invoked at some later point. It does not run for callbacks to be invoked on an already-complete instance. This is for performance reasons, where the intended behaviour is that the wrapper can

provide some amount of context save and restore, to return the operating environment for the callback back to what it was at the time it was saved.

For example, the following wrapper saves the value of a package variable at the time the callback was saved, and restores that value at invocation time later on. This could be useful for preserving context during logging in a Future-based program.

```
our $LOGGING_CTX;

no warnings 'redefine';

my $orig = Future->can( "wrap_cb" );
*Future::wrap_cb = sub {
   my $cb = $orig->( @_ );

   my $saved_logging_ctx = $LOGGING_CTX;

   return sub {
      local $LOGGING_CTX = $saved_logging_ctx;
      $cb->( @_ );
   };
};
```

At this point, any code deferred into a `Future` by any of its callbacks will observe the `$LOGGING_CTX` variable as having the value it held at the time the callback was saved, even if it is invoked later on when that value is different.

Remember when writing such a wrapper, that it still needs to invoke the previous version of the method, so that it plays nicely in combination with others (see the `$orig->( @_ )` part).

EXAMPLES

The following examples all demonstrate possible uses of a `Future` object to provide a fictional asynchronous API.

For more examples, comparing the use of `Future` with regular call/return style Perl code, see also Future::Phrasebook.

Providing Results

By returning a new `Future` object each time the asynchronous function is called, it provides a placeholder for its eventual result, and a way to indicate when it is complete.

```
sub foperation
{
   my %args = @_;

   my $future = Future->new;

   do_something_async(
      foo => $args{foo},
      on_done => sub { $future->done( @_ ); },
   );

   return $future;
}
```

In most cases, the `done` method will simply be invoked with the entire result list as its arguments. In that case, it is convenient to use the curry module to form a `CODE` reference that would invoke the `done` method.

```
 my $future = Future->new;
```

```
 do_something_async(
    foo => $args{foo},
    on_done => $future->curry::done,
 );
```

The caller may then use this future to wait for a result using the `on_ready` method, and obtain the result using `get`.

```
my $f = foperation( foo => "something" );

$f->on_ready( sub {
   my $f = shift;
   say "The operation returned: ", $f->get;
} );
```

Indicating Success or Failure

Because the stored exception value of a failed future may not be false, the `failure` method can be used in a conditional statement to detect success or failure.

```
my $f = foperation( foo => "something" );

$f->on_ready( sub {
   my $f = shift;
   if( not my $e = $f->failure ) {
      say "The operation succeeded with: ", $f->get;
   }
   else {
      say "The operation failed with: ", $e;
   }
} );
```

By using `not` in the condition, the order of the `if` blocks can be arranged to put the successful case first, similar to a `try`/`catch` block.

Because the `get` method re-raises the passed exception if the future failed, it can be used to control a `try`/`catch` block directly. (This is sometimes called *Exception Hoisting*).

```
use Syntax::Keyword::Try;

$f->on_ready( sub {
   my $f = shift;
   try {
      say "The operation succeeded with: ", $f->get;
   }
   catch {
      say "The operation failed with: ", $_;
   }
} );
```

Even neater still may be the separate use of the `on_done` and `on_fail` methods.

```
$f->on_done( sub {
   my @result = @_;
   say "The operation succeeded with: ", @result;
} );
$f->on_fail( sub {
   my ( $failure ) = @_;
   say "The operation failed with: $failure";
} );
```

Immediate Futures

Because the `done` method returns the future object itself, it can be used to generate a `Future` that is immediately ready with a result. This can also be used as a class method.

```
my $f = Future->done( $value );
```

Similarly, the `fail` and `die` methods can be used to generate a `Future` that is immediately failed.

```
my $f = Future->die( "This is never going to work" );
```

This could be considered similarly to a `die` call.

An `eval{}` block can be used to turn a `Future`-returning function that might throw an exception, into a `Future` that would indicate this failure.

```
my $f = eval { function() } || Future->fail( $@ );
```

This is neater handled by the `call` class method, which wraps the call in an `eval{}` block and tests the result:

```
my $f = Future->call( \&function );
```

Sequencing

The `then` method can be used to create simple chains of dependent tasks, each one executing and returning a `Future` when the previous operation succeeds.

```
my $f = do_first()
            ->then( sub {
               return do_second();
            })
            ->then( sub {
               return do_third();
            });
```

The result of the `$f` future itself will be the result of the future returned by the final function, if none of them failed. If any of them fails it will fail with the same failure. This can be considered similar to normal exception handling in synchronous code; the first time a function call throws an exception, the subsequent calls are not made.

Merging Control Flow

A `wait_all` future may be used to resynchronise control flow, while waiting for multiple concurrent operations to finish.

```
my $f1 = foperation( foo => "something" );
my $f2 = foperation( bar => "something else" );

my $f = Future->wait_all( $f1, $f2 );

$f->on_ready( sub {
   say "Operations are ready:";
   say "  foo: ", $f1->get;
   say "  bar: ", $f2->get;
} );
```

This provides an ability somewhat similar to `CPS::kpar()` or Async::MergePoint.

KNOWN ISSUES

Cancellation of Non-Final Sequence Futures

The behaviour of future cancellation still has some unanswered questions regarding how to handle the situation where a future is cancelled that has a sequence future constructed from it.

In particular, it is unclear in each of the following examples what the behaviour of `$f2` should be, were `$f1` to be cancelled:

```
$f2 = $f1->then( sub { ... } ); # plus related ->then_with_f, ...

$f2 = $f1->else( sub { ... } ); # plus related ->else_with_f, ...

$f2 = $f1->followed_by( sub { ... } );
```

In the `then`-style case it is likely that this situation should be treated as if `$f1` had failed, perhaps with some special message. The `else`-style case is more complex, because it may be that the entire operation should still fail, or it may be that the cancellation of `$f1` should again be treated simply as a special kind of failure, and the `else` logic run as normal.

To be specific; in each case it is unclear what happens if the first future is cancelled, while the second one is still waiting on it. The semantics for "normal" top-down cancellation of `$f2` and how it affects `$f1` are already clear and defined.

Cancellation of Divergent Flow

A further complication of cancellation comes from the case where a given future is reused multiple times for multiple sequences or convergent trees.

In particular, it is in clear in each of the following examples what the behaviour of `$f2` should be, were `$f1` to be cancelled:

```
my $f_initial = Future->new; ...
my $f1 = $f_initial->then( ... );
my $f2 = $f_initial->then( ... );

my $f1 = Future->needs_all( $f_initial );
my $f2 = Future->needs_all( $f_initial );
```

The point of cancellation propagation is to trace backwards through stages of some larger sequence of operations that now no longer need to happen, because the final result is no longer required. But in each of these cases, just because `$f1` has been cancelled, the initial future `$f_initial` is still required because there is another future (`$f2`) that will still require its result.

Initially it would appear that some kind of reference-counting mechanism could solve this question, though that itself is further complicated by the `on_ready` handler and its variants.

It may simply be that a comprehensive useful set of cancellation semantics can't be universally provided to cover all cases; and that some use-cases at least would require the application logic to give extra information to its `Future` objects on how they should wire up the cancel propagation logic.

Both of these cancellation issues are still under active design consideration; see the discussion on RT96685 for more information (<https://rt.cpan.org/Ticket/Display.html?id=96685>).

SEE ALSO

- Future::AsyncAwait – deferred subroutine syntax for futures

  Provides a neat syntax extension for writing future-based code.

- Future::IO – Future-returning IO methods

  Provides methods similar to core IO functions, which yield results by Futures.

- Promises – an implementation of the ‘‘Promise/A+’’ pattern for asynchronous programming

  A different alternative implementation of a similar idea.

- curry – Create automatic curried method call closures for any class or object

- ‘‘The Past, The Present and The Future’’ – slides from a talk given at the London Perl Workshop, 2012.

  <https://docs.google.com/presentation/d/1UkV5oLcTOOXBXPh8foyxko4PR28_zU_aVx6gBms7uoo/edit>

- ‘‘Futures advent calendar 2013’’

  <http://leonerds-code.blogspot.co.uk/2013/12/futures-advent-day-1.html>

- ‘‘Asynchronous Programming with Futures’’ – YAPC::EU 2014

  <https://www.youtube.com/watch?v=u9dZgFM6FtE>

TODO

- Consider the ability to pass the constructor a `block` CODEref, instead of needing to use a subclass. This might simplify async/etc.. implementations, and allows the reuse of the idea of subclassing to extend the abilities of `Future` itself – for example to allow a kind of Future that can report incremental progress.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>