mirai Documentation

Release 0.1

Daniel Duckworth

June 22, 2014

Contents

CHAPTER 1

Welcome to mirai

`mirai` is a multithreading library for Python that makes asynchronous computation a breeze. Built on `concurrent.futures` and modeled after Twitter Futures, `mirai` helps you write modular, easy-to-read asynchronous workflows without falling into callback hell.

What can `mirai` do for you? Here's a demo for fetching the weather forecast for San Francisco with a 10 second timeout,

```
from mirai import Promise
from pprint import pprint
import json
import requests

url    = "http://api.openweathermap.org/data/2.5/forecast"
query  = {"q": "San Francisco", "units": "imperial"}
result = (
  Promise.call(lambda: requests.get(url, params=query))
  .onsuccess(lambda response: pprint("Success!"))
  .map(lambda response: json.loads(response.text))
  .map(lambda weather: {
    "status": "success",
    "forecast": sorted([
      {
        "time"    : f['dt_txt'],
        "weather" : f['weather'][0]['description'],
        "temp"    : f['main']['temp'],
      } for f in weather['list']
    ])
  })
  .within(10.0)
  .handle(lambda e: {"status": "failure", "reason": unicode(e) })
  .get()
)

pprint(result)
```

You can install the library with,

```
$ pip install mirai
```

CHAPTER 2

Documentation

2.1 Why mirai?

Above all, `mirai` aims to make asynchronous code modular. The result of this is code that looks not unlike synchronous code – perhaps even cleaner. I'll illustrate with a simple example.

A common use case for multithreading is when your code is IO-bound. For example, the following code fetches a set of webpages within a timeout, then ranks them according to fetch time.

```
import time

from mirai import Promise
import requests


def fetch(url):
  start    = time.time()
  response = requests.get(url)
  response.raise_for_status()
  return (time.time() - start, url)


def fetch_within(url, timeout):
  return (
    Promise.call(fetch, url)
    .within(timeout)
    .handle(lambda e: (float('inf'), url))
  )


def fetch_times(urls, timeout):
  promises = [ fetch_within(url, timeout) for url in urls ]
  return sorted(Promise.collect(promises).get())
```

In total, we have 24 lines. Notice that exception handling is done independent of time calculation, and that there's no need to think about locking or queues.

2.1.1 Why not threading?

`threading` is Python's low-level library for multithreaded code. It's extremely scant in its offering and requires significant attention to locking, timing, and racing threads, obscuring the program's actual intent. The following 48(!) lines implement equivalent logic, employing a `Queue` to pass information between threads,

```
from Queue import Queue
from threading import Thread, Timer, Lock
import time

import requests


class FetchThread(Thread):

  def __init__(self, url, queue, timeout):
    super(FetchThread, self).__init__()
    self.url       = url
    self.queue     = queue
    self.timeout   = timeout
    self.lock      = Lock()
    self.submitted = False

  def run(self):
    timer = Timer(self.timeout, self._submit, args=(float('inf'),))
    timer.start()

    start = time.time()
    try:
      response = requests.get(self.url)
      response.raise_for_status()
      self._submit( time.time() - start )
    except Exception as e:
      self._submit(float('inf'))

  def _submit(self, elapsed):
    with self.lock:
      if not self.submitted:
        self.submitted = True
        self.queue.put( (elapsed, self.url) )


def fetch_async(url, queue, timeout):
  thread = FetchThread(url, queue, timeout)
  thread.start()
  return thread


def fetch_times(urls, timeout):
  queue = Queue()
  threads = [fetch_async(url, queue, timeout=timeout) for url in urls]
  [thread.join() for thread in threads]

  return sorted([queue.get() for url in urls])
```

2.1.2 Why not concurrent.futures?

`concurrent.futures` is the new asynchronous computation library added to Python's standard library in version 3.2. While the library offers the same core benefits of *mirai*, it lacks the method-chaining additions that make working with futures a breeze. The following 27 lines of code illustrate the same logic,

```
from concurrent.futures import ThreadPoolExecutor, wait
import time

import requests


EXECUTOR = ThreadPoolExecutor(max_workers=10)

def fetch_sync(url):
    start    = time.time()
    try:
      response = requests.get(url)
      response.raise_for_status()
      return (time.time() - start, url)
    except Exception as e:
      return (float('inf'), url)


def fetch_times(urls, timeout):
  threads = [EXECUTOR.submit(fetch_sync, url) for url in urls]
  complete, incomplete = wait(threads, timeout=timeout)
  results = [future.result() for future in complete]
  result_urls = set(r[1] for r in results)
  for url in urls:
    if url not in result_urls:
      results.append( (float('inf'), url) )
  return sorted(results)
```

2.1.3 Why not multiprocessing?

`multiprocessing` and *mirai* actually achieve different things and actually have very little overlap. Whereas *mirai* is designed to speed up *IO-bound* code, whereas *multiprocessing* is designed to speed up *CPU-bound* code. If the latter sounds more like what you're looking for, **mirai cannot help you!** as it still bound by the GIL. Instead, you should take a look at *multiprocessing*, celery, or joblib.

2.1.4 Why not gevent?

`gevent` replaces Python's default threads with "greenlets" managed by *libev*. The value in using *gevent* is that one can generate thousands of greenlets and still maintain a performant asynchronous system. Used directly, *gevent* is not dissimilar from *concurrent.futures*, but does require more work to compose results. The following 28 lines of code illustrate.

```
from gevent.monkey import patch_all; patch_all()

import gevent
import time

import requests


def fetch_sync(url):
    start    = time.time()
    try:
      response = requests.get(url)
      response.raise_for_status()
```

```
      return (time.time() - start, url)
    except Exception as e:
      return (float('inf'), url)


def fetch_times(urls, timeout):
  threads  = [gevent.spawn(fetch_sync, url) for url in urls]
  gevent.joinall(threads, timeout=timeout)
  results = []
  for (url, thread) in zip(urls, threads):
    try:
      results.append( thread.get(timeout=0) )
    except gevent.Timeout:
      results.append( (float('inf'), url) )
  return sorted(results)
```

"But *gevent* uses *libev*, which is way more scalable than any of the other alternatives, including *mirai*!" you might say, but fear not – *mirai* (and *threading* and *concurrent.futures*) can use greenlets by monkey patching the standard library at the start of your code. Simply put the following line at the top of your main script, before any other import statements,

```
from gevent.monkey import patch_all; patch_all()
```

Now *mirai* has all the performance benefits of greenlets!

2.2 Tutorial

The primary benefit of working with `mirai` is the ability to write asynchronous code much the same way you already write synchronous code. We'll illustrate this by writing a simple web scraper, step-by-step, with and without `mirai`.

2.2.1 Fetching a Page

We begin with the most basic task for any web scraper – fetching a single web page. Rather than directly returning the page's contents, we'll return a `Success` container indicating that our request went through successfully. Similarly, we'll return an `Error` container if the request failed.

Using a function `urlget()`, which returns a response if a request succeeds and raises an exception if a request fails, we can start with the following two *fetch* functions,

```
from commons import *


def fetch_sync(url):
  try:
    response = urlget(url)
    return Success(url, response)
  except Exception as e:
    return Error(url, e)


def fetch_async(url):
  return (
    Promise
    .call  (urlget, url)
    .map   (lambda response : Success(url, response))
```

```
    .handle(lambda error    : Error   (url, error   ))
  )
```

2.2.2 Retrying on Failure

Sometimes, an fetch failure is simply transient; that is to say, if we simply retry we may be able to fetch the page. Using recursion, let's add an optional *retries* argument to our *fetch* functions,

```
from commons import *


def fetch_sync(url, retries=3):
  try:
    response = urlget(url)
    return Success(url, response)
  except Exception as e:
    if retries > 0:
      return fetch_sync(url, retries-1)
    else:
      return Error(url, e)


def fetch_async(url, retries=3):
  return (
    Promise
    .call     (urlget, url)
    .map      (lambda response : Success(url, response))
    .rescue   (lambda error    :
      fetch_async(url, retries-1)
      if   retries > 0
      else Promise.value(Error(url, error))
    )
  )
```

2.2.3 Handling Timeouts

Another common concern is time – if a page takes too long to fetch, we may rather consider it a loss rather than wait for it to finish downloading. Let's construct a new container called `Timeout` that will indicate that a page took too long to retrieve. We'll give *fetch* a *finish_by* argument specifying when, in time, we want the function to return by.

You'll notice that rather than telling the *fetch* functions how much time they have, we give them a deadline by which they must finish. This is because relative durations become easily muddled in asynchronous code when functions are called with a delay.

```
from commons import *


def fetch_sync(url, finish_by, retries=3):
  remaining = finish_by - time.time()

  if remaining <= 0:
    return Timeout(url, None)

  try:
    response = urlget(url, finish_by)
```

```
    return Success(url, response)
  except Exception as e:
    if retries > 0:
      return fetch_sync(url, finish_by, retries-1)
    else:
      if isinstance(e, requests.exceptions.Timeout):
        return Timeout(url, e)
      else:
        return Error(url, e)


def fetch_async(url, finish_by, retries=3):
  remaining = finish_by - time.time()

  if remaining < 0:
    return Promise.value(Timeout(url, None))

  return (
    Promise
    .call     (urlget, url, finish_by)
    .map      (lambda response : Success(url, response))
    .rescue   (lambda error    :
      fetch_async(url, finish_by, retries-1)
      if   retries > 0
      else Promise.value(Timeout(url, error))
           if   isinstance(error, requests.exceptions.Timeout)
           else Promise.value(Error(url, error))
    )
  )
```

2.2.4 Scraping Links

Finally, let's complete our scraper by following each page's links. To keep our code from running forever, we'll only follow links up to a fixed maximum depth. Moreover, we'll add a *finish_by* to limit the amount of time until the function returns.

This is where `mirai`'s asynchronous nature really shines. While the synchronous version must fetch each page one at a time, `mirai` makes it easy to fetch pages in parallel with minimal change to the source,

```
from commons import *
from tutorial04 import fetch_sync, fetch_async


def scrape_sync(url, finish_by, retries=3, maxdepth=0):
  remaining = finish_by - time.time()
  if   remaining <= 0:
    return [Timeout(url, None)]
  elif maxdepth  == 0:
    return [fetch_sync(url, finish_by, retries)]
  elif maxdepth   < 0:
    return []
  else:
    status   = fetch_sync(url, finish_by, retries)
    if isinstance(status, Success):
      linkset  = links(url, status.response.text)
      children = [
        scrape_sync(link, finish_by, retries, maxdepth-1)
```

```
        for link in linkset
      ]
      return fu.cat([[status]] + children)
    else:
      return [status]


def scrape_async(url, finish_by, retries=3, maxdepth=0):
  remaining = finish_by - time.time()
  if   remaining <= 0:
    return Promise.value([Timeout(url, None)])
  elif maxdepth  == 0:
    return (
      fetch_async(url, finish_by, retries)
      .map(lambda status: [status])
    )
  elif maxdepth   < 0:
    return Promise.value([])
  else:
    status  = fetch_async(url, finish_by, retries)

    children = (
      status
      .map(lambda status: \
        links(url, status.response.text)
        if   isinstance(status, Success)
        else []
      )
      .map(lambda linkset: [
        scrape_async(link, finish_by, retries, maxdepth-1)
        for link in linkset
      ])
      .flatmap(Promise.collect)
    )

    return (
      status.join_(children)
      .map(lambda (status, children): [[status]] + children)
      .map(fu.cat)
    )
```

2.2.5 Wrapping Up

We now have a fully functional web scraper, capable of handling timeouts and retrying on failure. To try this scraper out for yourself, download the code in the [tutorial folder](https://github.com/duckworthd/mirai/tree/develop/docs/_tutorial) and see for yourself how `mirai` can make your life easier!

2.3 API

This part of the documentation shows the full API reference of all public classes and functions.

2.3.1 Creating Promises

classmethod `Promise.`**`value`**(*val*)
Construct a Promise that is already resolved successfully to a value.

Parameters **val** : anything

Value to resolve new Promise to.

Returns **result** : Future

Future containing *val* as its value.

classmethod `Promise.`**`exception`**(*exc*)
Construct a Promise that has already failed with a given exception.

Parameters **exc** : Exception

Exception to fail new Promise with

Returns **result** : Future

New Promise that has already failed with the given exception.

classmethod `Promise.`**`call`**(*fn*, **args*, ***kwargs*)
Call a function asynchronously and return a Promise with its result. If an exception is thrown inside *fn*, a new exception type will be constructed inheriting both from *MiraiError* and the exception's original type. The new exception is the same the original, except that it also contains a *context* attribute detailing the stack at the time the exception was thrown.

Parameters **fn** : function

Function to be called

***args** : arguments

****kwargs** : keyword arguments

Returns **result** : Future

Future containing the result of *fn(*args, **kwargs)* as its value or the exception thrown as its exception.

classmethod `Promise.`**`wait`**(*duration*)
Construct a Promise that succeeds in *duration* seconds with value *None*.

Parameters **duration** : number

Number of seconds to wait before resolving a *TimeoutError*

Returns **result** : Future

Promise that will resolve in *duration* seconds with value *None*.

classmethod `Promise.`**`eval`**(*fn*, **args*, ***kwargs*)
Call a function (synchronously) and return a Promise with its result. If an exception is thrown inside *fn*, a new exception type will be constructed inheriting both from *MiraiError* and the exception's original type. The new exception is the same the original, except that it also contains a *context* string detailing the stack at the time the exception was thrown.

Parameters **fn** : function

Function to be called

***args** : arguments

****kwargs** : keyword arguments

Returns **result** : Future

Future containing the result of *fn(*args, **kwargs)* as its value or the exception thrown as its exception.

2.3.2 Using Promises

class mirai.**Promise**(*future=None*)

A *Promise* encapsulates the result of an asynchronous computation. Think of it as a single-use mailbox – you receive a promise which will later contain a message.:

```
import requests
from mirai import Promise

def async_request(method, url, *args, **kwargs):
  "fetch a url asynchronously using `requests`"

  # construct a promise to fill later
  promise = Promise()

  def sync_request():
    "fetches synchronously & propagates exceptions"
    try:
      response = requests.request(method, url, *args, **kwargs)
      promise.setvalue(response)
    except Exception as e:
      promise.setexception(e)

  # start asynchronous computation
  Promise.call(sync_request)

  # return read-only version of promise
  return promise.future()
```

Parameters **future** : concurrent.futures.Future

Future this promise wraps.

Methods

andthen(*fn*)

Apply a function with a single argument: the value this Promise resolves to. The function must return another future. If this Promise fails, *fn* will not be called. Same as as *Promise.flatmap*.

Parameters **fn** : (value,) -> Promise

Function to apply. Takes 1 positional argument. Must return a Promise.

Returns **result** : Future

Promise *fn* will return.

ensure(*fn*)

Ensure that no-argument function *fn* is called when this Promise resolves, regardless of whether or not it completes successfuly.

Parameters **fn** : (,) -> None

function to apply upon Promise completion. takes no arguments. Return value ignored.

Returns **self** : Future

filter(*fn*)

Construct a new Promise that fails if *fn* doesn't evaluate truthily when given *self.get()* as its only argument. If *fn* evaluates falsily, then the resulting Promise fails with a *MiraiError*.

Parameters **fn** : (value,) -> bool

function used to check *self.get()*. Must return a boolean-like value.

Returns **result** : Future

Future whose contents are the contents of this Promise if *fn* evaluates truth on this Promise's contents.

flatmap(*fn*)

Apply a function with a single argument: the value this Promise resolves to. The function must return another future. If this Promise fails, *fn* will not be called.

Parameters **fn** : (value,) -> Promise

Function to apply. Takes 1 positional argument. Must return a Promise.

Returns **result** : Future

Future containing return result of *fn*.

foreach(*fn*)

Apply a function if this Promise resolves successfully. The function receives the contents of this Promise as its only argument.

Parameters **fn** : (value,) -> None

Function to apply to this Promise's contents. Return value ignored.

Returns **self** : Promise

future()

Retrieve a *Future* encapsulating this promise. A *Future* is a read-only version of the exact same thing.

Returns **future** : Future

Future encapsulating this Promise.

get(*timeout=None*)

Retrieve value of Promise; block until it's ready or *timeout* seconds have passed. If *timeout* seconds pass, then a *TimeoutError* will be raised. If this Promise failed, the set exception will be raised.

Parameters **timeout** : number or None

Number of seconds to wait until raising a *TimeoutError*. If *None*, then wait indefinitely.

Returns **result** : anything

Contents of this future if it resolved successfully.

Raises **Exception** :

Set exception if this future failed.

getorelse(*default*, *timeout=None*)

Like *Promise.get*, but instead of raising an exception when this Promise fails, returns a default value.

Parameters **default** : anything

default value to return in case of timeout or exception.

timeout : None or float

time to wait before returning default value if this promise is unresolved.

Returns result : anything

value this Promise resolves to, if it resolves successfully, else *default*.

handle(*fn*)

If this Promise fails, call *fn* on the ensuing exception to obtain a successful value.

Parameters fn : (exception,) -> anything

Function applied to recover from a failed exception. Its return value will be the value of the resulting Promise.

Returns result : Future

Resulting Future returned by applying *fn* to the exception, then setting the return value to *result*'s value. If this Promise is already successful, its value is propagated onto *result*.

isdefined()

Return *True* if this Promise has already been resolved, successfully or unsuccessfully.

Returns result : bool

isfailure()

Return *True* if this Promise failed, *False* if it succeeded, and *None* if it's not yet resolved.

Returns result : bool

issuccess()

Return *True* if this Promise succeeded, *False* if it failed, and *None* if it's not yet resolved.

Returns result : bool

join_(**others*)

Combine values of this Promise and 1 or more other Promises into a list. Results are in the same order *[self] + others* is in.

Parameters others : 1 or more Promises

Promises to combine with this Promise.

Returns result : Future

Future resolving to a list of containing the values of this Promise and all other Promises. If any Promise fails, *result* holds the exception in the one which fails soonest.

map(*fn*)

Transform this Promise by applying a function to its value. If this Promise contains an exception, *fn* is not applied.

Parameters fn : (value,) -> anything

Function to apply to this Promise's value on completion.

Returns result : Future

Future containing *fn* applied to this Promise's value. If this Promise fails, the exception is propagated.

onfailure(*fn*)

Apply a callback if this Promise fails. Callbacks can be added after this Promise has resolved. If *fn* throws an exception, a warning is printed via *logging*.

Parameters fn : (exception,) -> None

Function to call upon failure. Its only argument is the exception set to this Promise. If this future succeeds, *fn* will not be called.

Returns **self** : Promise

onsuccess(*fn*)
Apply a callback if this Promise succeeds. Callbacks can be added after this Promise has resolved. If *fn* throws an exception, a warning is printed via *logging*.

Parameters **fn** : (value,) -> None

Function to call upon success. Its only argument is the value set to this Promise. If this future fails, *fn* will not be called.

Returns **self** : Promise

or_(**others*)
Return the first Promise that finishes among this Promise and one or more other Promises.

Parameters **others** : one or more Promises

Other futures to consider.

Returns **result** : Future

First future that is resolved, successfully or otherwise.

proxyto(*other*)
Copy the state of this Promise to another.

Parameters **other** : Promise

Another Promise to copy the state of this Promise to, upon completion.

Returns **self** : Promise

Raises **MiraiError** :

if *other* isn't a Promise instance

rescue(*fn*)
If this Promise fails, call *fn* on the ensuing exception to recover another (potentially successful) Promise. Similar to *Promise.handle*, but must return a Promise (rather than a value).

Parameters **fn** : (exception,) -> Promise

Function applied to recover from a failed exception. Must return a Promise.

Returns **result** : Future

Resulting Future returned by apply *fn* to the exception this Promise contains. If this Promise is successful, its value is propagated onto *result*.

respond(*fn*)
Apply a function to this Promise when it's resolved. If *fn* raises an exception a warning will be printed via *logging*, but no action will be taken.

Parameters **fn** : (future,) -> None

Function to apply to this Promise upon completion. Return value is ignored

Returns **self** : Promise

select_(**others*)
Return the first Promise that finishes among this Promise and one or more other Promises.

Parameters **others** : one or more Promises

Other futures to consider.

Returns result : Future

First future that is resolved, successfully or otherwise.

setexception(*e*)

Set the state of this Promise as failed with a given Exception. State can only be set once; once a Promise is defined, it cannot be redefined. This operation is thread (but not process) safe.

Parameters e : Exception

Returns self : Promise

Raises AlreadyResolvedError :

if this Promise's value is already set

setvalue(*val*)

Set the state of this Promise as successful with a given value. State can only be set once; once a Promise is defined, it cannot be redefined. This operation is thread (but not process) safe.

Parameters val : value

Returns self : Promise

Raises AlreadyResolvedError :

if this Promise's value is already set

transform(*fn*)

Apply a function with a single argument (this Promise) after resolving. The function must return another future.

Parameters fn : (future,) -> Promise

Function to apply. Takes 1 positional argument. Must return a Promise.

Returns result : Future

Future containing return result of *fn*.

unit()

Convert this Promise to another that disregards its result.

Returns result : Future

Promise with a value of *None* if this Promise succeeds. If this Promise fails, the exception is propagated.

update(*other*)

Populate this Promise with the contents of another.

Parameters other : Promise

Promise to copy

Returns self : Promise

Raises MiraiError :

if *other* isn't a Promise

updateifempty(*other*)

Like *Promise.update*, but update only if this Promise isn't already defined.

Parameters other : Promise

Promise to copy, if necessary.

Returns **self** : Promise

Raises **MiraiError** :

if *other* isn't a Promise

within(*duration*)

Return a Promise whose state is guaranteed to be resolved within *duration* seconds. If this Promise completes before *duration* seconds expire, it will contain this Promise's contents. If this Promise is not resolved by then, the resulting Promise will fail with a *TimeoutError*.

Parameters **duration** : number

Number of seconds to wait before resolving a *TimeoutError*

Returns **result** : Promise

Promise guaranteed to resolve in *duration* seconds.

2.3.3 Combining Promises

classmethod Promise.**collect**(*fs*)

Convert a sequence of Promises into a Promise containing a sequence of values, one per Promise in *fs*. The resulting Promise resolves once all Promises in *fs* resolve successsfully or upon the first failure. In the latter case, the failing Promise's exception is propagated.

Parameters **fs** : [Promise]

List of Promises to merge.

Returns **result** : Future

Future containing values of all Futures in *fs*. If any Future in *fs* fails, *result* fails with the same exception.

classmethod Promise.**join**(*fs*)

Construct a Promise that resolves when all Promises in *fs* have resolved. If any Promise in *fs* fails, the error is propagated into the resulting Promise.

Parameters **fs** : [Promise]

List of Promises to merge.

Returns **result** : Future

Future containing None if all Futures in *fs* succeed, or the exception of the first failing Future in *fs*.

classmethod Promise.**select**(*fs*)

Return a Promise containing a tuple of 2 elements. The first is the first Promise in *fs* to resolve; the second is all remaining Promises that may or may not be resolved yet. The resolved Promise is not guaranteed to have completed successfully.

Parameters **fs** : [Promise]

List of Promises to merge.

Returns **result** : Future

Future containing the first Future in *fs* to finish and all remaining (potentially) unresolved Futures as a tuple of 2 elements for its value.

2.3.4 Thread Management

classmethod `Promise.`**`executor`**(*executor=None*, *wait=True*)
Set/Get the EXECUTOR Promise uses. If setting, the current executor is first shut down.

Parameters **executor** : concurrent.futures.Executor or None

If None, retrieve the current executor, otherwise, shutdown the current Executor object and replace it with this argument.

wait : bool, optional

Whether or not to block this thread until all workers are shut down cleanly.

Returns **executor** : Executor

Current executor

class `mirai.`**`UnboundedThreadPoolExecutor`**(*max_workers=None*)
A thread pool with an infinite number of threads.

This interface conforms to the typical *concurrent.futures.Executor* interface, but doesn't limit the user to a finite number of threads. In normal situations, this is undesirable – too many threads, and your program will spend more time switching contexts than actually working!

On the other hand, if you patch the *thread* module with *gevent*, spawning tens of thousands of threads is totally OK. This is where this executor comes in.

Parameters **max_workers: None or int, optional** :

Number of worker threads. If None, a new thread is created every time a new task is submitted. If an integer, this executor acts exactly like a normal *concurrent.futures.ThreadPoolExecutor*.

Methods

`shutdown`(*wait=True*)
Shutdown this thread pool, preventing future tasks from being enqueued.

Parameters **wait** : bool

Wait for all running threads to finish. Only used if this pool was initialized with a fixed number of workers.

`submit`(*fn*, **args*, ***kwargs*)
Submit a new task to be executed asynchronously.

If *self.max_workers* is an integer, then the behavior of this function will be identical to that of *concurrent.futures.ThreadPoolExecutor*. However, if it is None, then a new daemonized thread will be constructed and started.

Parameters **fn** : function-like

function (or callable object) to execute asynchronously.

args : list

positional arguments to pass to *fn*.

kwargs: dict :

keyword arguments to pass to *fn*.

Returns **future** : concurrent.futures.Future

Container for future result or exception

2.3.5 Exceptions

exception `mirai.`**`MiraiError`**
Base class for all exceptions raise by Promises

exception `mirai.`**`AlreadyResolvedError`**
Exception thrown when attempting to set the value or exception of a Promise that has already had its value or exception set.

exception `mirai.`**`TimeoutError`**
The operation exceeded the given deadline.

2.4 Caveats

While `mirai` tries to make multithreading as painless as possible, there are a few small cases to be mindful of.

2.4.1 You only have so many threads...

While `mirai` does its best to hide thread management from you, the fact remains that there are a finite number of worker threads (default: 10). If all of those worker threads are indefinitely busy on never-ending tasks, then all tasks queued after that won't execute!. For example,

```
from concurrent.futures import ThreadPoolExecutor
from mirai import Promise
import time

def forever():
  while True:
    time.sleep(1)

def work():
  return "I'll never run!"

# only 5 workers available
Promise.executor(ThreadPoolExecutor(max_workers=5))

# these threads take up all the executor's workers
traffic_jam = [Promise.call(forever) for i in range(5)]

# this will block forever, as all the workers are busy
real_work = Promise.call(work).get()
```

2.4.2 Waiting on other Promises

Under the hood, mirai executes all tasks registered with `Promise.call()` via a `ThreadPoolExecutor` with a finite number of threads (this can be access with `Promise.executor()`). This is to ensure that there are never too many threads active at once.

The one cardinal sin of `mirai` is waiting upon a promise with `Promise.get()` within a currently-running promise. The reason is that the *waiting* thread has reserved one of `mirai`'s finite number of worker threads, and if all such

worker threads are waiting upon *other* promises, then there will be no workers for *awaited upon* promises. In other words, all worker threads will wait forever. For example,

```
from concurrent.futures import ThreadPoolExecutor
from mirai import Promise
import time

def fanout(n):
  secondaries = [Promise.call(time.sleep, 0.1 * i) for i in range(n)]
  return Promise.collect(secondaries).get()

# only 5 workers available
Promise.executor(ThreadPoolExecutor(max_workers=5))

# start 5 "primary" threads. Each of these will wait on 5 "secondary" threads,
# but due to the maximum worker limit, those secondary threads will never get
# a chance to run. The primary threads are already taking up all the workers!
primaries = [Promise.call(fanout, 5) for i in range(5)]

# this will never return...
Promise.collect(primaries).get()
```

The workaround for this is to use `mirai.UnboundedThreadPoolExecutor`, which doesn't have an upper bound on the number of active threads.

2.4.3 Zombie threads

Standard behavior on multithreaded applications is to allow every thread to exit cleanly unless killed explicitly. For `mirai`, this means that even though all the threads *you care about* may be finished, there may still be other threads running, and thus your process will not end, even if you use `sys.exit()`.

If a thread is in an infinite loop for example, your code will never exit cleanly. The recourse for this is to use `mirai.UnboundedThreadPoolExecutor` as your executor with *max_workers* set to None. Unlike `ThreadPoolExecutor`, this executor will not wait for threads to finish cleanly when the process exits.

m

m