Ruby trunk - Feature #15527

Redesign of timezone object requirements

01/12/2019 09:02 AM - zverok (Victor Shepelev)

Status:	Open	
Priority:	Normal	
Assignee:		
Target version:		

Description

In #14850, there was timezone support introduced, there were pretty specific requirements for the Timezone object:

> A timezone argument must have local_to_utc and utc_to_local methods... The local_to_utc method should convert a Time-like object from the timezone to UTC, and utc_to_local is the opposite. ... The zone of the result is just ignored.

I understand this requirements were modelled after existing TZInfo gem, but the problem with them are:

- they are too ad-hoc (in fact, return values of methods aren't used as a "Time object", but as a tuple of time components)
- they belong to outdated tzinfo API (ignoring of offsets is due to support of **Ruby 1.8**, which didn't allowed constructing Time object with arbitrary offset, see discussion), recent release introduces also #to_local, which returns Time with proper offset.

The latter is a bit of time paradox: Ruby **2.6** new feature is designed after the library which works this way to support Ruby **1.8** :)
The bad thing is, this approach somehow "codifies" outdated API (so in future, any alternative timezone library should support pretty arbitrary API).

I believe, that in order to do everything that Time needs, timezone object should be able to answer exactly one question: "what offset from UTC is/was observed in this timezone at particular date". In fact, TZInfo **has** the API for this:

```
tz = TZInfo::Timezone.get('America/New_York')
# => #<TZInfo::DataTimezone: America/New_York>
tz.utc_offset(Time.now)
# => -18000
```

If I understand correctly, this requirement ("A timezone argument must have #utc_offset(at_time)") will greatly simplify the implementation of Time, while also being compatible with TZInfo gem and much more explainable. With this requirement, alternative implementations could now be much simpler and focus only on "find the proper timezone/period/offset", omitting any (hard) details of deconstructing/constructing Time objects.

History

#1 - 01/13/2019 09:53 AM - naruse (Yui NARUSE)

Sounds interesting, but zone.utc_offset(time) can only be a partial alternative of utc_to_local with using `gmtime(3).

Note that a timezone system requires two API.
One is an API which converts from [year, month, day, hour, minute, second] to epoch.
And another is an API which converts from epoch to [year, month, day, hour, minute, second, isdst, zonestr].

#2 - 01/14/2019 06:40 PM - zverok (Victor Shepelev)

> Note that a timezone system requires two API.

Sorry for my arrogance, but can you please explain this a bit?..
From what I can understand from code, local_to_utc is used only from Time.new, but I am not quite sure why exactly.
From "logical" point of view, we need just to understand what exact UTC offset it should have (so it covered with Zone#utc_offset), the only uncertainty here is *what exact moment* we want UTC offset for -- this can fire in the foot at DST change moment, but I am not sure how a pair of APIs could help here.

#3 - 02/01/2019 09:41 AM - nobu (Nobuyoshi Nakada)

zone.utc_offset(time) seems to consider the timezone of the argument, or the UTC offset.
When constructing a time from each components (year, month, day, hour...), we don't know the offset.
That means we have to know the offset to get it by utc_offset method.

It's the key in the locked box, isn't it?

#4 - 02/01/2019 06:06 PM - zverok (Victor Shepelev)

That's interesting problem indeed.
I'll look at it on particular example: my own timezone :)

We have GMT+2 at winter and GMT+3 at summer, transitions for 2018 were Mar 25 03:00 and Oct 28 04:00.

So....

For mid-period, everything is obvious, checking at UTC is enough:

```
tz = TZInfo::Timezone.get('Europe/Kiev')

tz.utc_offset(Time.utc(2018, 1, 1))
# => 7200
tz.utc_offset(Time.utc(2018, 6, 1))
# => 10800
```

But how do we decide for the 2018-10-28 4:00 for example? I believe it could be done this way (with the only method #utc_offset(at) required):

```
tz.utc_offset(Time.utc(2018, 10, 28, 2, 0)) # step 1: take "approximate" offset from UTC time with given value
# => 7200
tz.utc_offset(Time.new(2018, 10, 28, 2, 0, 0, 7200))
# step 2: take real offset from time with approximate offset
# => 10800
tz.utc_offset(Time.new(2018, 10, 28, 2, 0, 0, 10800)) # step 3: check it: yes, it is right
# => 10800
```

So, the real answer is: Time at 2018-10-28 02:00 Europe/Kiev has UTC offset GMT+3.

Same for transition in another direction:

```
tz.utc_offset(Time.new(2018, 3, 25, 3, 0, 0)) # "approximate" offset
# => 10800
tz.utc_offset(Time.new(2018, 3, 25, 3, 0, 0, 10800)) # real offset?
# => 7200
tz.utc_offset(Time.new(2018, 3, 25, 3, 0, 0, 7200)) # check it: no, it is transition point
# => 10800
```

The real answer is: it is lost hour (we have 04:00 after 02:59 at transition point), 2018-10-25 03:00 Europe/Kiev can't exist, should be an exception.

Yes, 3 calls to utc_offset are kinda indirect, but the current implementation is also "indirect" in a sense it requires timezone library to calculate Time object but doesn't use it.

What is worse is: if modern (2.6-aware) timezone library will try to make proper Time object (using Time.new with its timezone object), there could be infinite recursion (because Time itself and timezone library would call each other). That's because current requirements were designed with exactly one implementation in mind -- which is a third-party library with a legacy interface.

In fact, it is funny paradox that exactly this "legacy" feature (library is able to work with non-offsetted time, considering it as just "tuple of time values"). Maybe more robust API to require would be something like:

```
tz.utc_offset_by_tuple(2018, 3, 25, ...) # => consider it as a components of local time, return seconds offset
```

PS: In Ruby, currently, creating time on a border of transition is impossible

```
Time.new(2018, 10, 28, 3, 00, 0, tz)
# TZInfo::AmbiguousTime (2018-10-28 03:00:00 is an ambiguous local time.)
```

TZInfo itself solves it this way:

```
tz.local_time(2018, 10, 28, 3, 0, 0, 0)
# TZInfo::AmbiguousTime (2018-10-28 03:00:00 is an ambiguous local time.)
tz.local_time(2018, 10, 28, 3, 00, 0, 0, true) # last param is dst=true
# => 2018-10-28 03:00:00 +0300
tz.local_time(2018, 10, 28, 3, 00, 0, 0, false) # dst = false
# => 2018-10-28 03:00:00 +0200
```

#5 - 02/02/2019 11:15 AM - nobu (Nobuyoshi Nakada)

zverok (Victor Shepelev) wrote:

> Yes, 3 calls to utc_offset are kinda indirect, but the current implementation is also "indirect" in a sense it requires timezone library to calculate Time object but doesn't use it.

It is trivial, and is possible to change.
3 calls to utc_offset seems slower than 1 call to local_to_utc.

> What is worse is: if modern (2.6-aware) timezone library will try to make proper Time object (using Time.new with its timezone object), there could be infinite recursion (because Time itself and timezone library would call each other).

You mean the case a timezone library calls Time.new in utc_to_local method?
Time.new with a timezone object would call local_to_utc, with a UTC time-like object, and the result should be UTC.

> That's because current requirements were designed with exactly one implementation in mind -- which is a third-party library with a legacy interface.

It was designed with another implementation, timezone gem, too.

> In fact, it is funny paradox that exactly this "legacy" feature (library is able to work with non-offsetted time, considering it as just "tuple of time values"). Maybe more robust API to require would be something like:
>
> ```
> tz.utc_offset_by_tuple(2018, 3, 25, ...)
> # => consider it as a components of local time, return seconds offset
> ```

Non-offsetted Time-like object is used now.

> TZInfo itself solves it this way:
>
> ```
> tz.local_time(2018, 10, 28, 3, 0, 0, 0)
> # TZInfo::AmbiguousTime (2018-10-28 03:00:00 is an ambiguous local time.)
> tz.local_time(2018, 10, 28, 3, 00, 0, 0, true) # last param is dst=true
> # => 2018-10-28 03:00:00 +0300
> tz.local_time(2018, 10, 28, 3, 00, 0, 0, false) # dst = false
> # => 2018-10-28 03:00:00 +0200
> ```

Yes I know, but another implementation in my mind, timezone, doesn't support dst argument.