NAME

Glib::Object::Introspection − Dynamically create Perl language bindings

SYNOPSIS

```
use Glib::Object::Introspection;
Glib::Object::Introspection->setup(
  basename => 'Gtk',
  version => '3.0',
  package => 'Gtk3');
# now GtkWindow, to mention just one example, is available as
# Gtk3::Window, and you can call gtk_window_new as Gtk3::Window->new
```

ABSTRACT

Glib::Object::Introspection uses the gobject-introspection and libffi projects to dynamically create Perl bindings for a wide variety of libraries. Examples include gtk+, webkit, libsoup and many more.

DESCRIPTION FOR LIBRARY USERS

To allow Glib::Object::Introspection to create bindings for a library, the library must have installed a typelib file, for example `$prefix/lib/girepository-1.0/Gtk-3.0.typelib`. In your code you then simply call `Glib::Object::Introspection->setup` with the following key-value pairs to set everything up:

basename => `$basename`

The basename of the library that should be wrapped. If your typelib is called `Gtk-3.0.typelib`, then the basename is 'Gtk'.

version => `$version`

The particular version of the library that should be wrapped, in string form. For `Gtk-3.0.typelib`, it is '3.0'.

package => `$package`

The name of the Perl package where every class and method of the library should be rooted. If a library with basename 'Gtk' contains an class 'GtkWindow', and you pick as the package 'Gtk3', then that class will be available as 'Gtk3::Window'.

The Perl wrappers created by `Glib::Object::Introspection` follow the conventions of the Glib module and old hand-written bindings like Gtk2. You can use the included tool `perli11ndoc` to view the documentation of all installed libraries organized and displayed in accordance with these conventions. The guiding principles underlying the conventions are described in the following.

Namespaces and Objects

The namespaces of the C libraries are mapped to Perl packages according to the `package` option specified, for example:

```
gtk_ => Gtk3
gdk_ => Gtk3::Gdk
gdk_pixbuf_ => Gtk3::Gdk::Pixbuf
pango_ => Pango
```

Classes, interfaces and boxed and fundamental types get their own namespaces, in a way, as the concept of the GType is completely replaced in the Perl bindings by the Perl package name.

```
GtkButton => Gtk3::Button
GdkPixbuf => Gtk3::Gdk::Pixbuf
GtkScrolledWindow => Gtk3::ScrolledWindow
PangoFontDescription => Pango::FontDescription
```

With this package mapping and Perl's built-in method lookup, the bindings can do object casting for you. This gives us a rather comfortably object-oriented syntax, using normal Perl object semantics:

```
in C:
  GtkWidget * b;
  b = gtk_check_button_new_with_mnemonic ("_Something");
  gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (b), TRUE);
  gtk_widget_show (b);

in Perl:
  my $b = Gtk3::CheckButton->new_with_mnemonic ('_Something');
  $b->set_active (1);
  $b->show;
```

You see from this that cast macros are not necessary and that you don't need to type namespace prefixes quite so often, so your code is a lot shorter.

Flags and Enums

Flags and enum values are handled as strings, because it's much more readable than numbers, and because it's automagical thanks to the GType system. Values are referred to by their nicknames; basically, strip the common prefix, lower-case it, and optionally convert '_' to '-':

```
GTK_WINDOW_TOPLEVEL => 'toplevel'
GTK_BUTTONS_OK_CANCEL => 'ok-cancel' (or 'ok_cancel')
```

Flags are a special case. You can't (sensibly) bitwise-or these string-constants, so you provide a reference to an array of them instead. Anonymous arrays are useful here, and an empty anonymous array is a simple way to say 'no flags'.

```
FOO_BAR_BAZ | FOO_BAR_QUU | FOO_BAR_QUUX => [qw/baz quu qux/]
0 => []
```

In some cases you need to see if a bit is set in a bitfield; methods returning flags therefore return an overloaded object. See Glib for more details on which operations are allowed on these flag objects, but here is a quick example:

```
in C:
  /* event->state is a bitfield */
  if (event->state & GDK_CONTROL_MASK) g_printerr ("control was down\n");

in Perl:
  # $event->state is a special object
  warn "control was down\n" if $event->state & "control-mask";
```

But this also works:

```
warn "control was down\n" if $event->state * "control-mask";
warn "control was down\n" if $event->state >= "control-mask";
warn "control and shift were down\n"
                          if $event->state >= ["control-mask", "shift-mask"];
```

Memory Handling

The functions for ref'ing and unref'ing objects and free'ing boxed structures are not even mapped to Perl, because it's all handled automagically by the bindings. Objects will be kept alive so long as you have a Perl scalar pointing to it or the object is referenced in another way, e.g. from a container.

The only thing you have to be careful about is the lifespan of non reference counted structures, which means most things derived from `Glib::Boxed`. If it comes from a signal callback it might be good only until you return, or if it's the insides of another object then it might be good only while that object lives. If in doubt you can `copy`. Structs from`copy` or `new` are yours and li ve as long as referred to from Perl.

Callbacks

Use normal Perl callback/closure tricks with callbacks. The most common use you'll have for callbacks is with the Glib `signal_connect` method:

```
$widget->signal_connect (event => \&event_handler, $user_data);
$button->signal_connect (clicked => sub { warn "hi!\n" });
```

`$user_data` is optional, and with Perl closures you don't often need it (see ''Persistent variables with closures'' in perlsub).

The userdata is held in a scalar, initialized from what you give in `signal_connect` etc. It's passed to the callback in usual Perl ''call by reference'' style which means the callback can modify its last argument, ie. `$_[-1]`, to modify the held userdata. This is a little subtle, but you can use it for some ''state'' associated with the connection.

```
$widget->signal_connect (activate => \&my_func, 1);
sub my_func {
  print "activation count: $_[-1]\n";
  $_[-1] ++;
}
```

Because the held userdata is a new scalar there's no change to the variable (etc.) you originally passed to `signal_connect`.

If you have a parent object in the userdata (or closure) you have to be careful about circular references preventing parent and child being destroyed. See ''Two-Phased Garbage Collection'' in perlobj about this generally. Toplevel widgets like `Gtk3::Window` always need an explicit `$widget->destroy` so their `destroy` signal is a good place to break circular references. But for other widgets it's usually friendliest to avoid circularities in the first place, either by using weak references in the userdata, or possibly locating a parent dynamically with `$widget->get_ancestor`.

Exception handling

Anything that uses GError in C will `croak` on failure, setting $@ to a magical exception object, which is overloaded to print as the returned error message. The ideology here is that GError is to be used for runtime exceptions, and `croak` is how you do that in Perl. You can catch a croak very easily by wrapping the function in an eval:

```
eval {
  my $pixbuf = Gtk3::Gdk::Pixbuf->new_from_file ($filename);
  $image->set_from_pixbuf ($pixbuf);
};
if ($@) {
  print "$@\n"; # prints the possibly-localized error message
  if (Glib::Error::matches ($@, 'Gtk3::Gdk::Pixbuf::Error',
                                'unknown-format')) {
    change_format_and_try_again ();
  } elsif (Glib::Error::matches ($@, 'Glib::File::Error', 'noent')) {
    change_source_dir_and_try_again ();
  } else {
    # don't know how to handle this
    die $@;
  }
}
```

This has the added advantage of letting you bunch things together as you would with a try/throw/catch block in C++ — you get cleaner code. By using Glib::Error exception objects, you don't have to rely on string matching on a possibly localized error message; you can match errors by explicit and predictable conditions. See Glib::Error for more information.

Output arguments, lists, hashes

In C you can only return one value from a function, and it is a common practice to modify pointers passed in to simulate returning multiple values. In Perl, you can return lists; any functions which modify arguments are changed to return them instead.

Arguments and return values that have the types GList or GSList or which are C arrays of values will be converted to and from references to normal Perl arrays. The same holds for GHashTable and references to normal Perl hashes.

Object class functions

Object class functions like `Gtk3::WidgetClass::find_style_propery` can be called either with a package name or with an instance of the package. For example:

```
Gtk3::WidgetClass::find_style_property ('Gtk3::Button', 'image-spacing')

my $button = Gtk3::Button->new;
Gtk3::WidgetClass::find_style_property ($button, 'image-spacing')
```

Overriding virtual functions

When subclassing a gtk+ class or when implementing a gtk+ interface with Glib::Object::Subclass, you can override any virtual functions that the class has by simply defining sub routines with names obtained by capitalizing the original names of the virtual functions. So, for example, if you implement a custom subclass of `Gtk3::CellRenderer` and want to override its virtual function `render`, you provide a sub routine with the name `RENDER` in your package.

```
sub RENDER {
  my ($cell, $cr, $widget, $background_area, $cell_area, $flags) = @_;
  # do something
}
```

DESCRIPTION FOR LIBRARY BINDING AUTHORS

`Glib::Object::Introspection->setup`

`Glib::Object::Introspection->setup` takes a few optional arguments that augment the generated API:

search_path => `$search_path`

A path that should be used when looking for typelibs. If you use typelibs from system directories, or if your environment contains a properly set `GI_TYPELIB_PATH` variable, then this should not be necessary.

name_corrections => { auto_name => new_name, ... }

A hash ref that is used to rename functions and methods. Use this if you don't like the automatically generated mapping for a function or method. For example, if `g_file_hash` is automatically represented as `Glib::IO::file_hash` but you want `Glib::IO::File::hash` then pass

```
name_corrections => {
  'Glib::IO::file_hash' => 'Glib::IO::File::hash'
}
```

class_static_methods => [function1, ...]

An array ref of function names that you want to be treated as class-static methods. That is, if you want be able to call `Gtk3::Window::list_toplevels` as `Gtk3::Window->list_toplevels`, then pass

```
class_static_methods => [
  'Gtk3::Window::list_toplevels'
]
```

The function names refer to those after name corrections.

flatten_array_ref_return_for => [function1, ...]

An array ref of function names that return an array ref that you want to be flattened so that they return plain lists. For example

```
flatten_array_ref_return_for => [
  'Gtk3::Window::list_toplevels'
]
```

The function names refer to those after name corrections. Functions occurring in `flatten_array_ref_return_for` may also occur in `class_static_methods`.

handle_sentinel_boolean_for => [function1, ...]

An array ref of function names that return multiple values, the first of which is to be interpreted as indicating whether the rest of the returned values are valid. This frequently occurs with functions that have out arguments; the boolean then indicates whether the out arguments have been written. With `handle_sentinel_boolean_for`, the first return value is taken to be the sentinel boolean. If it is true, the rest of the original return values will be returned, and otherwise an empty list will be returned.

```
handle_sentinel_boolean_for => [
  'Gtk3::TreeSelection::get_selected'
]
```

The function names refer to those after name corrections. Functions occurring in `handle_sentinel_boolean_for` may also occur in `class_static_methods`.

use_generic_signal_marshaller_for => [[package1, signal1, [arg_converter1]], ...]

Use an introspection-based generic signal marshaller for the signal `signal1` of type `package1`. If given, use the code reference `arg_converter1` to convert the arguments that are passed to the signal handler. In contrast to Glib's normal signal marshaller, the generic signal marshaller supports, among other things, pointer arrays and out arguments.

reblessers => { package => \&reblesser, ... }

Tells G:O:I to invoke *reblesser* whenever a Perl object is created for an object of type *package*. Currently, this only applies to boxed unions. The reblesser gets passed the pre-created Perl object and needs to return the modified Perl object. For example:

```
sub Gtk3::Gdk::Event::_rebless {
  my ($event) = @_;
  return bless $event, lookup_real_package_for ($event);
}
```

`Glib::Object::Introspection->invoke`

To invoke specific functions manually, you can use the low-level `Glib::Object::Introspection->invoke`.

```
Glib::Object::Introspection->invoke(
  $basename, $namespace, $function, @args)
```

- `$basename` is the basename of a library, like 'Gtk'.
- `$namespace` refers to a namespace inside that library, like 'Window'. Use undef here if you want to call a library-global function.
- `$function` is the name of the function you want to invoke. It can also refer to the name of a constant.
- `@args` are the arguments that should be passed to the function. For a method, this should include the invocant. For a constructor, this should include the package name.

`Glib::Object::Introspection->invoke` returns whatever the function being invoked returns.

Overrides

To override the behavior of a specific function or method, create an appropriately named sub in the correct package and have it call `Glib::Object::Introspection->invoke`. Say you want to override `Gtk3::Window::list_toplevels`, then do this:

```
sub Gtk3::Window::list_toplevels {
  # ...do something...
  my $ref = Glib::Object::Introspection->invoke (
              'Gtk', 'Window', 'list_toplevels',
              @_);
  # ...do something...
  return wantarray ? @$ref : $ref->[$#$ref];
}
```

The sub's name and package must be those after name corrections.

Converting a Perl variable to a GValue

If you need to marshal into a GValue, then Glib::Object::Introspection cannot do this automatically because the type information is missing. If you do have this information in your module, however, you can use Glib::Object::Introspection::GValueWrapper to do the conversion. In the wrapper for a function that expects a GValue, do this:

```
...
my $type = ...; # somehow get the package name that
                # corresponds to the correct GType
my $wrapper =
  Glib::Object::Introspection::GValueWrapper->new ($type, $value);
# now use Glib::Object::Introspection->invoke and
# substitute $wrapper where you'd use $value
...
```

If you need to call a function that expects an already set-up GValue and modifies it, use `get_value` on the wrapper afterwards to obtain the value. For example:

```
my $wrapper =
  Glib::Object::Introspection::GValueWrapper->new ('Glib::Boolean', 0);
$box->child_get_property ($label, 'expand', $gvalue);
my $value = $gvalue->get_value
```

Handling raw enumerations and flags

If you need to handle raw enumerations/flags or extendable enumerations for which more than the predefined values might be valid, then use `Glib::Object::Introspection->convert_enum_to_sv`, `Glib::Object::Introspection->convert_sv_to_enum`, `Glib::Object::Introspection->convert_flags_to_sv` and `Glib::Object::Introspection->convert_sv_to_flags`. They will raise an exception on unknown values; catching it then allows you to implement fallback behavior.

```
Glib::Object::Introspection->convert_enum_to_sv (package, enum_value)
Glib::Object::Introspection->convert_sv_to_enum (package, sv)

Glib::Object::Introspection->convert_flags_to_sv (package, flags_value)
Glib::Object::Introspection->convert_sv_to_flags (package, sv)
```

SEE ALSO

perl-Glib: Glib
gobject-introspection: <http://live.gnome.org/GObjectIntrospection>
libffi: <http://sourceware.org/libffi/>

AUTHORS

Emmanuele Bassi <ebassi at linux intel com>
muppet <scott asofyet org>
Torsten Schönfeld <kaffeetisch at gmx de>

LICENSE