Application Notes for PHP

Previous Next

On this page:

PHP Request Semantics

In PHP terminology, a *request* is the execution of a PHP script on behalf of a Web client. Each request essentially runs in its own instance of the PHP interpreter, isolated from any other requests that may be executing concurrently. Upon the completion of a request, the interpreter reclaims memory and other resources that were acquired during the request, including objects created by the Ice extension.

Using Communicators in PHP

A communicator represents an instance of the Ice run time. A PHP script that needs to invoke an operation on a remote Ice object must initialize a communicator, obtain and narrow a proxy, and make the invocation. For example, here is a minimal (but complete) Ice script:

PHP

```
<?php
require_once 'Ice.php';
require_once 'Hello.php';

$communicator = null;

try
{
    $data = new Ice\InitializationData;
    $data->properties = Ice\createProperties();
    $data->properties->load("props.cfg");
    $communicator = Ice\initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo\HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice\LocalException $ex)
{
    // Deal with exception...
}

if($communicator)
{
    try
    {
        $communicator->destroy();
    }
    catch(Ice\LocalException $ex)
    {
        // Ignore.
    }
}
?>
```

By default, the Ice extension automatically destroys any communicator that was created during a request. This means a script can usually omit the call to d`estroy` unless there is an application-specific reason to destroy the communicator explicitly. Consequently, we can simplify our script to the following:

PHP

```
<?php
require 'Ice.php';
require 'Hello.php';

try
{
    $data = new Ice\InitializationData;
    $data->properties = Ice\createProperties();
    $data->properties->load("props.cfg");
    $communicator = Ice\initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo\HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice\LocalException $ex)
{
    // Deal with exception...
}
?>
```

Now we allow the Ice extension to destroy our communicator automatically. (The extension traps and ignores any exception raised by `destroy`.)

Although the automatic destruction of communicators is convenient, it is important to consider the performance characteristics of this script. Specifically, each execution of the script involves the following activities:

1. Create an Ice property set

2. Load and parse a property file
3. Initialize a communicator with the given configuration properties
4. Obtain a proxy for the remote Ice object
5. Establish a socket connection to the server
6. Send a request message and wait for the reply
7. Destroy the communicator, which closes the socket connection

Of primary concern are the activities that involve system calls, such as opening and reading files, creating and using sockets, and so on. The overhead incurred by these calls may not matter if the script is only executed infrequently, but for an application with high request rates it is necessary to minimize this overhead:

- A pre-configured property set eliminates the need to parse a property file in each request.
- Timeouts prevent a script from blocking indefinitely in case Ice encounters delays while performing socket operations.
- Registering a communicator avoids the need to create and destroy a communicator in every request.
- Be aware of the number of "round trips" (request-reply pairs) your script makes. For example, the script above uses `checkedCast` to verify that the remote Ice object supports the desired Slice interface. However, calling `checkedCast` causes the Ice run time to send a request to the server and await its reply, therefore this script is actually making two remote invocations. It is unnecessary to perform a checked cast if it is safe for the client to assume that the Ice object supports the correct interface, in which case using an `uncheckedCast` instead avoids the extra round trip.

Back to Top ^

Managing Property Sets in PHP

A PHP application can manually construct a property set for configuring its communicator. The Ice extension also provides a PHP-specific property set API that helps to minimize the overhead associated with initializing a communicator, allowing you to configure a default property set along with an unlimited number of named property sets (or *profiles*). You can populate a property set using a configuration file, command-line options, or both. Property sets are initialized using the normal Ice semantics: command-line options override any settings from a configuration file.

The Ice extension creates these property sets during web server startup, which means any subsequent changes you might make to the configuration have no effect until the web server is restarted. Also keep in mind that specifying a relative path name for a configuration file usually means the path name is evaluated relative to the web server's working directory.

Back to Top ^

Default Property Set in PHP

The INI directives `ice.config` and `ice.options` specify the configuration file and the command-line options for the default property set, respectively. These directives must appear in PHP's configuration file, which is usually named `php.ini`:

```
; Snippet from php.ini on Linux
extension=IcePHP.so
ice.config=/opt/MyApp/default.cfg
ice.options="--Ice.Override.Timeout=2000"
```

Back to Top ^

Profiles in PHP

Profiles are useful when several unrelated applications execute in the same web server, or when a script needs to choose among multiple configurations. To configure your profiles, add an `ice.profiles` directive to PHP's configuration file. The value of this directive is a file containing profile definitions:

```
; Snippet from php.ini on Linux
ice.profiles=/opt/MyApp/profiles
```

The profile definition file uses INI syntax:

```
[Production]
config=/opt/MyApp/prod.cfg
options="..."

[Debug]
config=/opt/MyApp/debug.cfg
options="--Ice.Trace.Network=3 ..."
```

The name of each profile is enclosed in square brackets. The configuration file and command-line options for each profile are defined using the `config` and `options` entries, respectively.

Back to Top ^

Using Property Sets in PHP

The `Ice\getProperties` function allows a script to obtain a copy of a property set. When called without an argument, or with an empty string, the function returns the default property set. Otherwise, the function expects the name of a configured profile and returns the property set associated with that profile. The return value is an instance of `Ice\Properties`, or `null` if no matching profile was found.

Note that the Ice extension always creates the default property set, which is empty if the `ice.config` and `ice.options` directives are not defined. Also note that changes a script might make to a property set returned by this function have no effect on other requests because the script is modifying a *copy* of the original property set.

Now we can modify our script to use `Ice\getProperties` and avoid the need to load a configuration file in each request:

PHP

```
<?php
require_once 'Ice.php';
require_once 'Hello.php';

try
{
    $data = new Ice\InitializationData;
    $data->properties = Ice\getProperties();
    $communicator = Ice\initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo\HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice\LocalException $ex)
{
    // Deal with exception...
}
?>
```

Back to Top ^

Security Considerations for Property Sets in PHP

Ice configuration properties may contain sensitive information such as the path name of the private key for an X.509 certificate. If multiple untrusted PHP applications run in the same web server, avoid the use of the default property set and choose sufficiently unique names for your named profiles. The Ice extension does not provide a means for enumerating the names of the configured profiles, therefore a malicious script would have to guess the name of a profile in order to examine its configuration properties.

To prevent a script from using the value of `ice.profiles` to open the profile definition file directly, enable the `ice.hide_profiles` directive to cause the Ice extension to replace the `ice.profiles` setting after it has processed the file. The `ice.hide_profiles` directive is enabled by default.

Back to Top ^

Timeouts in PHP

All twoway remote invocations made by a PHP script have synchronous semantics: the script does not regain control until Ice receives a reply from the server. As a result, we recommend configuring a suitable timeout value for all of your proxies as a defensive measure against network delays.

Back to Top ^

Registered Communicators in PHP

You can register a communicator to prevent it from being destroyed at the completion of a script. For example, a session-based PHP application can create a communicator for each new session and register it for reuse in subsequent requests of the same session. Reusing a communicator in this way avoids the overhead associated with creating and destroying a communicator in each request. Furthermore, it allows socket connections established by the Ice run time to remain open and available for use in another request.

Back to Top ^

Limitations of Registered Communicators in PHP

A communicator object is local to the process that created it, which in the case of PHP is usually a web server process. The usefulness of a registered communicator is therefore limited to situations in which an application can ensure that subsequent page requests are handled by the same web server process as the one that originally created the registered communicator. For example, registered communicators would not be appropriate in a typical CGI configuration because the CGI process terminates at the end of each request. A simple (but often impractical) solution is to configure your web server to use a single persistent process. The topic of configuring a web server to take advantage of registered communicators is outside the scope of this manual.

Back to Top ^

Using Registered Communicators in PHP

The API for registered communicators consists of three functions:

- `Ice\register($communicator, $name, $expires=0)`
  Registers a communicator with the given name. On success, the function returns true. If another communicator is already registered with the same name, the function returns false. The `expires` argument specifies a timeout value in minutes; if `expires` is greater than zero, the Ice extension automatically destroys the communicator if it has not been retrieved (via `Ice\find`) for the specified number of minutes. The default value (zero) means the communicator never expires, in which case the Ice extension only destroys the communicator when the current process terminates.
  It is legal to register a communicator with more than one name. In that case, the most recent value of expires takes precedence.

- `Ice\unregister($name)`
  Removes the registration for a communicator with the given name. Returns true if a match was found or false otherwise. Calling `Ice\unregister` does not cause the communicator to be destroyed; rather, the communicator is destroyed as soon as all pending requests that are currently using the communicator have completed. Destroying a registered communicator explicitly also removes its registration.

- `Ice\find($name)`
  Retrieves the communicator associated with the given name. Returns `null` if no match is found.

An application typically uses registered communicators as follows:

PHP

```
<?php
require_once 'Ice.php';

$communicator = Ice\find('MyCommunicator');
$expires = ...;
if($communicator == null)
{
    $communicator = Ice\initialize(...);
    Ice\register($communicator, 'MyCommunicator', $expires);
}

...
?>
```

Note that communicators consume resources such as threads, sockets, and memory, therefore an application should be designed to minimize the number of communicators it registers. Using a suitable expiration timeout prevents registered communicators from accumulating indefinitely.

A simple application that demonstrates the use of registered communicators can be found in the `Glacier2/hello` subdirectory of the PHP sample programs.

Back to Top ^

Security Considerations for Registered Communicators in PHP

There are risks associated with allowing untrusted applications to gain access to a registered communicator. For example, if a malicious script obtains a registered communicator that is configured with SSL credentials, the script could potentially make secure invocations as if it were the trusted script.

Registering a communicator with a sufficiently unique name reduces the chance that a malicious script could guess the communicator's name. For applications that make use of PHP's session facility, the session ID is a reasonable choice for a communicator name. The sample application in `Glacier2/hello` demonstrates this solution.

Back to Top ^

Lifetime of Object Factories in PHP

PHP reclaims all memory at the end of each request, which means any object factories that a script might have installed in a registered communicator are destroyed when the request completes even if the communicator is not destroyed. As a result, a script must install its object factories in a registered communicator for *every* request, as shown in the example below:

PHP

```
<?php
require 'Ice.php';

$communicator = Ice\find('MyCommunicator');
$expires = ...;
if($communicator == null)
{
    $communicator = Ice_initialize(...);
    Ice\register($communicator, 'MyCommunicator', $expires);
}

$communicator->addObjectFactory(new MyFactory, MyClass::ice_staticId());
...
?>
```

The Ice extension invokes the `destroy` method of each factory prior to the completion of a request.

Back to Top ^

See Also

- Properties and Configuration
- Connection Timeouts

Previous Next