Thread Safety

Previous Next

The Ice run time itself is fully thread safe, meaning multiple application threads can safely call methods on objects such as communicators, object adapters, and proxies without synchronization problems. As a developer, you must also be concerned with thread safety because the Ice run time can dispatch multiple invocations concurrently in a server. In fact, it is possible for multiple requests to proceed in parallel within the same servant and within the same operation on that servant. It follows that, if the operation implementation manipulates non-stack storage (such as member variables of the servant or global or static data), you must interlock access to this data to avoid data corruption.

The need for thread safety in an application depends on its configuration. Using the default thread pool configuration typically makes synchronization unnecessary because at most one operation can be dispatched at a time. Thread safety becomes an issue once you increase the maximum size of a thread pool.

Ice uses the native synchronization and threading primitives of each platform. For C++ users, Ice provides a collection of convenient and portable wrapper classes for use by Ice applications.

On this page:

Threading Issues with Marshaling

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference:

- Ice is marshaling this data, outside any synchronization under the control of the application
- At the same time, another request updates the same very same data

Ice provides an elegant solution for this problem with the marshaled-result metadata, available for the following language mappings:

- C++11
- C#
- Java
- Python

Back to Top ^

Thread Creation and Destruction Hooks

On occasion, it is necessary to intercept the creation and destruction of threads created by the Ice run time, for example, to interoperate with libraries that require applications to make thread-specific initialization and finalization calls (such as COM's `CoInitializeEx` and `CoUninitialize`). Ice provides callbacks to inform an application when each run-time thread is created and destroyed.

The callback or callbacks are registered through the `InitializationData` parameter passed to `initialize`:

C++11

```
struct InitializationData
{
    // ...
    std::function<void()> threadStart;
    std::function<void()> threadStop;
};
```

C++98

```
class ThreadNotification : public IceUtil::Shared
{
public:
    virtual void start() = 0;
    virtual void stop() = 0;
};
typedef IceUtil::Handle<ThreadNotification> ThreadNotificationPtr;


struct InitializationData
{
    // ...
    ThreadNotificationPtr threadHook;
};
```

C#

```
public class InitializationData
{
    // ...
    public System.Action threadStart;
    public System.Action threadStop;
}
```

Java

```
public class InitializationData
{
    // ...
    public Runnable threadStart;
    public Runnable threadStop;
}
```

Java Compat

```
public interface ThreadNotification
{
    void start();
    void stop();
}


public class InitializationData
{
    // ...
    ThreadNotification threadHook;
}
```

Python

```
initData = Ice.InitializationData()
initData.threadStart = lambda: # handle thread start...
initData.threadStop = lambda: # handle thread stop...
```

To receive notification of thread creation and destruction, you must implement and register these callbacks. They will be called by the Ice run time by each thread as soon as it is created, and just before it exits.

For example, you could define callbacks and register them with the Ice run time as follows:

C++11

```
int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.start = [] { cout << "start: id = " << std::this_thread::get_id() << endl; };
    initData.stop = [] { cout << "stop: id = " << std::this_thread::get_id() << endl; };
    Ice::CommunicatorHolder ich(argc, argv, initData);

    // ...
}
```

C++98

```
class MyHook : public Ice::ThreadNotification
{
public:
    void start()
    {
        cout << "start: id = " << ThreadControl().id() << endl;
    }
    void stop()
    {
        cout << "stop: id = " << ThreadControl().id() << endl;
    }
};

int
main(int argc, char* argv[])
{
    Ice::InitializationData initData;
    initData.threadHook = new MyHook;
    Ice::CommunicatorHolder ich(argc, argv, initData);

    // ...
}
```

Back to Top ^

Installing Thread Hooks with a Plug-in

The thread hook facility described above requires that you modify a program's source code in order to receive callbacks when threads in the Ice run time are created and destroyed. It is also possible to install thread hooks using the Ice plug-in facility, which is useful for adding thread hooks to an existing program that you cannot (or prefer not to) modify.

Ice provides a base class named `ThreadHookPlugin` for C++, Java, and C# that supplies the necessary functionality:

C++11

```
namespace Ice
{
    class ThreadHookPlugin : public Ice::Plugin
    {
    public:

        ThreadHookPlugin(const std::shared_ptr<Communicator>& communicator, std::function<void()>, std::
function<void()>);

        virtual void initialize();
        virtual void destroy();
    };
}
```

C++98

```
namespace Ice
{
    class ThreadHookPlugin : public Ice::Plugin
    {
    public:

        ThreadHookPlugin(const CommunicatorPtr& communicator, const ThreadNotificationPtr&);

        virtual void initialize();
        virtual void destroy();
    };
}
```

C#

```
namespace Ice
{
    public class ThreadHookPlugin : Plugin
    {
        public ThreadHookPlugin(Communicator communicator, System.Action threadStart, System.Action threadStop)
{ ... }

        public void initialize() {}
        public void destroy() {}
    }
}
```

Java

```
package com.zeroc.Ice;
public class ThreadHookPlugin implements Plugin
{
    public ThreadHookPlugin(Communicator communicator, Runnable threadStart, Runnable threadStop) { ... }

    @Override
    public void initialize() {}

    @Override
    public void destroy() {}
}
```

Java Compat

```
package Ice;
public class ThreadHookPlugin implements Plugin
{
    public ThreadHookPlugin(Communicator communicator, ThreadNotificationHook threadHook) { ... }

    @Override
    public void initialize() {}

    @Override
    public void destroy() {}
}
```

The `ThreadHookPlugin` constructor installs the given thread callbacks into the specified communicator. The `initialize` and `destroy` methods are empty, but you can subclass `ThreadHookPlugin` and override these methods if necessary.

In order to create a thread hook plug-in, you must do the following:

- Define and export a factory class (for Java and C#) or factory function (for C++) that returns an instance of `ThreadHookPlugin`, as described in the plug-in API.
- Implement the callback(s) that you will pass to the `ThreadHookPlugin` constructor.
- Package your code into a format that is suitable for dynamic loading, such as a shared library or DLL for C++ or an assembly for C#.

See the Plug-in Facility for more details on how to package and register your plug-in.

Back to Top ^

See Also

- Communicator Initialization
- Plug-in Facility
- Plug-in API
- Ice.Plugin.*
- Ice.InitPlugins
- Ice.PluginLoadOrder

Previous Next