Setting Properties

Previous Next

The `setProperty` operation on the `Properties` interface sets a property to the specified value:

Slice

```
module Ice
{
    local interface Properties
    {
        void setProperty(string key, string value);
        // ...
    }
}
```

You can clear a property by setting it to the empty string.

For properties that control the Ice run time and its services (that is, properties that start with one of the reserved prefixes), this operation is useful only if you call it *before* you call `initialize`. This is because property values are usually read by the Ice run time only once, when you call `initialize`, so the Ice run time does not pay attention to a property value that is changed after you have initialized a communicator. Of course, this begs the question of how you can set a property value and have it also recognized by a communicator.

To permit you to set properties before initializing a communicator, the Ice run time provides an overloaded helper function or method called `createProperties` that creates a property set.

`createProperties` is provided in all language mappings with overloads that accept the following parameters:

1. no parameter at all
   This `createProperties` creates an empty property set, and does *not* check `ICE_CONFIG` for a configuration file to parse.

2. the arguments given to the application plus a default property set
   This `createProperties` returns a property set that contains all the property settings that are passed as the default, plus any property settings in the arguments. If the argument vector sets a property that is also set in the passed default property set, the setting in the argument vector overrides the default. It also looks for the `--Ice.Config` option; if the argument vector specifies a configuration file, the configuration file is parsed. The order of precedence of property settings, from lowest to highest, is:
   - Property settings passed in the default parameter
   - Property settings set in the configuration file
   - Property settings in the argument vector (parsed as described in Command-Line Parsing and Initialization)

   This `createProperties` also looks for the setting of the `ICE_CONFIG` environment variable and, if that variable specifies a configuration file, parse that file. (However, an explicit `--Ice.Config` option in the argument vector or the `defaults` parameter overrides any setting of the `ICE_CONFIG` environment variable.)

3. the arguments given to the application
   This `initialize` is like the initialize in `2.` above, with an empty default property set.

C++11

```
namespace Ice
{
    // (1): empty properties
    std::shared_ptr<Properties> createProperties();

    // (2) and (3): clone default properties parameter (if present) then parse args
    std::shared_ptr<Properties> createProperties(int argc&, const char* argv[], const std::
shared_ptr<Properties>& = nullptr);
    std::shared_ptr<Properties> createProperties(StringSeq& args, const std::shared_ptr<Properties>& defaults =
nullptr);

#ifdef _WIN32
    // (2) and (3): clone default properties parameter (if present) then parse args
    std::shared_ptr<Properties> createProperties(int argc&, const wchar_t* argv[], const std::
shared_ptr<Properties>& defaults = nullptr);
#endif
}
```

C++98

```
namespace Ice
{
    // (1): empty properties
    PropertiesPtr createProperties();

    // (2) and (3): clone default properties parameter (if present) then parse args
    PropertiesPtr createProperties(int argc&, const char* argv[], const PropertiesPtr& = 0);
    PropertiesPtr createProperties(StringSeq& args, const PropertiesPtr& defaults = 0);

#ifdef _WIN32
    // (2) and (3): clone default properties parameter (if present) then parse args
    PropertiesPtr createProperties(int argc&, const wchar_t* argv[], const PropertiesPtr& defaults = 0);
#endif
}
```

C#

```
namespace Ice
{
    public sealed class Util
    {
        // (1): empty properties
        public static Properties createProperties() { ... }

        // (2): clone default properties and then parse args
        public static Properties createProperties(ref string[] args, Properties defaults) { ... }

        // (3): parse args
        public static Properties createProperties(ref string[] args) { ... }
    }
}
```

Java

```
package com.zeroc.Ice;

public final class Util
{
    // (1): empty properties
    public static Properties createProperties() { ... }

    // (2): clone default properties and then parse args
    public static Properties createProperties(String[] args, Properties defaults)  { ... }
    public static Properties createProperties(String[] args, Properties defaults, java.util.List<String>
remainingArgs)  { ... }
    // (3): parse args
    public static Properties createProperties(String[] args) { ... }
    public static Properties createProperties(String[] args, java.util.List<String> remainingArgs)  { ... }

    // ...
}
```

Java Compat

```
package Ice;

public final class Util
{
    // (1): empty properties
    public static Properties createProperties() { ... }
    // (2): clone default properties and then parse args
    public static Properties createProperties(String[] args, Properties defaults) { ... }
    public static Properties createProperties(StringSeqHolder args, Properties defaults) { ... }

    // (3): parse args
    public static Properties createProperties(String[] args) { ... }
    public static Properties createProperties(StringSeqHolder args) { ... }

    // ...
}
```

JavaScript

```
Ice.createProperties = function(args, defaults) { ... }
```

MATLAB

```
% in Ice package

% Valid arguments:
%
% (1): no parameter at all, returns empty properties
% props = Ice.createProperties();
% (2): clone default properties and then parse args (a cell array of character vectors)
% [props, remainingArgs] = createProperties(args, defaults);
% (3): parse args (a cell array of character vectors)
% [props, remainingArgs] = createProperties(args);

function [props, remainingArgs] = createProperties(varargin)
```

ObjC

```
@interface ICEUtil : NSObject

// (1): empty properties
+(id<ICEProperties>) createProperties;

// (3): parse args
+(id<ICEProperties>) createProperties:(int*)argc argv:(char*[])argv;
...
@end
```

PHP

```
namespace Ice
{
    function createProperties(args=array(), defaults=null) { ... }
}
```

Python

```
# in Ice module
def createProperties(args=[], defaults=None)
```

Ruby

```
module Ice
    def createProperties(args=[], defaults=None)
```

Like `initialize`, `createProperties` strips Ice-related command-line options from the passed argument vector.

`createProperties` is useful if you want to ensure that a property is set to a particular value, regardless of any setting of that property in a configuration file or in the argument vector. Here is an example:

C++11

```
// Get the initialized property set.
//
auto props = Ice::createProperties(argc, argv);

// Make sure that network and protocol tracing are off.
//
props->setProperty("Ice.Trace.Network", "0");
props->setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
Ice::InitializationData initData;
initData.properties = props;
Ice::CommunicatorHolder  ich(initData);
```

Java

```
// Get the initialized property set.
//
com.zeroc.Ice.Properties props = com.zeroc.Ice.Util.createProperties(args);

// Make sure that network and protocol tracing are off.
//
props.setProperty("Ice.Warn.Connections", "0");
props.setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
com.zeroc.Ice.InitializationData initData = new com.zeroc.Ice.InitializationData();
initData.properties = props;
com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.initialize(initData);
```

MATLAB

```
% Get the initialized property set.
%
props = Ice.createProperties(args);

% Make sure that network and protocol tracing are off.
%
props.setProperty('Ice.Warn.Connections', '0');
props.setProperty('Ice.Trace.Protocol', '0');

% Initialize a communicator with these properties.
%
initData = Ice.InitializationData();
initData.properties_ = props;
communicator = Ice.initialize(initData);
```

PHP

```
// Get the initialized property set.
//
$props = Ice\createProperties($args);

// Make sure that network and protocol tracing are off.
//
$props->setProperty("Ice.Trace.Network", "0");
$props->setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
$initData = new Ice\InitializationData();
$initData->properties = $props;
$communicator = Ice\initialize($initData);
```

Python

```
# Get the initialized property set.
#
props = Ice.createProperties(sys.argv)

# Make sure that network and protocol tracing are off.
#
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")

# Initialize a communicator with these properties.
#
initData = Ice.InitializationData()
initData.properties = props
communicator = Ice.initialize(initData)
```

Ruby

```
# Get the initialized property set.
#
props = Ice::createProperties(ARGV)

# Make sure that network and protocol tracing are off.
#
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")

# Initialize a communicator with these properties.
#
initData = Ice::InitializationData.new
initData.properties = props
initData = Ice::initialize(initData)
```

You can also set properties "in bulk" remotely through the Properties facet of Ice Admin.

Back to Top ^

See Also

- The Properties Interface
- Using Configuration Files
- Command-Line Parsing and Initialization
- Reading Properties
- Parsing Properties
- Communicator Initialization

Previous Next