***roscpp overview (/roscpp/Overview)**: Initialization and Shutdown (/roscpp/Overview/Initialization%20and%20Shutdown) | Basics (/roscpp/Overview/Messages) | Advanced: Traits [ROS C Turtle] (/roscpp/Overview/MessagesTraits) | Advanced: Custom Allocators [ROS C Turtle] (/roscpp/Overview/MessagesCustomAllocators) | Advanced: Serialization and Adapting Types [ROS C Turtle] (/roscpp/Overview/MessagesSerializationAndAdaptingTypes) | Publishers and Subscribers | Services (/roscpp/Overview/Services) | Parameter Server (/roscpp/Overview/Parameter%20Server) | Timers (Periodic Callbacks) (/roscpp/Overview/Timers) | NodeHandles (/roscpp/Overview/NodeHandles) | Callbacks and Spinning (/roscpp/Overview/Callbacks%20and%20Spinning) | Logging (/roscpp/Overview/Logging) | Names and Node Information (/roscpp/Overview/Names%20and%20Node%20Information) | Time (/roscpp/Overview/Time) | Exceptions (/roscpp/Overview/Exceptions) | Compilation Options (/roscpp/Overview/Compilation%20Options) | Advanced: Internals (/roscpp/Overview/Internals) | tf/Overview (/tf/Overview) | tf/Tutorials (/tf/Tutorials) | C++ Style Guide (/CppStyleGuide)*

目录

1. Publishing to a Topic

See also: ros::NodeHandle::advertise() API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html#41fcdf7fd2a2bfa1c42d14d35f8f5a8e), ros::Publisher API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1Publisher.html), ros::NodeHandle API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html)

Creating a handle to publish messages to a topic is done using the `ros::NodeHandle` class, which is covered in more detail in the NodeHandles overview (/roscpp/Overview/NodeHandles).

The `NodeHandle::advertise()` methods are used to create a `ros::Publisher` which is used to publish on a topic, e.g.:

切换行号显示

```
1 ros::Publisher pub = nh.advertise<std_msgs::String>("topic_name", 5);
2 std_msgs::String str;
3 str.data = "hello world";
4 pub.publish(str);
```

Note: it is possible (though rare) for `NodeHandle::advertise()` to return an empty `ros::Publisher`. In the future these cases will probably throw exceptions, but for now they simply print an error. You can check for this with:

切换行号显示

```
1 if (!pub)
2 {
3 ...
4 }
```

1.1 Intraprocess Publishing

When a publisher and subscriber to the same topic both exist inside the same node, roscpp can skip the serialize/deserialize step (potentially saving a large amount of processing and latency). It can only do this though if the message is published as a `shared_ptr`:

切换行号显示

```
1 ros::Publisher pub = nh.advertise<std_msgs::String>("topic_name", 5);
2 std_msgs::StringPtr str(new std_msgs::String);
3 str->data = "hello world";
4 pub.publish(str);
```

This form of publishing is what can make nodelets (/nodelet) such a large win over nodes in separate processes.

Note that when publishing in this fashion, there is an implicit contract between you and roscpp: you may not modify the message you've sent after you send it, since that pointer will be passed directly to any intraprocess subscribers. If you want to send another message, you must allocate a new one and send that.

1.2 Publisher Options

The signature for the simple version of `advertise()` is:

切换行号显示

```
1 template<class M>
2 ros::Publisher advertise(const std::string& topic, uint32_t queue_size,
bool latch = false);
```

`M` [required]

This is a template argument specifying the message type to be published on the topic

`topic` [required]

This is the topic to publish on.

`queue_size` [required]

This is the size of the outgoing message queue. If you are publishing faster than roscpp can send the messages over the wire, roscpp will start dropping old messages. A value of 0 here means an infinite queue, which can be dangerous. See the rospy documentation on choosing a good queue_size (/rospy/Overview/Publishers%20and%20Subscribers#Choosing_a_good_queue_size) for more information.

`latch` [optional]

Enables "latching" on a connection. When a connection is latched, the last message published is saved and automatically sent to any future subscribers that connect. This is useful for slow-changing to static data like a map. Note that if there are multiple publishers on the same topic, instantiated in the same node, then only the last published message from that node will be sent, as opposed to the last published message from each publisher on that single topic.

1.3 Other Operations on the Publisher Handle

`ros::Publisher` is reference counted internally -- this means that copying them is very fast, and does not create a "new" version of the `ros::Publisher`. Once all copies of a `ros::Publisher` are destructed the topic will be shutdown. There are some exceptions to this:

1. `ros::shutdown()` is called -- this shuts down all publishers (and everything else).
2. `ros::Publisher::shutdown()` is called. This will shutdown this topic, unless there have been...
3. Multiple calls to `NodeHandle::advertise()` for the same topic, with the same message type. In this case all the `ros::Publisher`s on a specific topic are treated as copies of each other.

`ros::Publisher` implements the `==`, `!=` and `<` operators, and it is possible to use them in `std::map`, `std::set`, etc.

You can retrieve the topic of a publisher with the `ros::Publisher::getTopic()` method.

1.4 publish() behavior and queueing

`publish()` in roscpp is asynchronous, and only does work if there are subscribers connected on that topic. `publish()` itself is meant to be very fast, so it does as little work as possible:

1. Serialize the message to a buffer
2. Pushes that buffer onto a queue for later processing

The queue it's pushed onto is then serviced as soon as possible by one of roscpp's internal threads, where it gets put onto a queue for each connected subscriber -- this second set of queues are the ones whose size is set with the queue_size parameter in `advertise()`. If one of these queues fills up the oldest message will be dropped before adding the next message to the queue.

Note that there may also be an OS-level queue at the transport level, such as the TCP/UDP send buffer.

2. Subscribing to a Topic

See also: ros::NodeHandle::subscribe() API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html#3b8e4b07d397119cd5c5e4439b170cbc), ros::Subscriber API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1Subscriber.html), ros::NodeHandle API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html), ros::TransportHints API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1TransportHints.html)

Subscribing to a topic is also done using the `ros::NodeHandle` class (covered in more detail in the NodeHandles overview (/roscpp/Overview/NodeHandles)). A simple subscription using a global function would look like:

切换行号显示

```
1 void callback(const std_msgs::StringConstPtr& str)
2 {
3 ...
4 }
5
6 ...
7 ros::Subscriber sub = nh.subscribe("my_topic", 1, callback);
```

2.1 Subscriber Options

There are many different versions of `ros::NodeHandle::subscribe()`, but the simple versions boil down to:

切换行号显示

```
1 template<class M>
2 ros::Subscriber subscribe(const std::string& topic, uint32_t queue_size
, <callback, which may involve multiple arguments>, const ros::TransportHint
s& transport_hints = ros::TransportHints());
```

`M` [usually unnecessary]

This is a template argument specifying the message type to be published on the topic. For most versions of this `subscribe()` you do not need to explicitly define this, as the compiler can deduce it from the callback you specify.

`topic`

The topic to subscribe to

`queue_size`

This is the incoming message queue size roscpp will use for your callback. If messages are arriving too fast and you are unable to keep up, roscpp will start throwing away messages. A value of 0 here means an infinite queue, which can be dangerous.

`<callback>`

Depending on the version of `subscribe()` you're using, this may be any of a few things. The most common is a class method pointer and a pointer to the instance of the class. This is explained in more detail later.

`transport_hints`

The transport hints allow you to specify hints to roscpp's transport layer. This lets you specify things like preferring a UDP transport, using tcp nodelay, etc. This is explained in more detail later.

2.2 Callback Signature

The signature for the callback is:

```
切换行号显示

1 void callback(const boost::shared_ptr<Message const>&);
```

Every generated message provides typedefs for the shared pointer type, so you can also use, for example:

```
切换行号显示

1 void callback(const std_msgs::StringConstPtr&);
```

or

```
切换行号显示

1 void callback(const std_msgs::String::ConstPtr&);
```

Other Valid Signatures [ROS 1.1+]

As of ROS 1.1 we also support variations on the above callbacks:

```
切换行号显示

1 void callback(boost::shared_ptr<std_msgs::String const>);
2 void callback(std_msgs::StringConstPtr);
3 void callback(std_msgs::String::ConstPtr);
4 void callback(const std_msgs::String&);
5 void callback(std_msgs::String);
6 void callback(const ros::MessageEvent<std_msgs::String const>&);
```

You can also request a non-const message, in which case a copy will be made if necessary (i.e. there are multiple subscriptions to the same topic in a single node):

```
切换行号显示

1 void callback(const boost::shared_ptr<std_msgs::String>&);
2 void callback(boost::shared_ptr<std_msgs::String>);
3 void callback(const std_msgs::StringPtr&);
4 void callback(const std_msgs::String::Ptr&);
5 void callback(std_msgs::StringPtr);
6 void callback(std_msgs::String::Ptr);
7 void callback(const ros::MessageEvent<std_msgs::String>&);
```

The `MessageEvent` versions are described below.

2.3 Callback Types

roscpp supports any callback supported by boost::function (http://www.boost.org/doc/libs/1_37_0/doc/html/function.html):

1. functions
2. class methods
3. functor objects (including boost::bind (http://www.boost.org/doc/libs/1_37_0/libs/bind/bind.html))

2.3.1 Functions

Functions are the easiest to use:

切换行号显示

```
1 void callback(const std_msgs::StringConstPtr& str)
2 {
3 ...
4 }
5
6 ...
7 ros::Subscriber sub = nh.subscribe("my_topic", 1, callback);
```

2.3.2 Class Methods

Class methods are also easy, though they require an extra parameter:

切换行号显示

```
1 void Foo::callback(const std_msgs::StringConstPtr& message)
2 {
3 }
4
5 ...
6 Foo foo_object;
7 ros::Subscriber sub = nh.subscribe("my_topic", 1, &Foo::callback, &foo_
object);
```

2.3.3 Functor Objects

A functor object is a class that declares `operator()`, e.g.:

切换行号显示

```
1 class Foo
2 {
3 public:
4   void operator()(const std_msgs::StringConstPtr& message)
5   {
6   }
7 };
```

A functor passed to `subscribe()` must be **copyable**. The `Foo` functor could be used with `subscribe()` like so:

切换行号显示

```
1 ros::Subscriber sub = nh.subscribe<std_msgs::String>("my_topic", 1, Foo
());
```

Note: when using functor objects (like boost::bind, for example) you must explicitly specify the message type as a template argument, because the compiler cannot deduce it in this case.

2.4 MessageEvent [ROS 1.1+]

The MessageEvent (http://www.ros.org/doc/api/roscpp_traits/html/classros_1_1MessageEvent.html) class allows you to retrieve metadata about a message from within a subscription callback:

切换行号显示

```
1 void callback(const ros::MessageEvent<std_msgs::String const>& event)
2 {
3   const std::string& publisher_name = event.getPublisherName();
4   const ros::M_string& header = event.getConnectionHeader();
5   ros::Time receipt_time = event.getReceiptTime();
6
7   const std_msgs::StringConstPtr& msg = event.getMessage();
8 }
```

(see ROS/Connection Header (/ROS/Connection%20Header) for details on the fields in the connection header)

2.5 Queueing and Lazy Deserialization

When a message first arrives on a topic it gets put into a queue whose size is specified by the `queue_size` parameter in `subscribe()`. If the queue is full and a new message arrives, the oldest message will be thrown out. Additionally, the message is not actually deserialized until the first callback which needs it is about to be called.

2.6 Transport Hints

See also: ros::TransportHints API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1TransportHints.html)

`ros::TransportHints` are used to specify hints about how you want the transport layer to behave for this topic. For example, if you wanted to specify an "unreliable" connection (and not allow a "reliable" connection as a fall back):

切换行号显示

```
1 ros::Subscriber sub = nh.subscribe("my_topic", 1, callback, ros::Transp
ortHints().unreliable());
```

Note that `ros::TransportHints` uses the Named Parameter Idiom (http://www.cs.technion.ac.il/users/yechiel/c++-faq/named-parameter-idiom.html), a form of method-chaining. This means you can do things like:

切换行号显示

```
1 ros::TransportHints()
2         .unreliable()
3         .reliable()
4         .maxDatagramSize(1000)
5         .tcpNoDelay();
```

As is this example you can specify multiple different transport options (`unreliable` as well as `reliable`). If the publisher on the topic you're subscribing to does not support the first connection (`unreliable`), the connection will be made with the second (`reliable`) if supported. In this case the order of the two methods is important since it determines the order of considered transports.

You cannot currently specify the transport hints on the `Publisher` side. This will likely be an option in the future.

Wiki: roscpp/Overview/Publishers and Subscribers (2018-04-10 13:10:52由FrancescoW (/FrancescoW)编辑)