API参考

1. SystemManager接口

SystemManager是整个系统的核心管理器，提供系统初始化、运行和关闭的功能。

1.1 获取SystemManager实例

```
SystemManager& sys = SystemManager::instance();
```

说明：SystemManager采用单例模式，通过`instance()`静态方法获取全局唯一实例。

1.2 初始化系统

```
// 默认初始化
void init();

// 指定端口初始化
void init(int port);

// 指定端口和节点名初始化
void init(int port, std::string node_name);

// 指定节点名初始化
void init(const std::string& node_name);
```

参数说明:

- `port`：节点监听的端口号，用于与其他节点通信
- `node_name`：节点名称，用于在网络中标识节点

使用示例:

```
SystemManager::instance().init(12345, "my_node");
```

1.3 运行系统

```
// 进入主循环，处理消息队列
void spin();

// 处理一次消息队列中的消息
void spinOnce();
```

说明:

- `spin()`方法会阻塞当前线程，进入无限循环，不断处理消息队列中的消息
- `spinOnce()`方法仅处理一次消息队列中的消息，不会阻塞线程

使用示例:

```
// 启动系统后进入主循环
SystemManager::instance().init(12345, "my_node");
SystemManager::instance().spin();
```

1.4 关闭系统

```
void shutdown();
```

说明：关闭系统，释放资源，断开所有连接。

1.5 获取系统组件

```
// 获取消息队列指针
std::shared_ptr<MessageQueue> getMessageQueue() const;

// 获取PollManager指针
std::shared_ptr<PollManager> getPollManager() const;

// 获取EventLoop指针
std::shared_ptr<muduo::net::EventLoop> getEventLoop() const;

// 获取全局RPC客户端
std::shared_ptr<RosRpcClient> getRpcClient() const;

// 获取节点信息
NodeInfo getNodeInfo() const;
```

2. NodeHandle接口

NodeHandle是用户与系统交互的主要接口，提供创建发布者、订阅者和定时器的功能。

2.1 创建NodeHandle实例

```
NodeHandle nh;
```

2.2 创建订阅者

2.2.1 函数对象版本

```
template<typename MsgType>
std::shared_ptr<Subscriber> subscribe(
    const std::string& topic,
    uint32_t queue_size,
    std::function<void(const std::shared_ptr<MsgType>&)> callback);
```

参数说明:

- `topic`: 要订阅的主题名称
- `queue_size`: 消息队列大小
- `callback`: 消息处理回调函数

使用示例:

```
nh.subscribe<example::SensorData>(
    "sensor_data",
    10,
    [](const std::shared_ptr<example::SensorData>& msg) {
        // 处理传感器数据消息
        std::cout << "Received sensor data with id: " << msg->sensor_id()
<< std::endl;
    }
);
```

2.2.2 类成员函数版本

```
template<typename MsgType, typename Class>
std::shared_ptr<Subscriber> subscribe(
    const std::string& topic,
    uint32_t queue_size,
    void(Class::*callback)(const std::shared_ptr<MsgType>&),
    Class* instance);
```

参数说明:

- `topic`: 要订阅的主题名称
- `queue_size`: 消息队列大小
- `callback`: 类成员函数指针
- `instance`: 类实例指针

使用示例:

```

```

```
class DataProcessor {
public:
    void processData(const std::shared_ptr<example::SensorData>& msg) {
        // 处理传感器数据消息
    }
};

DataProcessor processor;
nh.subscribe<example::SensorData>(
    "sensor_data",
    10,
    &DataProcessor::processData,
    &processor
);
```

2.2.3 非模板版本

```
std::shared_ptr<Subscriber> subscribe(
    const std::string& topic,
    uint32_t queue_size,
    const std::string& msg_type_name,
    MessageQueue::Callback callback);
```

参数说明:

- `topic`: 主题名称
- `queue_size`: 消息队列大小
- `msg_type_name`: 消息类型名称
- `callback`: 回调函数

2.3 创建发布者

```
template<typename MsgType>
std::shared_ptr<Publisher<MsgType>> advertise(const std::string& topic);
```

参数说明:

- `topic`: 要发布的主题名称

返回值: 返回一个Publisher智能指针，用于发布消息

使用示例:

```
auto sensor_pub = nh.advertise<example::SensorData>("sensor_data");

// 发布消息
```

```
example::SensorData data;
data.set_sensor_id(1);
data.set_value(23.5);
data.set_timestamp(SystemManager::instance().now().secondsSinceEpoch());
sensor_pub->publish(data);
```

2.4 创建定时器

```
std::shared_ptr<Timer> createTimer(
    double period,
    const TimerCallback& callback,
    bool oneshot = false);
```

参数说明:

- `period`: 定时器周期，单位为秒
- `callback`: 定时器回调函数
- `oneshot`: 是否为一次性定时器，默认为false（周期性）

返回值: 返回一个Timer智能指针，用于控制定时器

使用示例:

```
auto timer = nh.createTimer(
    0.1,  // 100ms周期
    [](const TimerEvent& event) {
        std::cout << "Timer triggered at: " << event.current_real <<
std::endl;
    }
);
```

3. Publisher接口

Publisher用于向特定主题发布消息。

3.1 发布消息

```
template<typename T>
void publish(const T& msg);
```

参数说明:

- `msg`: 要发布的消息对象，必须是Protobuf消息类型

使用示例:

```
example::SensorData data;
data.set_sensor_id(1);
data.set_value(23.5);
publisher->publish(data);
```

3.2 取消注册

```
void unregister();
```

说明：取消发布者的注册，不再发布消息。

4. Subscriber接口

Subscriber用于订阅特定主题的消息。Subscriber的生命周期由智能指针管理，当Subscriber对象被销毁时，会自动取消订阅。

5.1 发布Marker消息

```
// 创建Marker发布者
auto marker_pub = nh.advertise<visualization_msgs::Marker>
("visualization_marker");

// 填充Marker消息
visualization_msgs::Marker marker;
marker.set_ns("basic_shapes");
marker.set_id(0);
marker.set_type(visualization_msgs::MarkerType::CUBE);
marker.set_action(visualization_msgs::MarkerAction::ADD);

// 设置位置和姿态
marker.mutable_pose()->mutable_position()->set_x(0.0);
marker.mutable_pose()->mutable_position()->set_y(0.0);
marker.mutable_pose()->mutable_position()->set_z(0.0);
marker.mutable_pose()->mutable_orientation()->set_w(1.0);

// 设置颜色和大小
marker.mutable_color()->set_r(0.0);
marker.mutable_color()->set_g(1.0);
marker.mutable_color()->set_b(0.0);
marker.mutable_color()->set_a(1.0);
marker.mutable_scale()->set_x(1.0);
marker.mutable_scale()->set_y(1.0);
marker.mutable_scale()->set_z(1.0);

// 发布消息
marker_pub->publish(marker);
```

5.2 发布MarkerArray消息

```
// 创建MarkerArray发布者
auto marker_array_pub = nh.advertise<visualization_msgs::MarkerArray>
("visualization_marker_array");

// 填充MarkerArray消息
visualization_msgs::MarkerArray marker_array;

// 添加多个Marker到MarkerArray
visualization_msgs::Marker marker1;
// 设置marker1属性...
*marker_array.add_markers() = marker1;

visualization_msgs::Marker marker2;
// 设置marker2属性...
*marker_array.add_markers() = marker2;

// 发布消息
marker_array_pub->publish(marker_array);
```

5.3 发布路径可视化

```
// 创建路径发布者
auto path_pub = nh.advertise<visualization_msgs::Marker>("path");

// 创建路径Marker
visualization_msgs::Marker path_marker;
path_marker.set_ns("path");
path_marker.set_id(0);
path_marker.set_type(visualization_msgs::MarkerType::LINE_STRIP);
path_marker.set_action(visualization_msgs::MarkerAction::ADD);
path_marker.set_lifetime(-1); // 永久存在

// 设置颜色和线宽
path_marker.mutable_color()->set_r(1.0);
path_marker.mutable_color()->set_g(0.0);
path_marker.mutable_color()->set_b(0.0);
path_marker.mutable_color()->set_a(1.0);
path_marker.mutable_scale()->set_x(0.1); // 线宽

// 添加路径点
for (const auto& point : path_points) {
    geometry_msgs::Point* p = path_marker.add_points();
    p->set_x(point.x);
    p->set_y(point.y);
    p->set_z(point.z);
}
```

```
// 发布消息
path_pub->publish(path_marker);
```

6. 消息定义

系统使用Protobuf定义消息类型，主要包括以下几种预定义消息类型：

6.1 基础消息类型

- `example/SensorData`：传感器数据消息（包含sensor_id、value、timestamp）
- `example/ControlCommand`：控制命令消息（包含cmd_id、cmd）
- `example/Heartbeat`：心跳消息（用于长连接维持）

6.2 几何消息类型

- `geometry_msgs/Point`：三维点（包含x、y、z）
- `geometry_msgs/Quaternion`：四元数（包含x、y、z、w）
- `geometry_msgs/Pose`：位姿（位置和姿态，包含position和orientation）
- `geometry_msgs/Vector3`：三维向量（包含x、y、z）
- `geometry_msgs/Odometry`：里程计信息（包含pose、linear_velocity、angular_velocity）

6.3 可视化消息类型

- `visualization_msgs/Marker`：可视化标记
- `visualization_msgs/MarkerArray`：标记数组
- `visualization_msgs/ColorRGBA`：颜色信息（RGBA格式）

6.4 创建自定义消息类型

simple_ros系统允许用户创建自定义的Protobuf消息类型，步骤如下：

6.4.1 创建.proto文件

首先，在项目的proto目录下创建一个新的.proto文件，定义消息结构。例如，创建一个名为`my_msgs.proto`的文件：

```
syntax = "proto3";

package my_msgs;

// 定义自定义消息类型
message MyCustomMsg {
  int32 id = 1;
  string name = 2;
  double value = 3;
  repeated double data_points = 4;
}

message StatusMsg {
```

```
  enum Status {
    OK = 0;
    WARNING = 1;
    ERROR = 2;
  }
  Status status = 1;
  string message = 2;
  int64 timestamp = 3;
}
```

6.4.2 编译自定义消息

使用protoc编译器编译自定义消息，生成对应的C++代码。系统提供了编译脚本`proto.sh`，可以使用该脚本编译所有proto文件：

```
cd <path_to_simple_ros>
./proto.sh
```

6.4.3 在代码中使用自定义消息类型

编译完成后，可以在代码中包含生成的头文件，并使用自定义消息类型：

```
#include "my_msgs.pb.h"

// 创建发布者
auto pub = nh.advertise<my_msgs::MyCustomMsg>("custom_topic");

// 创建并发布消息
my_msgs::MyCustomMsg msg;
msg.set_id(1);
msg.set_name("test_message");
msg.set_value(3.14);
msg.add_data_points(1.0);
msg.add_data_points(2.0);
msg.add_data_points(3.0);
pub->publish(msg);

// 订阅自定义消息
nh.subscribe<my_msgs::MyCustomMsg>(
    "custom_topic",
    10,
    [](const std::shared_ptr<my_msgs::MyCustomMsg>& msg) {
        std::cout << "Received custom message: " << msg->name() <<
std::endl;
        std::cout << "Value: " << msg->value() << std::endl;
    }
);
```

7. 工具程序

系统提供了一些工具程序，用于调试和测试：

7.1 master

主节点程序，负责节点注册和发现。

启动方法：

```
./master
```

功能：

- 维护节点列表
- 管理主题发布和订阅关系
- 提供节点发现服务

7.2 rosnode

节点管理工具，用于查看和管理节点。

常用命令：

7.2.1 查看节点列表

```
./rosnode list
```

功能：列出当前运行的所有节点。

7.2.2 查看节点信息

```
./rosnode info <node_name>
```

功能：显示指定节点的详细信息，包括发布的主题、订阅的主题等。

参数：

- `<node_name>`：节点名称

7.3 rostopic

主题管理工具，用于查看和发布主题。

常用命令：

7.3.1 查看主题列表

```
./rostopic list
```

功能：列出当前所有可用的主题。

7.3.2 查看主题信息

```
./rostopic info <topic_name>
```

功能：显示指定主题的详细信息，包括发布者、订阅者、消息类型等。

参数：

- `<topic_name>`：主题名称

7.3.3 查看主题数据

```
./rostopic echo <topic_name>
```

功能：实时显示指定主题上发布的消息内容。

参数：

- `<topic_name>`：主题名称

7.3.4 查看主题频率

```
./rostopic hz <topic_name>
```

功能：显示指定主题的消息发布频率。

参数：

- `<topic_name>`：主题名称

7.4 foxglove_bridge_node

Foxglove Bridge节点，用于连接Foxglove Studio进行可视化。

启动方法：

```
./foxglove_bridge_node
```

功能:

- 提供WebSocket服务器，默认端口为8765
- 转发主题消息到Foxglove Studio
- 支持多种可视化消息类型

8. 错误处理

系统使用异常处理错误情况，主要包括以下几种常见异常：

- 连接异常：当无法连接到其他节点时抛出
- 消息序列化异常：当消息序列化或反序列化失败时抛出
- 资源不足异常：当系统资源不足时抛出
- 超时异常：当操作超时时抛出

用户在使用API时应当适当捕获这些异常，确保程序的稳定运行。