示例代码解析

1. 概述

simple_ros系统提供了多个示例代码，用于展示系统的各种功能和用法。本文档将详细解析这些示例代码，帮助用户理解如何使用系统的各种功能。

目前系统提供的主要示例包括：

- **marker_publisher_example.cpp**：演示如何发布可视化标记，用于在Foxglove Studio中显示机器人模型和路径
- **quad_simulator.cpp**：演示四旋翼模拟器的实现
- **quad_visualizer.cpp**：演示四旋翼可视化节点的实现
- **foxglove_bridge_tool.cpp**：演示Foxglove Bridge工具的使用

本文档将重点解析这些示例代码的实现细节和使用方法。

2. marker_publisher_example.cpp解析

marker_publisher_example.cpp是一个演示如何发布可视化标记的示例程序。这个示例展示了如何创建和发布各种类型的可视化标记，包括机器人模型、路径等。

2.1 整体结构

示例代码的整体结构如下：

```
// 包含必要的头文件
#include "global_init.h"
#include "node_handle.h"
#include "publisher.h"
#include "timer.h"
#include "visualization_msgs/Marker.h"
#include "visualization_msgs/MarkerArray.h"

// 定义一些辅助结构体和函数
struct QuadState {
    // 定义四旋翼状态
};

class MarkerPublisherExample {
public:
    MarkerPublisherExample();
    void run();
private:
    // 私有成员和方法
};

// 主函数
int main(int argc, char** argv) {
```

```
    // 初始化并运行示例
}
```

2.2 QuadState结构体

`QuadState`结构体用于表示四旋翼的状态，包括位置、四元数和各种速度：

```
struct QuadState {
    double x, y, z;           // 位置坐标
    double qx, qy, qz, qw;    // 四元数表示的姿态
    double vx, vy, vz;        // 线速度
    double wx, wy, wz;        // 角速度
};
```

2.3 MarkerPublisherExample类

`MarkerPublisherExample`类是示例程序的主要类，用于管理节点、发布者和定时器，并实现可视化标记的发布功能。

2.3.1 构造函数和初始化

构造函数初始化了节点句柄、发布者和定时器：

```
MarkerPublisherExample::MarkerPublisherExample() {
    // 初始化系统
    SystemManager::instance().init("marker_publisher_example");

    // 创建节点句柄
    nh_ = std::make_shared<NodeHandle>();

    // 创建发布者
    marker_pub_ = nh_->advertise<visualization_msgs::MarkerArray>
("visualization_marker");

    // 创建定时器，每50ms调用一次回调函数
    timer_ = nh_->createTimer(0.05,
std::bind(&MarkerPublisherExample::timerCallback, this,
std::placeholders::_1));

    // 初始化状态
    state_ = std::make_shared<QuadState>();
    time_ = 0.0;

    // 初始化路径点
    path_points_.reserve(5000);
}
```

2.3.2 timerCallback函数

`timerCallback`函数是定时器的回调函数，用于更新状态和发布可视化标记：

```
void MarkerPublisherExample::timerCallback(const TimerEvent& event) {
    // 更新时间
    double dt = event.current_real.secondsSinceEpoch() -
event.last_real.secondsSinceEpoch();
    time_ += dt;

    // 更新状态（简单的正弦运动）
    state_->pos.x() = 2.0 * std::cos(time_);
    state_->pos.y() = 2.0 * std::sin(time_);
    state_->pos.z() = 1.0 + 0.5 * std::sin(2 * time_);

    // 计算四元数（使用微分平坦方法）
    state_->q = computeFlatness(state_->pos, time_);

    // 更新路径点
    if (path_points_.size() < 5000) {
        path_points_.push_back(state_->pos);
    } else {
        path_points_.erase(path_points_.begin());
        path_points_.push_back(state_->pos);
    }

    // 创建并发布可视化标记
    visualization_msgs::MarkerArray marker_array;

    // 创建四旋翼标记
    auto quad_markers = createQuadrotorMarkerArray(*state_);
    marker_array.markers.insert(marker_array.markers.end(),
quad_markers.markers.begin(), quad_markers.markers.end());

    // 创建路径标记
    auto path_marker = createShortPathMarker(path_points_);
    marker_array.markers.push_back(path_marker);

    // 发布标记数组
    marker_pub_->publish(marker_array);
}
```

2.3.3 computeFlatness函数

`computeFlatness`函数用于计算四旋翼的姿态（四元数），使用微分平坦方法：

```
eigen::Quaterniond computeFlatness(const Eigen::Vector3d& pos, double t) {
    // 计算期望的偏航角（简单的绕z轴旋转）
    double yaw = t;
```

```
    // 计算前向速度向量
    Eigen::Vector3d forward_vector(std::cos(yaw), std::sin(yaw), 0.0);

    // 计算向上向量（始终指向z轴正方向）
    Eigen::Vector3d up_vector(0.0, 0.0, 1.0);

    // 计算右向向量（叉乘）
    Eigen::Vector3d right_vector = up_vector.cross(forward_vector);
    right_vector.normalize();

    // 重新计算向前向量（确保正交）
    forward_vector = right_vector.cross(up_vector);
    forward_vector.normalize();

    // 构建旋转矩阵
    Eigen::Matrix3d R;
    R.col(0) = right_vector;
    R.col(1) = forward_vector;
    R.col(2) = up_vector;

    // 转换为四元数
    return Eigen::Quaterniond(R);
}
```

2.3.4 createQuadrotorMarkerArray函数

`createQuadrotorMarkerArray`函数用于创建四旋翼的可视化标记数组，包括机身、臂和螺旋桨：

```
visualization_msgs::MarkerArray createQuadrotorMarkerArray(const QuadState&
state) {
    visualization_msgs::MarkerArray markers;

    // 创建机身标记
    visualization_msgs::Marker body_marker;
    body_marker.header.frame_id = "docs";
    body_marker.header.stamp =
SystemManager::instance().now().secondsSinceEpoch();
    body_marker.ns = "quadrotor";
    body_marker.id = 0;
    body_marker.type = visualization_msgs::Marker::CYLINDER;
    body_marker.action = visualization_msgs::Marker::ADD;
    // 设置位置和方向
    body_marker.pose.position.x = state.pos.x();
    body_marker.pose.position.y = state.pos.y();
    body_marker.pose.position.z = state.pos.z();
    body_marker.pose.orientation.w = state.q.w();
    body_marker.pose.orientation.x = state.q.x();
    body_marker.pose.orientation.y = state.q.y();
    body_marker.pose.orientation.z = state.q.z();
    // 设置大小
    body_marker.scale.x = 0.2;
```

```
    body_marker.scale.y = 0.2;
    body_marker.scale.z = 0.1;
    // 设置颜色
    body_marker.color.r = 0.0;
    body_marker.color.g = 0.0;
    body_marker.color.b = 1.0;
    body_marker.color.a = 1.0;
    // 添加到标记数组
    markers.markers.push_back(body_marker);

    // 创建臂标记（4个）
    // ... 代码略 ...

    // 创建螺旋桨标记（4个）
    // ... 代码略 ...

    return markers;
}
```

2.3.5 createShortPathMarker函数

`createShortPathMarker`函数用于创建短路径的可视化标记：

```
visualization_msgs::Marker createShortPathMarker(const
std::vector<Eigen::Vector3d>& path_points) {
    visualization_msgs::Marker path_marker;
    path_marker.header.frame_id = "docs";
    path_marker.header.stamp =
SystemManager::instance().now().secondsSinceEpoch();
    path_marker.ns = "path";
    path_marker.id = 100;
    path_marker.type = visualization_msgs::Marker::LINE_STRIP;
    path_marker.action = visualization_msgs::Marker::ADD;
    // 设置路径点
    for (const auto& point : path_points) {
        geometry_msgs::Point p;
        p.x = point.x();
        p.y = point.y();
        p.z = point.z();
        path_marker.points.push_back(p);
    }
    // 设置颜色
    path_marker.color.r = 1.0;
    path_marker.color.g = 0.0;
    path_marker.color.b = 0.0;
    path_marker.color.a = 0.8;
    // 设置线宽
    path_marker.scale.x = 0.03;
    // 设置生命周期
    path_marker.lifetime.fromSec(0.1);
```

```
    return path_marker;
}
```

2.4 main函数

`main`函数是程序的入口点，负责初始化和运行示例：

```
int main(int argc, char** argv) {
    // 创建示例对象
    MarkerPublisherExample example;
    // 运行示例
    example.run();
    return 0;
}

void MarkerPublisherExample::run() {
    // 启动消息循环
    SystemManager::instance().spin();
}
```

3. quad_simulator.cpp解析

quad_simulator.cpp是一个演示四旋翼模拟器的示例程序。这个示例展示了如何实现一个简单的四旋翼模拟器，包括状态更新、轨迹生成和里程计消息发布等功能。

3.1 整体结构

示例代码的整体结构如下：

```
// 包含必要的头文件
#include "global_init.h"
#include "node_handle.h"
#include "geometry_msgs.pb.h"
#include <thread>
#include <chrono>
#include <memory>
#include <iostream>
#include <Eigen/Dense>
#include <cmath>

using namespace std::chrono_literals;

// 定义四旋翼状态结构体
struct QuadState {
    double x, y, z;
    double qx, qy, qz, qw;
    double vx, vy, vz;
    double wx, wy, wz;
};
```

```
// 定义四旋翼模拟器类
class QuadSimulator {
public:
    QuadSimulator() : counter_(0) {}

    void run();
private:
    // 私有成员和方法
    int counter_;
    std::shared_ptr<NodeHandle> nh_;
    std::shared_ptr<Publisher<geometry_msgs::Odometry>> odom_pub_;
    std::shared_ptr<Timer> timer_;

    void timerCallback(const TimerEvent& event);
    QuadState computeFlatness(double x, double y, double z,
                              double vx, double vy, double vz,
                              double ax, double ay, double az);
};

// 主函数
int main() {
    // 初始化并运行示例
}
```

3.2 QuadSimulator类

`QuadSimulator`类是示例程序的主要类，用于管理节点、发布者和定时器，并实现四旋翼的模拟功能。

3.2.1 构造函数和初始化

构造函数初始化了计数器，`run()`方法负责初始化系统、创建节点句柄、发布者和定时器：

```
QuadSimulator::QuadSimulator() : counter_(0) {}

void QuadSimulator::run() {
    auto& sys = SystemManager::instance();
    sys.init("quad_simulator");
    std::this_thread::sleep_for(200ms);

    nh_ = std::make_shared<NodeHandle>();
    odom_pub_ = nh_->advertise<geometry_msgs::Odometry>("quad_odometry");

    timer_ = nh_->createTimer(
        0.02,
        std::bind(&QuadSimulator::timerCallback, this,
std::placeholders::_1),
        false
    );

    std::cout << "Quad Simulator Running..." << std::endl;
```

```
    sys.spin();
}
```

3.2.2 timerCallback函数

`timerCallback`函数是定时器的回调函数，用于更新四旋翼状态并发布里程计消息：

```
void QuadSimulator::timerCallback(const TimerEvent& event) {
    double dt = 0.02;
    double radius = 2.0;
    double speed = 2.0;
    double z_amp = 0.2;
    double z_freq = 0.5;

    double t = counter_ * dt;

    // 位置
    double x = radius * cos(speed * t);
    double y = radius * sin(speed * t);
    double z = 0.5 + z_amp * sin(z_freq * t);

    // 速度
    double vx = -speed * radius * sin(speed * t);
    double vy =  speed * radius * cos(speed * t);
    double vz =  z_amp * z_freq * cos(z_freq * t);

    // 加速度
    double ax = -speed*speed*radius*cos(speed*t);
    double ay = -speed*speed*radius*sin(speed*t);
    double az = -z_amp*z_freq*z_freq*sin(z_freq*t);

    // 计算四元数
    QuadState state = computeFlatness(x, y, z, vx, vy, vz, ax, ay, az);

    // 发布Odometry
    geometry_msgs::Odometry odom_msg;
    odom_msg.mutable_pose()->mutable_position()->set_x(state.x);
    odom_msg.mutable_pose()->mutable_position()->set_y(state.y);
    odom_msg.mutable_pose()->mutable_position()->set_z(state.z);
    odom_msg.mutable_pose()->mutable_orientation()->set_x(state.qx);
    odom_msg.mutable_pose()->mutable_orientation()->set_y(state.qy);
    odom_msg.mutable_pose()->mutable_orientation()->set_z(state.qz);
    odom_msg.mutable_pose()->mutable_orientation()->set_w(state.qw);

    odom_msg.mutable_linear_velocity()->set_x(state.vx);
    odom_msg.mutable_linear_velocity()->set_y(state.vy);
    odom_msg.mutable_linear_velocity()->set_z(state.vz);

    odom_msg.mutable_angular_velocity()->set_x(state.wx);
    odom_msg.mutable_angular_velocity()->set_y(state.wy);
    odom_msg.mutable_angular_velocity()->set_z(state.wz);
```

```
    odom_pub_->publish(odom_msg);

    counter_++;
}
```

3.2.3 computeFlatness函数

computeFlatness函数用于使用微分平坦方法计算四旋翼的姿态（四元数）：

```
QuadState computeFlatness(double x, double y, double z,
                          double vx, double vy, double vz,
                          double ax, double ay, double az) {
    QuadState state;
    state.x = x; state.y = y; state.z = z;
    state.vx = vx; state.vy = vy; state.vz = vz;

    Eigen::Vector3d acc(ax, ay, az + 9.81);
    double yaw = std::atan2(vy, vx);
    Eigen::Vector3d Z_b = acc.normalized();
    Eigen::Vector3d X_c(cos(yaw), sin(yaw), 0);
    Eigen::Vector3d Y_b = Z_b.cross(X_c).normalized();
    Eigen::Vector3d X_b = Y_b.cross(Z_b);

    Eigen::Matrix3d R;
    R.col(0) = X_b; R.col(1) = Y_b; R.col(2) = Z_b;

    Eigen::Quaterniond quat(R);
    state.qx = quat.x(); state.qy = quat.y();
    state.qz = quat.z(); state.qw = quat.w();

    state.wx = 0; state.wy = 0; state.wz = 0;

    return state;
}
```

3.2.4 main函数

main函数是程序的入口点，负责创建模拟器对象并运行：

```
int main() {
    try {
        QuadSimulator sim;
        sim.run();
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }
```

```
    return 0;
}
```

4. quad_visualizer.cpp解析

quad_visualizer.cpp是一个演示四旋翼可视化节点的示例程序。这个示例展示了如何订阅四旋翼的里程计消息，并将其在Foxglove Studio中可视化。

4.1 整体结构

示例代码的整体结构如下：

```
// 包含必要的头文件
#include "global_init.h"
#include "marker.pb.h"
#include "node_handle.h"
#include "geometry_msgs.pb.h"
#include <deque>
#include <vector>
#include <iostream>
#include <memory>
#include <Eigen/Dense>
#include <cmath>

// 定义四旋翼可视化器类
class QuadVisualizer {
public:
    void run();
private:
    // 私有成员和方法
    std::shared_ptr<NodeHandle> nh_;
    std::shared_ptr<Publisher<visualization_msgs::MarkerArray>>
marker_pub_;
    std::shared_ptr<Publisher<visualization_msgs::Marker>> short_path_pub_;
    std::shared_ptr<Publisher<visualization_msgs::Marker>>
incremental_path_pub_;
    std::shared_ptr<Subscriber> odom_sub_;

    std::deque<geometry_msgs::Point> short_path_points_;
    std::vector<geometry_msgs::Point> incremental_path_points_;
    geometry_msgs::Point last_point_;
    bool has_last_point_ = false;

    void odomCallback(const std::shared_ptr<geometry_msgs::Odometry>&
odom);
};

// 主函数
int main() {
    // 初始化并运行示例
}
```

4.2 QuadVisualizer类

`QuadVisualizer`类是示例程序的主要类，用于管理节点、发布者和订阅者，并实现四旋翼的可视化功能。

4.2.1 run方法

`run`方法负责初始化系统、创建节点句柄、发布者和订阅者：

```
void QuadVisualizer::run() {
    auto& sys = SystemManager::instance();
    sys.init("quad_visualizer");
    nh_ = std::make_shared<NodeHandle>();

    marker_pub_ = nh_->advertise<visualization_msgs::MarkerArray>
("quad_marker_array");
    short_path_pub_ = nh_->advertise<visualization_msgs::Marker>
("quad_path_short");
    incremental_path_pub_ = nh_->advertise<visualization_msgs::Marker>
("quad_path_incremental");

    odom_sub_ = nh_->subscribe<geometry_msgs::Odometry>(
        "quad_odometry",
        10,
        [this](const std::shared_ptr<geometry_msgs::Odometry>& odom) {
            this->odomCallback(odom);
        }
    );

    std::cout << "Quad Visualizer Running..." << std::endl;
    sys.spin();
}
```

4.2.2 odomCallback函数

`odomCallback`函数是里程计订阅者的回调函数，用于处理接收到的里程计消息并发布可视化标记：

```
void QuadVisualizer::odomCallback(const
std::shared_ptr<geometry_msgs::Odometry>& odom) {
    double x = odom->pose().position().x();
    double y = odom->pose().position().y();
    double z = odom->pose().position().z();
    double qx = odom->pose().orientation().x();
    double qy = odom->pose().orientation().y();
    double qz = odom->pose().orientation().z();
    double qw = odom->pose().orientation().w();

    // 更新轨迹
    geometry_msgs::Point p;
```

```
    p.set_x(x); p.set_y(y); p.set_z(z);
    short_path_points_.push_back(p);
    while (short_path_points_.size() > 150) short_path_points_.pop_front();

    incremental_path_points_.push_back(p);

    // 发布MarkerArray（机身Cube + 四臂 + 螺旋桨）
    visualization_msgs::MarkerArray marker_array;
    int id = 0;

    // --- 机身 ---
    visualization_msgs::Marker body;
    body.set_ns("quadrotor");
    body.set_id(id++);
    body.set_type(visualization_msgs::MarkerType::CUBE);
    body.set_action(visualization_msgs::MarkerAction::ADD);
    body.mutable_scale()->set_x(0.3);
    body.mutable_scale()->set_y(0.3);
    body.mutable_scale()->set_z(0.1);
    body.mutable_pose()->mutable_position()->set_x(x);
    body.mutable_pose()->mutable_position()->set_y(y);
    body.mutable_pose()->mutable_position()->set_z(z);
    body.mutable_pose()->mutable_orientation()->set_x(qx);
    body.mutable_pose()->mutable_orientation()->set_y(qy);
    body.mutable_pose()->mutable_orientation()->set_z(qz);
    body.mutable_pose()->mutable_orientation()->set_w(qw);
    body.mutable_color()->set_r(0.2);
    body.mutable_color()->set_g(0.2);
    body.mutable_color()->set_b(0.8);
    body.mutable_color()->set_a(1.0);
    *marker_array.add_markers() = body;

    // --- 四个臂 ---
    double arm_length = 0.6;
    double arm_radius = 0.03;
    double offsets[4][3] = {
        {arm_length/2, 0.0, 0.02},
        {-arm_length/2, 0.0, 0.02},
        {0.0, arm_length/2, 0.02},
        {0.0, -arm_length/2, 0.02}
    };
    Eigen::Matrix3d R;
    R = Eigen::Quaterniond(qw, qx, qy, qz).toRotationMatrix();
    for (int i = 0; i < 4; i++) {
        visualization_msgs::Marker arm;
        arm.set_ns("quadrotor");
        arm.set_id(id++);
        arm.set_type(visualization_msgs::MarkerType::CYLINDER);
        arm.set_action(visualization_msgs::MarkerAction::ADD);
        arm.mutable_scale()->set_x(arm_radius);
        arm.mutable_scale()->set_y(arm_radius);
        arm.mutable_scale()->set_z(0.02);
        Eigen::Vector3d offset(offsets[i][0], offsets[i][1], offsets[i]
[2]);
```

```
        Eigen::Vector3d pos = R * offset + Eigen::Vector3d(x, y, z);
        arm.mutable_pose()->mutable_position()->set_x(pos.x());
        arm.mutable_pose()->mutable_position()->set_y(pos.y());
        arm.mutable_pose()->mutable_position()->set_z(pos.z());
        arm.mutable_pose()->mutable_orientation()->set_x(qx);
        arm.mutable_pose()->mutable_orientation()->set_y(qy);
        arm.mutable_pose()->mutable_orientation()->set_z(qz);
        arm.mutable_pose()->mutable_orientation()->set_w(qw);
        arm.mutable_color()->set_r(0.8);
        arm.mutable_color()->set_g(0.2);
        arm.mutable_color()->set_b(0.2);
        arm.mutable_color()->set_a(1.0);
        *marker_array.add_markers() = arm;
    }

    // --- 四个螺旋桨 ---
    double prop_radius = 0.3;
    double prop_thick = 0.02;
    for (int i = 0; i < 4; i++) {
        visualization_msgs::Marker prop;
        prop.set_ns("quadrotor");
        prop.set_id(id++);
        prop.set_type(visualization_msgs::MarkerType::CYLINDER);
        prop.set_action(visualization_msgs::MarkerAction::ADD);
        prop.mutable_scale()->set_x(prop_radius);
        prop.mutable_scale()->set_y(prop_radius);
        prop.mutable_scale()->set_z(prop_thick);
        Eigen::Vector3d offset(offsets[i][0], offsets[i][1], offsets[i]
[2]+0.02);
        Eigen::Vector3d pos = R * offset + Eigen::Vector3d(x, y, z);
        prop.mutable_pose()->mutable_position()->set_x(pos.x());
        prop.mutable_pose()->mutable_position()->set_y(pos.y());
        prop.mutable_pose()->mutable_position()->set_z(pos.z());
        prop.mutable_pose()->mutable_orientation()->set_x(qx);
        prop.mutable_pose()->mutable_orientation()->set_y(qy);
        prop.mutable_pose()->mutable_orientation()->set_z(qz);
        prop.mutable_pose()->mutable_orientation()->set_w(qw);
        prop.mutable_color()->set_r(0.2);
        prop.mutable_color()->set_g(0.8);
        prop.mutable_color()->set_b(0.2);
        prop.mutable_color()->set_a(1.0);
        *marker_array.add_markers() = prop;
    }

    marker_pub_->publish(marker_array);

    // --- 短期轨迹 ---
    visualization_msgs::Marker line_short;
    line_short.set_ns("quad_path_short");
    line_short.set_id(0);
    line_short.set_type(visualization_msgs::MarkerType::LINE_STRIP);
    line_short.set_action(visualization_msgs::MarkerAction::ADD);
    line_short.mutable_color()->set_r(1.0);
    line_short.mutable_color()->set_g(0.0);
```

```
        line_short.mutable_color()->set_b(0.0);
        line_short.mutable_color()->set_a(1.0);
        line_short.mutable_scale()->set_x(0.35);
        line_short.set_lifetime(50);
        for (const auto& pt : short_path_points_) {
            geometry_msgs::Point* new_pt = line_short.add_points();
            new_pt->set_x(pt.x());
            new_pt->set_y(pt.y());
            new_pt->set_z(pt.z());
        }
        short_path_pub_->publish(line_short);

        // --- 增量轨迹 ---
        if (has_last_point_) {
            visualization_msgs::Marker line_inc;
            line_inc.set_ns("quad_path_incremental");
            line_inc.set_id(1);
            line_inc.set_type(visualization_msgs::MarkerType::LINE_STRIP);
            line_inc.set_action(visualization_msgs::MarkerAction::ADD);
            line_inc.set_lifetime(-1); // 永久保留
            line_inc.mutable_color()->set_r(0.0);
            line_inc.mutable_color()->set_g(1.0);
            line_inc.mutable_color()->set_b(0.0);
            line_inc.mutable_color()->set_a(1.0);
            line_inc.mutable_scale()->set_x(0.15);

            // 只加两个点 (last_point, current_point)
            geometry_msgs::Point* p1 = line_inc.add_points();
            p1->set_x(last_point_.x());
            p1->set_y(last_point_.y());
            p1->set_z(last_point_.z());

            geometry_msgs::Point* p2 = line_inc.add_points();
            p2->set_x(p.x());
            p2->set_y(p.y());
            p2->set_z(p.z());

            incremental_path_pub_->publish(line_inc);
        }

        // 更新 last_point
        last_point_ = p;
        has_last_point_ = true;
    }
```

4.2.3 main函数

main函数是程序的入口点，负责创建可视化器对象并运行：

```
    int main() {
        try {
```

```
        QuadVisualizer vis;
        vis.run();
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }
    return 0;
}
```

5. foxglove_bridge_tool.cpp解析

foxglove_bridge_tool.cpp是一个演示Foxglove Bridge工具的示例程序。这个示例展示了如何启动Foxglove Bridge，实现与Foxglove Studio的连接。

5.1 整体结构

示例代码的整体结构如下：

```cpp
// 包含必要的头文件
#include "global_init.h"
#include "foxglove_bridge/foxglove_bridge.h"

// 主函数
int main(int argc, char** argv) {
    // 初始化系统
    SystemManager::instance().init("foxglove_bridge_tool");

    // 启动Foxglove Bridge
    auto foxglove_bridge = std::make_shared<FoxgloveBridge>();
    foxglove_bridge->start(8765);  // 在8765端口启动服务

    // 启动消息循环
    SystemManager::instance().spin();

    // 停止Foxglove Bridge
    foxglove_bridge->stop();

    return 0;
}
```

5.2 功能解析

`foxglove_bridge_tool.cpp`示例程序相对简单，主要演示了如何启动和停止Foxglove Bridge服务：

1. 初始化系统
2. 创建FoxgloveBridge对象
3. 在指定端口（8765）启动Foxglove Bridge服务
4. 启动消息循环
5. 当程序退出时，停止Foxglove Bridge服务

6. 如何运行示例代码

6.1 编译示例代码

示例代码通常会在构建系统时自动编译。如果需要单独编译示例代码，可以使用以下命令：

```
cd <path_to_simple_ros>
mkdir build && cd build
cmake ..
make
```

6.2 运行示例代码

编译完成后，可以使用以下命令运行示例代码：

```
# 运行标记发布器示例
./examples/marker_publisher_example

# 运行四旋翼模拟器节点
./examples/quad_simulator_node

# 运行四旋翼可视化节点
./examples/quad_visualizer_node

# 运行Foxglove Bridge工具
./examples/foxglove_bridge_tool
```

6.3 查看可视化结果

运行示例代码后，可以使用Foxglove Studio查看可视化结果：

1. 启动Foxglove Studio
2. 点击"Open Connection"按钮
3. 选择"Foxglove WebSocket"连接类型
4. 输入连接地址：ws://localhost:8765
5. 点击"Connect"按钮
6. 在左侧面板中选择要查看的主题（如visualization_marker）

7. 示例代码的常见用法

7.1 发布可视化标记

使用`marker_publisher_example.cpp`中的方法，可以发布各种类型的可视化标记：

```
// 创建节点句柄
auto nh = std::make_shared<NodeHandle>();
```

```
// 创建发布者
auto marker_pub = nh->advertise<visualization_msgs::MarkerArray>
("visualization_marker");

// 创建标记
visualization_msgs::Marker marker;
marker.header.frame_id = "docs";
marker.header.stamp = SystemManager::instance().now().secondsSinceEpoch();
marker.ns = "my_namespace";
marker.id = 0;
marker.type = visualization_msgs::Marker::SPHERE;
marker.action = visualization_msgs::Marker::ADD;
// 设置位置和大小
marker.pose.position.x = 0.0;
marker.pose.position.y = 0.0;
marker.pose.position.z = 0.0;
marker.scale.x = 0.1;
marker.scale.y = 0.1;
marker.scale.z = 0.1;
// 设置颜色
marker.color.r = 1.0;
marker.color.g = 0.0;
marker.color.b = 0.0;
marker.color.a = 1.0;

// 创建标记数组
visualization_msgs::MarkerArray marker_array;
marker_array.markers.push_back(marker);

// 发布标记数组
marker_pub->publish(marker_array);
```

7.2 订阅和发布消息

使用quad_simulator_node.cpp和quad_visualizer_node.cpp中的方法，可以订阅和发布各种类型的消息：

```
// 创建节点句柄
auto nh = std::make_shared<NodeHandle>();

// 创建发布者
auto pub = nh->advertise<std_msgs::String>("my_topic");

// 创建订阅者
auto sub = nh->subscribe<std_msgs::String>(
    "other_topic", 10, [](const std::shared_ptr<std_msgs::String>& msg) {
        std::cout << "Received message: " << msg->data << std::endl;
    });

// 发布消息
std_msgs::String msg;
```

```
msg.data = "Hello, docs!";
pub->publish(msg);
```

7.3 使用定时器

使用`marker_publisher_example.cpp`和`quad_simulator_node.cpp`中的方法，可以使用定时器定期执行回调函数：

```
// 创建节点句柄
auto nh = std::make_shared<NodeHandle>();

// 创建定时器，每100ms执行一次回调函数
auto timer = nh->createTimer(0.1, [](const TimerEvent& event) {
    std::cout << "Timer callback executed at " <<
event.current_real.secondsSinceEpoch() << std::endl;
});

// 启动一次性定时器，5秒后执行一次回调函数
auto oneshot_timer = nh->createTimer(5.0, [](const TimerEvent& event) {
    std::cout << "Oneshot timer callback executed" << std::endl;
}, true);
```

8. 总结

simple_ros系统提供的示例代码展示了系统的各种功能和用法，包括：

- 发布和订阅消息
- 创建和使用定时器
- 发布可视化标记
- 实现四旋翼模拟器
- 使用Foxglove Bridge进行可视化

通过学习和理解这些示例代码，用户可以快速掌握simple_ros系统的使用方法，并应用到自己的机器人应用中。