想问一下什么是API，具体是什么意思？

都在说API，API到底是什么？

其实日常生活中，我们有很多类似API的场景，比如：

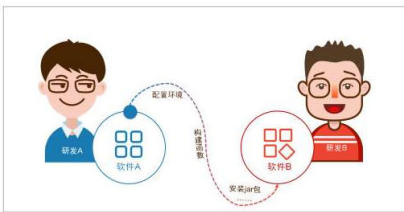
目录

电脑需要调用手机里面的信息，这时候你会拿一根数据线将电脑手机连接起来，电脑和手机上连接数据线的接口就是传说中的API接口。

但比喻到底是比喻，并非本质。想要真正理解API，还得老老实实去理解API的使用场景。

我们不妨把API的诞生过程用一个小故事展示出来：

> 研发人员A开发了软件A，研发人员B正在研发软件B。
>
> 有一天，研发人员B想要调用软件A的部分功能来用，但是他又不想从头看一遍软件A的源码和功能实现过程，怎么办呢？
>
> 研发人员A想了一个好主意：我把软件A里你需要的功能打包好，写成一个函数，你按照我说的流程，把这个函数放在软件B里，就能直接用我的功能了！
>
> 其中，API就是研发人员A说的那个函数。



这就是API的诞生。

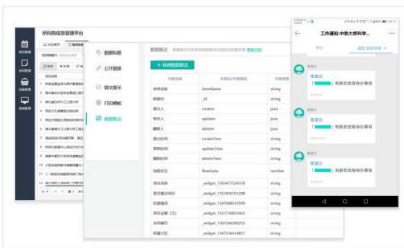
再举个实例辅助你理解：

【中铁大桥科研院】有一个自研的信息平台，用于管理业务数据。

他们面临一个问题——尽管有信息平台，却因为系统的独立性，数据的上传和备份，需要依靠人工在excel里来回操作，效率很低。

由于系统的开发周期长、成本高，桥科院将目光聚焦到现成的功能软件上。

后来通过API将简道云直接插入公司数据库，数据可自动上传至信息平台上并统一展示；再通过webhook把数据推送到服务器，实现自动备份。



API将信息平台与简道云相连

在这一过程中，通过简道云配备的API接口，可以对接外部系统，让桥科院不用开发直接实现了数据自动上传、备份的功能。

那么，这就是API的使用。

常用 API（Application Programming Interface）

应用程序　编程　接口

API（接口）是什么？举个常见的例子，在京东上下单付款之后，商家选用顺丰发货，然后你就可以在京东上实时查看当前的物流信息。京东和顺丰作为两家独立的公司，为什么会在京东上实时看到顺丰的快递信息，这就要用到API，当查看自己的快递信息时，京东利用顺丰提供的API接口，可以实时调取信息呈现在自己的网站上。除此，你也可以在快递100上输入订单号查取到快递信息。只要有合作，或是有允许，别的公司都可以通过顺丰提供的API接口调取到快递信息。既然有多方调用，那提供一个统一的调用规范会方便很多。

API（Application Programming Interface,应用程序编程接口）是一些预先定义的函数，目的是提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，而又无需访问源码，或理解内部工作机制的细节。

*PS：有人说从编程的角度讲，就是封装的函数，避免重复造轮子

封装参数的个数　封装参数(字符串数组)　给节点命名，保证唯一性

```
ros::init(int &argc, char **argv, const std::String &name,
          uint32_t options=0);
```

init函数无返回值

ros::init_options::AnonymousName　节点启动选项

解决节点重复启动的问题

rosrun pkg_name node hello 11451⁴

⇒ argc=3

argv[1]="hello"　argv[2]="11451⁴"

拓扑结构相关

ros::Publisher pub = nh.advertise(const std::string& topic,

uint32_t queue_size, bool latch = false),

(可选) 默认false

如果latch=true，最后一条被发布的一条消息将被保存，当有新的订阅者订阅该消息时，发给它

将latch设置为true的作用

3.1.2 话题对象

函数：

```
ros::NodeHandle nh;

nh.advertise<>("topic",queue_size,latch);
```

作用：

创建发布者对象。

模板：

被发布对象的类型

参数：

参数名称	作用
topic	话题名称
queue_size	队列长度
latch（可选）	默认为false，如果是true，会保存发布方的最后一个消息，并且新的订阅对象连接到发布方时，发布方会将这条消息发布给订阅者

使用：

`latch` 设置为 `true` 的作用：

类似server_client？

以静态地图发布为例，方案1：可以使用固定频率发送地图数据，但是效率低；方案2：可以将地图发布对象的 `latch` 设置为 `true`，并且发布方只发送一次数据，每当订阅者连接时，将地图数据发送给订阅者（只发送一次），从而提高了数据的发送效率。

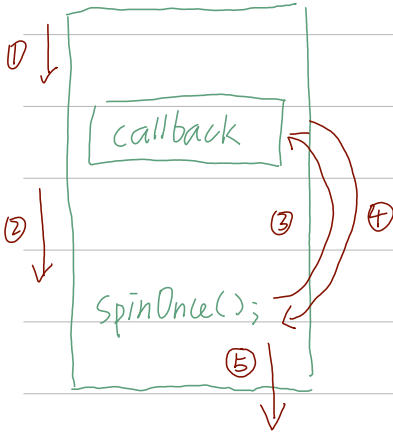
```
nh.advertise<std::String>("House",10,true);
```

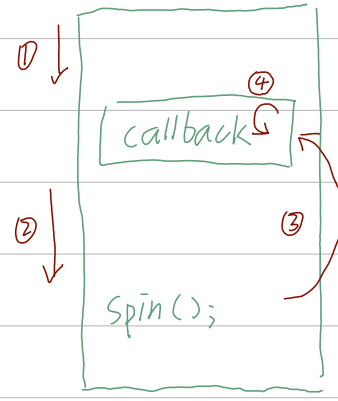
这样将发布方的最后一个数据发送给订阅节点。

循环回调　　一轮回调

回旋函数　ros::spin();　ros::spinOnce();

用于处理回调函数 callback();





1.时刻

获取时刻，或是设置指定时刻：

```
ros::init(argc,argv,"hello_time");
ros::NodeHandle nh;//必须创建句柄，否则时间没有初始化，导致后续API调用失败
ros::Time right_now = ros::Time::now();//将当前时刻封装成对象
ROS_INFO("当前时刻:%.2f",right_now.toSec());//获取距离 1970年01月01日 00:00:00 的秒数
ROS_INFO("当前时刻:%d",right_now.sec);//获取距离 1970年01月01日 00:00:00 的秒数

ros::Time someTime(100,100000000);// 参数1:秒数  参数2:纳秒
ROS_INFO("时刻:%.2f",someTime.toSec()); //100.10
ros::Time someTime2(100.3);//直接传入 double 类型的秒数
ROS_INFO("时刻:%.2f",someTime2.toSec()); //100.30
```

PS：必须创建节点句柄，ros::NodeHandle nh，否则会导致时间API调用失败（再NodeHandle会初始化时间操作）

2.持续时间

设置一个时间区间(间隔):

和 ros::Rate r(Hz) 的区别在哪？返回值？参数？

```
ROS_INFO("当前时刻:%.2f",ros::Time::now().toSec());
ros::Duration du(10);//持续10秒钟，参数是double类型的，以秒为单位
du.sleep();//按照指定的持续时间休眠
ROS_INFO("持续时间:%.2f",du.toSec());//将持续时间换算成秒
ROS_INFO("当前时刻:%.2f",ros::Time::now().toSec());
```

为了方便使用，ROS中提供了时间与时刻的运算:

```
ROS_INFO("时间运算");
ros::Time now = ros::Time::now();
ros::Duration du1(10);
ros::Duration du2(20);
ROS_INFO("当前时刻:%.2f",now.toSec());
//1.time 与 duration 运算
ros::Time after_now = now + du1;
ros::Time before_now = now - du1;
ROS_INFO("当前时刻之后:%.2f",after_now.toSec());
ROS_INFO("当前时刻之前:%.2f",before_now.toSec());

//2.duration 之间相互运算
ros::Duration du3 = du1 + du2;
ros::Duration du4 = du1 - du2;
ROS_INFO("du3 = %.2f",du3.toSec());
ROS_INFO("du4 = %.2f",du4.toSec());
//PS: time 与 time 不可以运算
// ros::Time nn = now + before_now;//异常
```

可以为负数

ros::Duration du = after_now - before_now

ROS_INFO("du=%.2f",du.toSec());

可减不可加

ROS 中内置了专门的定时器，可以实现与 ros::Rate 类似的效果:

```
ros::NodeHandle nh;//必须创建句柄，否则时间没有初始化，导致后续API调用失败

 // ROS 定时器
 /**
* \brief 创建一个定时器，按照指定频率调用回调函数。
*
* \param period 时间间隔
* \param callback 回调函数
* \param oneshot 如果设置为 true，只执行一次回调函数，设置为 false，就循环执行。
* \param autostart 如果为true，返回已经启动的定时器，设置为 false，需要手动启动。
*/
 //Timer createTimer(Duration period, const TimerCallback& callback, bool oneshot = false,
 //                  bool autostart = true) const;

 // ros::Timer timer = nh.createTimer(ros::Duration(0.5),doSomeThing);
ros::Timer timer = nh.createTimer(ros::Duration(0.5),doSomeThing,true);//只执行一次

 // ros::Timer timer = nh.createTimer(ros::Duration(0.5),doSomeThing,false,false);//需要手
 // timer.start();
 ros::spin(); //必须 spin
```

ros::Duration
const ros::TimerCallback & callback
bool
bool
timer start();

定时器的回调函数:

```
void doSomeThing(const ros::TimerEvent &event)
{
    ROS_INFO("-------------");
    ROS_INFO("event:%s",std::to_string(event.current_real.toSec()).c_str());
}
```

ROS_INFO("event:%.2f", event.current_real.toSec());

ros::shutdown(); 关闭节点　作用区别于 ctrl+C 手动关闭，可设置条件：

if(count >=50){ros::shutdown();}

日志函数

3.日志函数

使用示例

```
ROS_DEBUG("hello,DEBUG"); //不会输出
ROS_INFO("hello,INFO"); //默认白色字体
ROS_WARN("Hello,WARN"); //默认黄色字体
ROS_ERROR("hello,ERROR");//默认红色字体
ROS_FATAL("hello,FATAL");//默认红色字体
```

只是调试用，此类消息不会输出到控制台

异常，程序可执行

错误，会影响程序运行

严重错误，将阻止节点运行

0、CMake

CMake中，我们先输入cmake 命令对工程进行分析，生成 makefile 文件；

然后用make命令根据makefile内容编译整个工程。

1、可执行文件

含有mian函数的可以编译成可执行文件；增加可执行文件可以通过添加如下代码实现：

```
add_executable( 程序名 源代码文件 )
```

复制

2、库文件

没有main的函数，即无可执行函数。

```
add_library( 库名 库文件)    #静态库,
add_library( 库名 SHARED 库文件)  #共
```

静态库每次调用都会生成一个副本，共享库只有一个副本。

3、包含头文件

当头文件不在当前目录时，CMake中添加头文件路径，在参数中把所有需要添加的路径，对应的函数叫include。

```
#指定头文件目录
include_directories( "/usr/include
#注意：eigen库只有头文件，没有库文件，因
```

4、头文件和库文件关系

[例子]

假如写了代码：main.cpp func1.h func1.cpp func2.h func2.cpp

cmake处理后：main.cpp->main.exe

func1.cpp->func1.a func2.cpp->func2.a;

两个头文件保留作为指引，将它们链接起来。

最终我们运行程序只需要：main.exe func1.h func1.a func2.h func2.a

由此可见，cmake实际上的作用是隐藏了源代码，并链接了可执行程序和库文件。

只要有头文件和库文件就可以调用库了，中间函数的cpp文件可以省略。

库文件通过头文件向外导出接口。用户通过头文件找到库文件中函数实现的代码从而把这段代码链接到用户程序中去。

5、CMakeLists.txt 结构

假如写了如下代码：

CMakeLists.txt 如下：

```
# 声明要求的 cmake 最低版本
cmake_minimum_required( VERSION 2.8

# 声明一个 cmake 工程
project( HelloSLAM )

# 设置编译模式
set( CMAKE_BUILD_TYPE "Debug" )

# 添加一个可执行程序
add_executable( helloSLAM helloSLAM
# 添加一个可执行程序，同样含有main函数
add_executable( useHello useHello.c

# 添加一个静态库
add_library( hello libHelloSLAM.cpp
# 添加共享库
add_library( hello_shared SHARED li

# 将库文件链接到可执行程序上
target_link_libraries( useHello hel
```

运行结果如下：