



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。

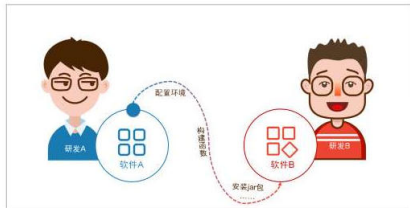
我们不妨把API的诞生过程用一个小故事展示出来：

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。

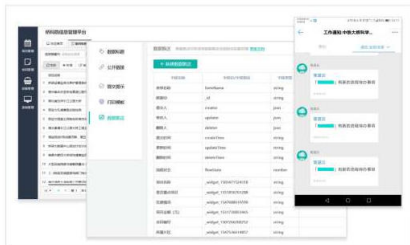
再举个实例辅助你理解：

【中铁大桥科研】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将同道云直接接入公司数据库，数据可自动上传至信息平台并统一展示；再通过webhook<sup>2</sup>把数据推送至服务器，实现自动备份。



API将信息平台与同道云相连

在这一过程中，通过同道云配备的API接口，可以对外部系统，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。

应用程序 编程 接口  
常用 API (Application Programming Interface)

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速信息，这就用到API，当查看自己的快速信息时，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上输入订单号查询到快速信息。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接口调取到快速信息。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，就是对封装的函数，避免重复造轮子

封装参数个数      封装参数 (字符数组)      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  
启动启动的问题

`roslaunch pkg-name node hello 114514`

$\Rightarrow \text{argc} = 3$

`argv[1] = "hello" argv[2] = "114514"`

↑

拓扑结构相关



**基本术语**

1. 节点

节点就是网络单元。网络单元是网络系统中的各种数据交换设备、数据通信控制设备和数据终端设备。

节点分为：终端点。它的作用是发送网络的连接。它通过通信线路接收和传递信息。

访问节点。它是信息交换的源点和目标。

2. 链路

链路是两个节点间的通信。链路分“物理链路”和“逻辑链路”两种。前者是指实际存在的通信线路。后者是指逻辑上起作用的网络通路。链路容量是指每个链路在单位时间内可提供的最大信息量。

3. 通路

通路是从发出信息的节点到接收信息的节点之间的一条节点和链路。也就是说，它是一系列按通信顺序建立起来的节点和链路的通路。<sup>[1]</sup>

**含义**

网络拓扑结构是由网络节点设备和通信介质组成的网络结构图。网络拓扑定义了各种计算机、打印机、网络设备和其他设备的连接方式。换句话说，网络拓扑描述了物理和网络设备的物理以及数据传输时所使用的路径。网络拓扑会在很大程度上影响网络附加的工作。

网络拓扑包括物理拓扑和逻辑拓扑。物理拓扑是指物理结构上各种设备和传输介质的布局。物理拓扑通常有总线型、星型、环型、树型、网状型等几种。<sup>[2]</sup>

**常见网络逻辑拓扑结构**

**星型结构**

星型结构是以一个中心节点为处理系统，各种类型终端用户均须与中心节点有物理链路直接相连。

星型结构的优点结构简单、使用容易、控制和管理。其缺点是需集中控制、主节点负荷过重、可靠性低、通信线路利用率低。<sup>[3]</sup>

**总线型结构**

总线型结构是数据通信采用的一种方式。它把所有的人网计算机的输入到一条通信线上，为防止信号冲突，一般在通信线路两端设置收发信机。

总线型结构的优点是数据集中管理、结构简单、使用容易。其缺点是：每个节点只能靠一个收发信机与总线通信，网络性能较差、网络管理较为复杂、通信线路利用率低、在总线上出现一个信息源时，会占用整个总线的正常时间，目前在局域网中多采用此种结构。<sup>[4]</sup>

**环型结构**

环型结构是将各信息源计算机或通信线路连接成一个闭合的环。

环型网络是一个闭合的环型结构。每个节点都直接连接到环上，通过一个接口设备和收发信机连接到环上。在环上安装时，每个节点都按顺序排列，像环上节点的排列。数据通信按顺序进行，对每个节点来说环上信息是双向流动的。可以设置多种数据通信的传输。传输速率和容量都很大。在环型网络上出现任何故障，都会影响整个网络的正常工作。目前在局域网中多采用此种结构。<sup>[5]</sup>

**树型结构**

星型网络拓扑结构的一种扩展形式就是星型扩展。每个主机与网络用户的数据均为星型结构的连接形式。然而，当出现主机故障时，网络扩展的树型结构扩展。

树型网络是星型网络中控制式网络。与星型相比，它的通信距离长、成本较低、节点易于扩展、维护费用也比较方便。但缺点是：节点及其连接的设备，在一点出现故障或链路故障都会使网络受到影响。

星型网络：只适用于总线。星型网络有唯一的信号。比如没有电报线路的情况下，电话的布线就可以采用这种形式。<sup>[6]</sup>

**网状结构**

网状网络分为全连接网状和不完全连接网状两种。全连接网状，每一个节点都直接连接到其他所有节点。但不全连接网状，网状中只有一部分是连接起来的。它们之间的通信，依靠路由节点。这种网络的特点是节点冗余度高，链路和节点冗余度大大减少。网络故障不会影响整个网络的正常工作。可扩展。网络扩展和主机接入比较灵活。简单。但此种网络及其复杂。使用简单。网络结构复杂。广域网中一般用不完全连接网状网络的结构。<sup>[7]</sup>

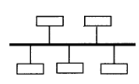
**混合型拓扑**

就是两种或两种以上拓扑结构的组合使用。

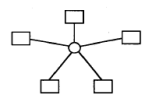
优点：可以克服网络的缺点和缺点。缺点：网络复杂度高。<sup>[8]</sup>

**蜂窝网络拓扑**

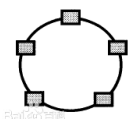
蜂窝网络拓扑是移动通信中常用的结构。它可以分为蜂窝状（蜂窝、星型、环型）和蜂窝状多边形蜂窝状。是一种无线路、通信网络、移动通信、无线网。<sup>[9]</sup>



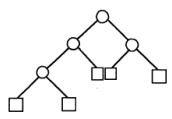
总线型



星型



环型



树型

```
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, 最后一条被发布的消息将被保存, 当有新的订阅者订阅该消息时, 发送它

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);
```

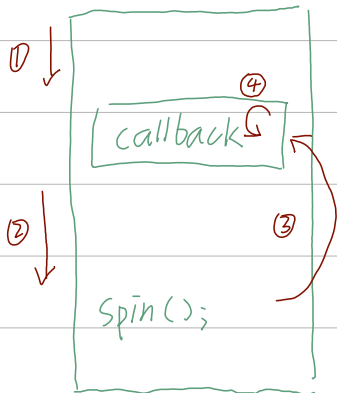
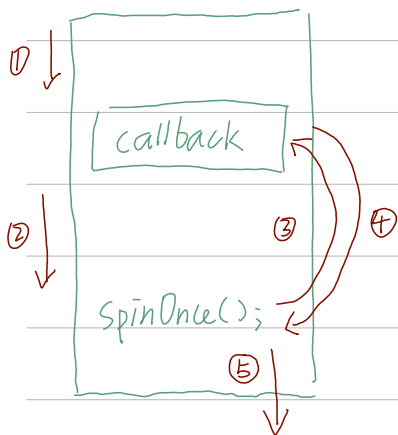
这样将发布方的最后一个数据发送给订阅节点。

循环回调

一轮回调

回调函数 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  
// 逗号 无()   
ros::Time someTime(1000100000000); // 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()); // 将持续时间转换成秒
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
```

Copy

```
// 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 ros::Duration
 * \param callback 回调函数 const ros::TimerCallback & callback
 * \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。 bool
 * \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。 bool
 */
// 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i
// timer.start();

ros::spin(); // 必须 spin
```

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());
}
```

Copy

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
```

只是测试用，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main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 复制 )
```

2、库文件

没有main的函数，即无可执行函数。

```
1 add_library( 库名 库文件)           #静态库，
2 add_library( 库名 SHARED 库文件)    #动态库
```

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。

```
1 #指定头文件目录
2 include_directories( "/usr/include
3 #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 and 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：

```
1 # 声明要求的 cmake 最低版本
2 cmake_minimum_required( VERSION 2.8
3
4 # 声明一个 cmake 工程
5 project( HelloSLAM )
6
7 # 设置编译模式
8 set( CMAKE_BUILD_TYPE "Debug" )
9
10 # 添加一个可执行程序
11 add_executable( helloSLAM helloSLAM
12 # 添加一个可执行程序，同样含有main函数
13 add_executable( useHello useHello.c
14
15 # 添加一个静态库
16 add_library( hello libHelloSLAM.cpp
17 # 添加共享库
18 add_library( hello_shared SHARED li
19
20 # 将库文件链接到可执行程序上
21 target_link_libraries( useHello hel
```

运行结果如下：

