进阶课程㉝丨Apollo ROS原理—2

知识点

敲黑板，本文需要学习的知识点有

去中心化　网络拓扑

鲁棒性　数据类型

耦合　校验

在ROS系统中，从数据的发布到订阅节点之间需要进行数据的拷贝。在数据量很大的情况下，很显然这会影响数据的传输效率。所以Apollo项目对于ROS第一个改造就是通过共享内存来减少数据拷贝，以提升通信性能。

目前ROS仅适用于Apollo 3.0之前的版本，最新代码及功能还请参照Apollo 3.5及5.0版本。

以下，ENJOY

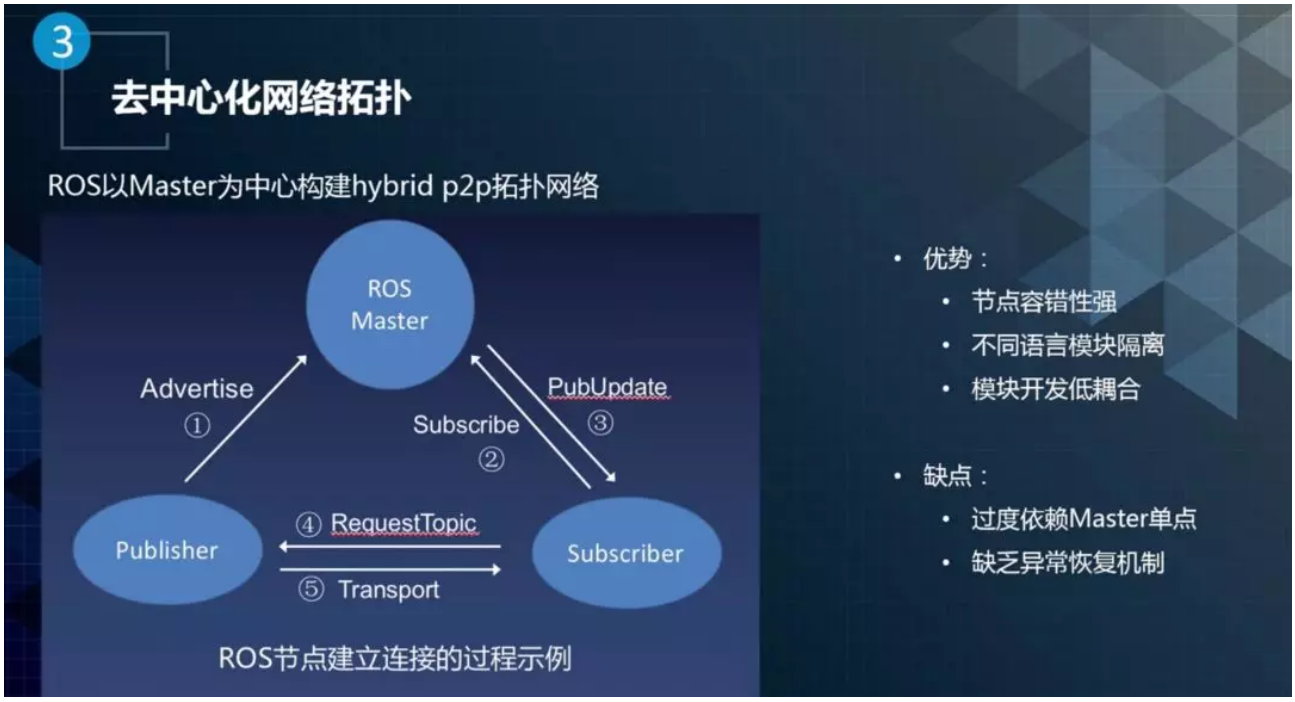
01

Apollo ROS对ROS的改进

02

去中心化网络拓扑

去中心化网络拓扑的原因

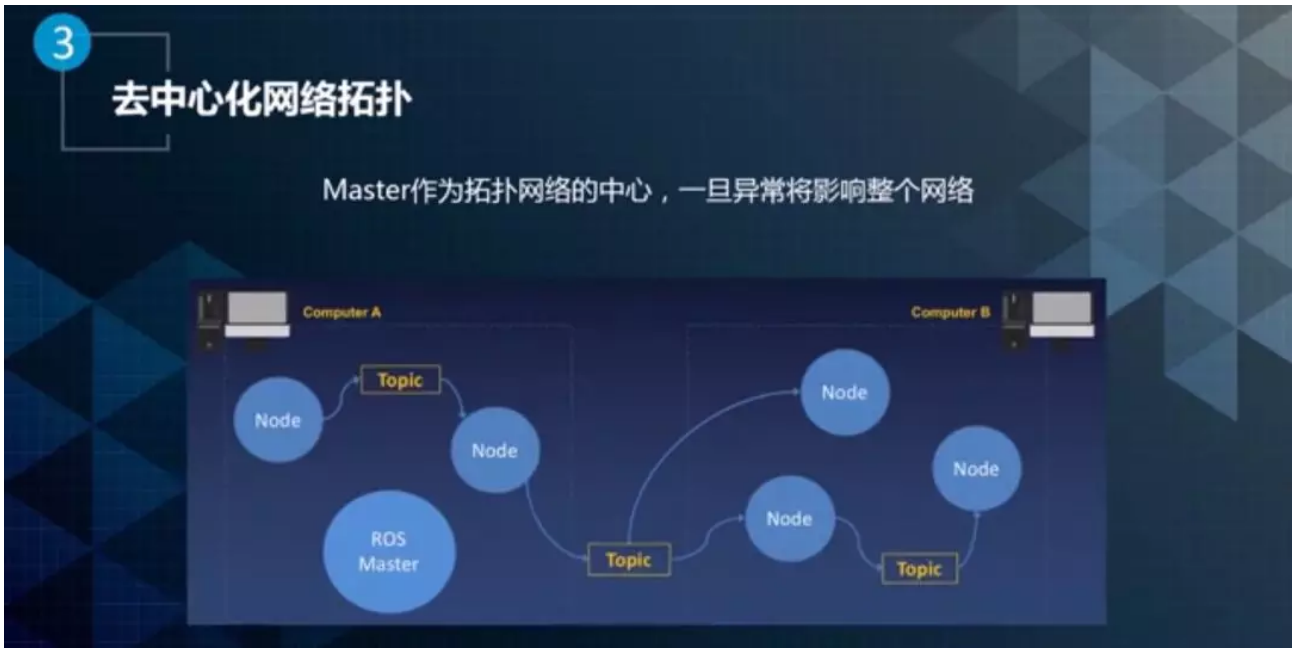


ROS是以Rosmaster节点管理器建立起来的一个P2P拓扑网络，这种拓扑网络有很明显的优势，如下：

1. 节点之间相互独立，容错性比较强。
2. 每个模块用不同的语言去开发，对其它的模块是透明的，其它模块不用关注和它通信的数据节点以及模块使用什么语言来开发。
3. 模块开发之间是比较解耦合的，你只要定义好使用Topic/Service/Param的信息，然后按照这个格式去开发自己的模块。

但是也有两个比较明显的缺点。第一，节点之间的通信过于依赖Rosmaster单点。两个节点进行通信的链路过程，大概分为五部。第一步：发送节点去向Master注册一个发送节点。第二步：接收节点去向Master注册一个接收节点。第三步：Master向接收节点发送一个已有发送节点的一个信息拓扑。第四步：接收节点拿到这个拓扑信息之后去向发送节点请求建立一个tcp连接。第五步：在发送节点和接收节点建立一个P2P的单点拓扑连接之后就持续不断的向接收节点发送信息。整个过程中对Master依赖包含三步，在建立实际通信之后，对Master的依赖可能会降低很多，但是在建立之前是比较依赖Master节点的。

第二，ROS没有提供一种异常恢复机制。如果某一个节点挂掉，尤其是Master节点挂掉，其它的节点却不知道发生了这样的行为，还会认为整个系统运行仍然处在正确的状态中。比如发送节点里面有一些Service或者Param相关的请求，它还是会照常去发请求或者是设置这个参数信息，这样就会产生一些不可控的行为。



Master单点在多机的方案里，这个单机单点的不足就会更加凸显。比如现在很多自动驾驶厂商所采用的比较主流的Nvidia Drive PX2板卡，它就包括两个系统：一个**主系统**，一个是**冗余备份系统即容错系统**。如果使用ROS通信在PX2上进行部署，Master只能起在一个节点上。如上图所示：左侧是它的主系统，右侧是它的冗余备份系统。当主系统里面Rosmaster宕机之后，备份系统里面的节点其实并不知道Rosmaster已经处于一个宕机的状态，那么备份系统就起不到其目的和意义了。因为此时整个系统处于一个功能不完整的状态，所以就失去了冗余备份的意义。

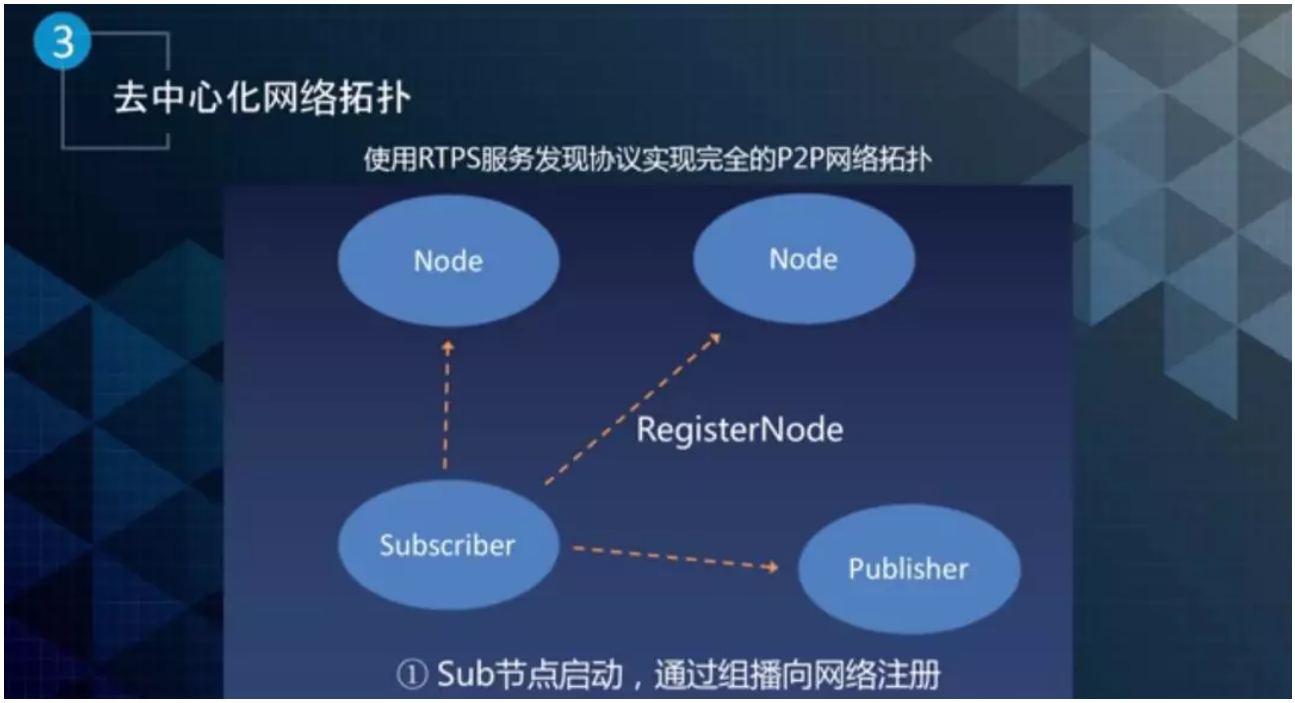
使用RTPS服务发现协议实现完全的P2P网络拓扑

Apollo ROS进行了比较大的改造：先把这个中心化的网络拓扑给去掉，然后建立了一个点对点之间的一个复杂网络拓扑，主要是**使用RTPS服务发现协议去完成P2P网络拓扑**。如上图所示：右侧是ROS Node的一个框架图。左下角是引入RTPS服务相关的一些功能。其它部分是ROS Node现有的一些功能。Ros Node是分层级式的结构，最上层是Handler，Handler提供节点和ROS整个通信的基本交互的句柄。下一层左侧和右侧定义了这个节点发送和订阅的Channel信息。再下一层是Middleware，Middleware是这个节点和其它节点进行通信的时候去完成链路的建立和数据的发送。

接下来的一层左下角RTPS是新引入的一个功能。改造之后的ROS Node架构，当一个节点被启动的时候，它会通过RTPS向所有的节点发送信息：现在有一个新的节点要加入到这个拓扑网络。当它离开的时候，也会发送消息告诉所有的节点：现在这个节点要退出。以前这些功能都是通过Rosmaster来完成的。

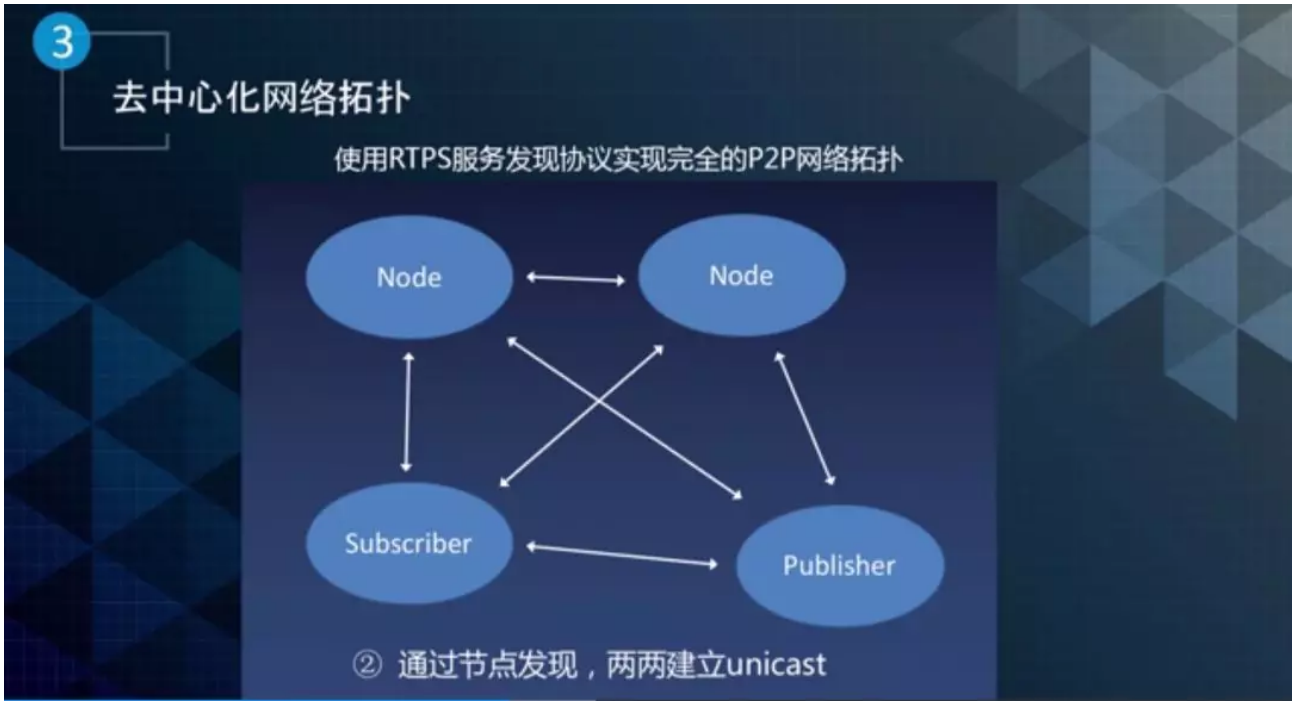
下面通过几张图来描述：节点建立连接和通讯的一个主要流程。

第一步：**Sub节点启动，通过组播向网络注册。**



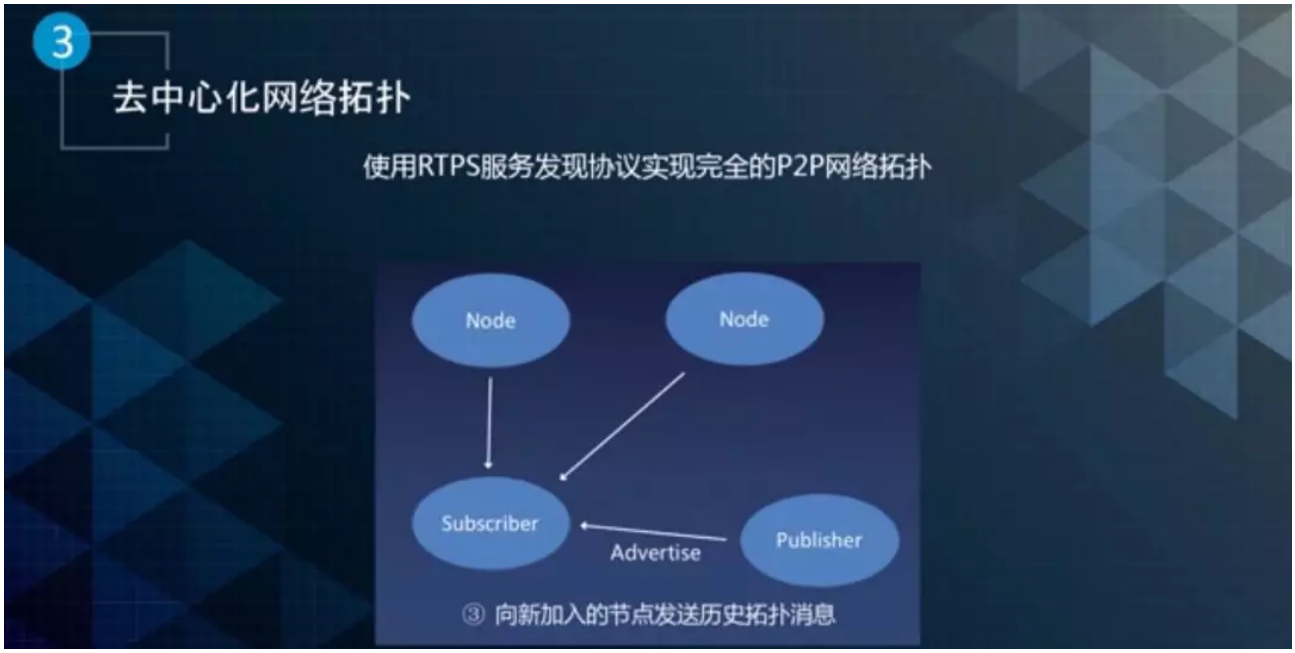
订阅节点在启动的时候，它会向当前这个域里面所有的节点发送信息：现在有一个新的节点要启动。

第二步：**通过节点发现，两两建立unicast。**



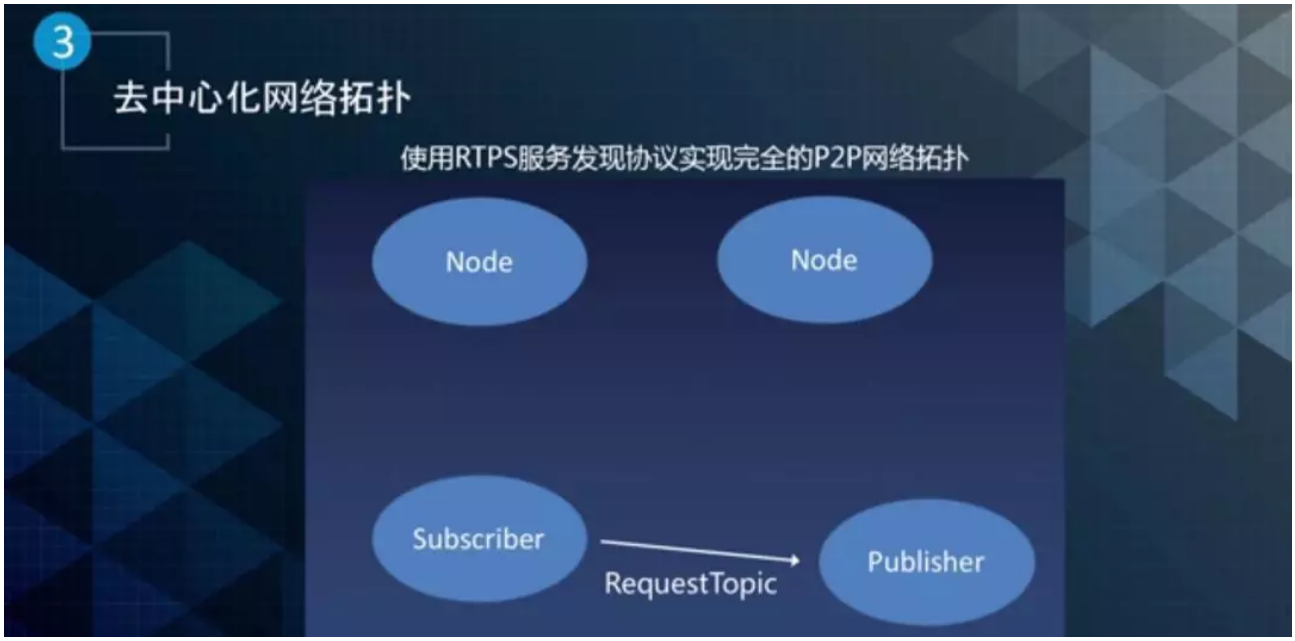
所有的节点在接收到新加入这个节点发生拓扑信息变更之后，会和新加入这个节点分别建立两两连接关系。

第三步：**向新加入的节点发送它们已经有拓扑信息。**



所有已经存在的节点会向新加入的节点发送它们已经有拓扑信息，也就是在新节点加入之前每个节点其实是维护了它和其它所有节点的一个连接关系，这个连接关系发送给接收节点，供接收节点去更新自己的网络拓扑结构。

第四步：**收发双方建立连接，开始通信。**



当新加入节点接到所有节点发送出来的历史拓扑信息之后，它会根据它自己注册的实际消息内容去决定和哪些节点建立实际的通信连接。如上图所示：新加入节点只和右下角的一个节点之间有拓扑关系，它除了维护所有的节点给它发送出的整个网络拓扑信息之外，同时会和发送节点建立点对点的通信连接。

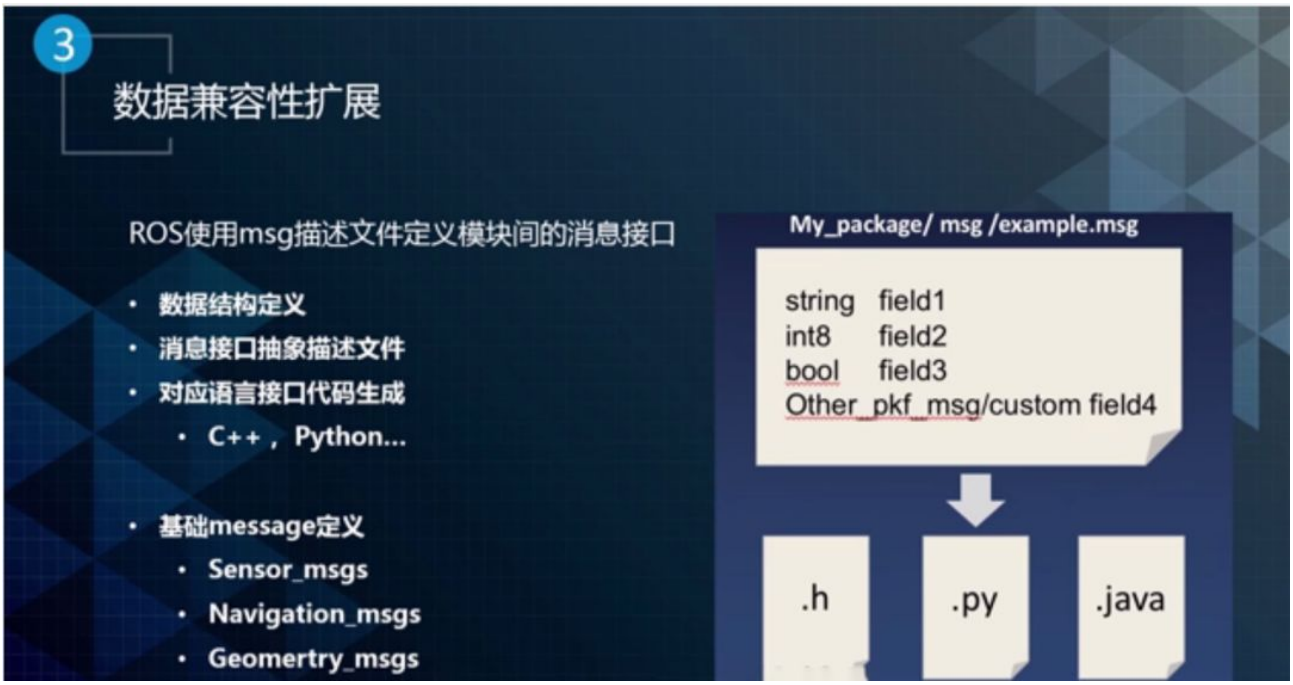
通过RTPS拓扑发现方式，Apollo ROS去除了对Rosmaster这一个单点的依赖，从而提升整个系统的鲁棒性。这个修改完全是对ROS底层的修改，用户基于原生ROS代码写的节点程序，到Apollo ROS是完全兼容的一个迁移即开发者不需要去改动任何的接口，就可以直接使用RTPS网络拓扑这种新的关系建立。

03 数据兼容性扩展

数据兼容性扩展的原因

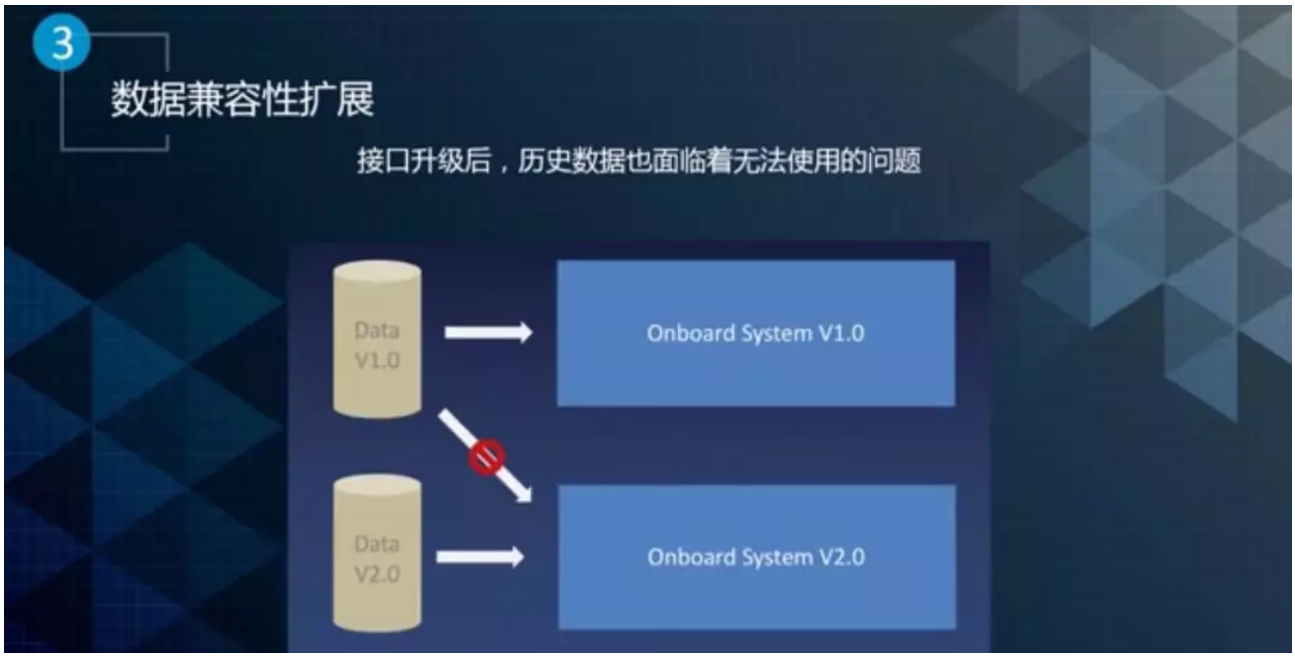
原生ROS基于Message的订阅发布消息模型。发送者和接收者在进行实际通讯之前需要进行消息格式定义，其包含字段：基础的数据类型或者复杂的数据类型。在它们进行消息通信的时候，才可以有选择性的去建立通信连接和数据实际发送。如果有一个节点订阅的消息类型不是Channel预先指定的

消息类型，这种通信连接是建立不起来的。或是强制指定一个节点去订阅某一种类型的Channel信息，但是它的实际回调函数里却写的是另外一种消息类型，这种编译可能在实际运行的时候就会报相关的一些错误。



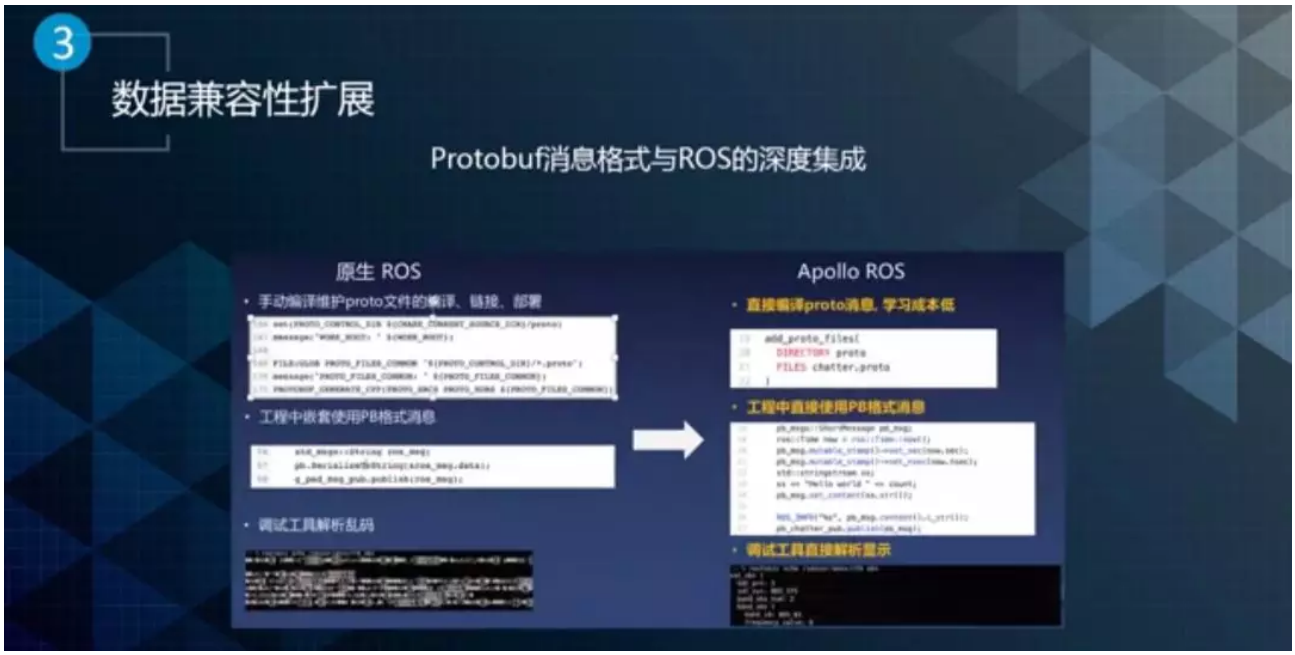
Message是两个节点进行消息通信的抽象描述文件。这个描述文件提前定义好两个节点之间进行消息通信的基本数据类型。ROS采用这种方式是因为能比较大概率地对两个节点之间进行解耦合，同时两个节点之间也是跨语言的，即不需要关注两个节点是用什么语言写的，都可以通过这种描述文件去进行实际的消息通信。通过Message通信的时候接、收节点在接受到信息之后，会进行MD5的校验、验证这个消息是否符合它的预先订阅，或者是在使用消息之后才会去进行消息的回调处理。

但是ROS基于Message这种通讯方式有很多的缺点。它最大限度解放两个点之点的一个耦合关系也带来了一些问题。比如Message接口升级，不同版本之间的兼容是需要做大量的适配工作。再如某个模块进行升级，之前所录制的一些实验数据，在进行回放的时候就会产生不匹配的现象。



历史数据在接口升级之后也面临着无法转化和兼容使用的问题。

深度整合Protobuf功能，实现数据兼容性扩展



Apollo ROS实践里面引入了一种新的消息描述的格式去实现很好的向后兼容即Protobuf。只需要在使用的过程中，定义好必须的字段或者是一些新增的字段，新增的字段我们可以使用Optional属性去描述。在进行模块升级或者是模块之间的消息接口升级的时候，下游模块其实不需要关注新增字段对它来说会造成什么样的影响。如果它要去使用这个字段的话才需要去进行一定程度的适配。如果它的程序不使用这个新增的字段，就不需要做任何的修改。

上图是原生ROS和Apollo ROS对数据兼容支持的对比。

为了做好数据兼容，在原生ROS里面，开发者使用了一个trick：将Proto文件序列化成一个字符串信息放到Message信息里面，完成消息的向后兼容。比起Apollo ROS这个方式有两个明显的缺点：

1. 它增加了一次数据序列化和反序列化。并把Proto序列化信息压到Message里面，增加了两次额外的数据Copy。
2. 如果想实时调试信息，通过Rostopic echo打印出来Message里面那个序列化的字母串，若是采用Wrapper的方式，则这个字符串信息在屏幕上就会是一堆乱码。

Apollo ROS 为了满足数据兼容，深度整合了Protobuf的功能。用户可以直接定义Proto的字段信息，同时信息传递的过程不需要再进行额外的Message的数据转化。另外，在使用调试工具的时候，通过Rostopic echo可以看出原始消息传递的实际展示。