


CommonAPI 使用说明文档



IPC 专栏收录该内容

0 订阅

4 篇文章

订阅专栏

一、概述

1.CommonAPI C++是什么？

CommonAPI C++是用于开发分布式应用程序的标准C++ API规范，该分布式应用程序通过中间件进行进程间通信。

2.CommonAPI C++的目的是什么？

CommonAPI C++依靠FrancaIDL来描述静态接口，根据通信协议部署参数，一起组建完整的实例依赖关系模型。目的是封装通信协议和相邻的中间件，使应用程序的C++接口独立于底层IPC堆栈。

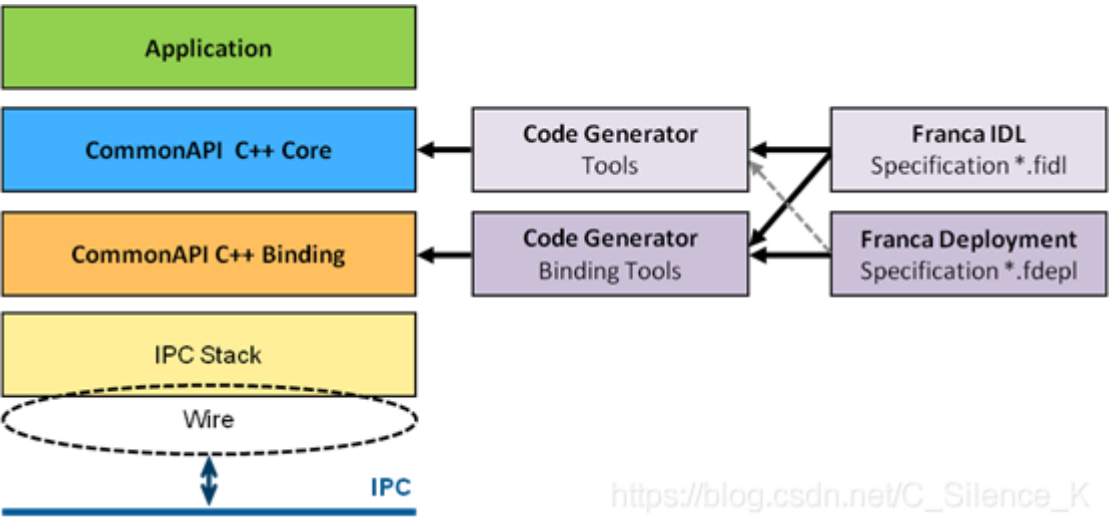
也就是IPC Common API允许针对开发的应用程序（即使用C++的客户端和服务端）可以与不同的IPC后端链接（someip，或D-Bus），而无需更改应用程序代码。因此，为使用特定IPC X（例如someip）的系统开发的组件可以轻松部署到另一个使用IPC Y（例如D-Bus）的系统，只需要交换IPC Common API后端（someip或D-Bus），而无需重新编译应用程序代码。

实际的接口定义将使用Franca IDL创建（*.fild文件）。

而各项部署根据部署文件定义（*.fdepl文件）。

3.CommonAPI的组成原理

（1）基本原理如下图所示：

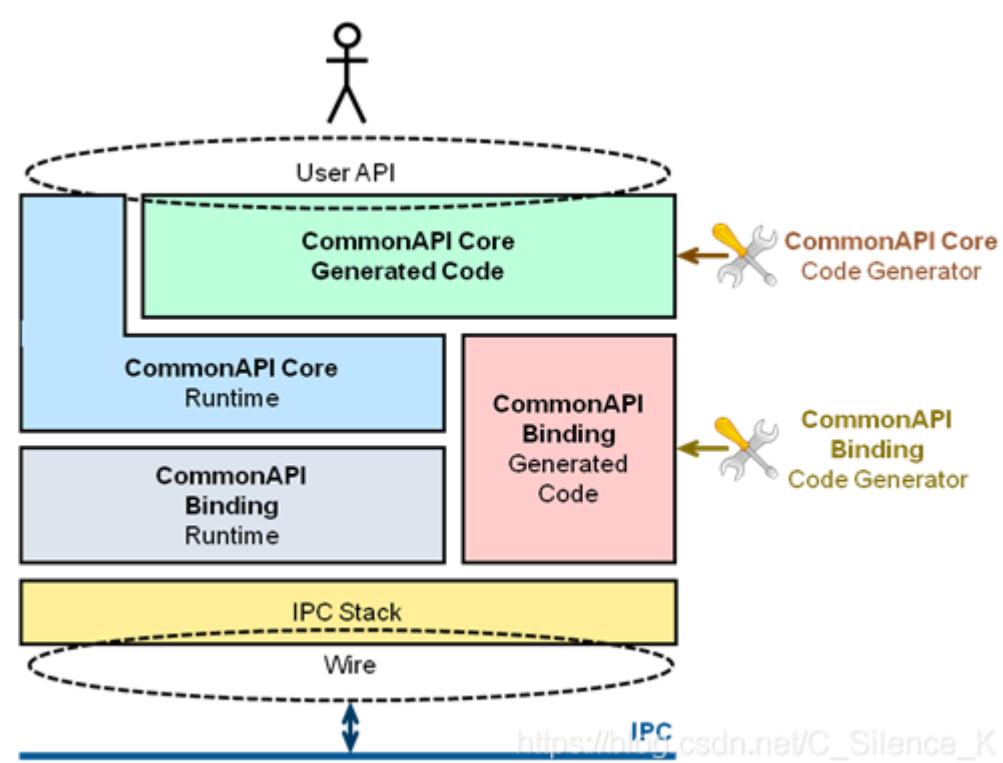


- CommonAPI C++分为独立于中间件的部分（CommonAPI Core，仅仅指CommonAPI接口）和特定于中间件的部分（CommonAPI Binding，用于选择使用的IPC协议的代码）。
- CommonAPI将接口描述语言Franca IDL用于接口规范（逻辑接口规范，*.fild文件）。Franca IDL的代码生成的是CommonAPI的组合部分。主要指逻辑接口的变量部分，那是接口的一部分，它取决于Franca IDL文件中的规范（数据类型，数组，枚举和就基础知识，包括属性，方法，回调，错误处理，广播）。
- CommonAPI C++ binding的代码生成器需要特定于中间件的参数（部署参数，例如String数据类型的编码/解码格式）。这些参数在Franca部署文件（*.fdepl）中定义。主要独立于接口规范。

（2）CommonAPI进一步划分

CommonAPI的用户API分为两部分：

- 基于FrancaIDL的生成部分，其中包含与FrancaIDL文件的类型，属性和方法有关的API函数。也就是根据*.fidl文件生成的API函数。
- “公共”部分（ Runtime API ），其中包含用于加载运行时环境，创建proxy等的API函数。也就是根据*.fidl文件与*.fdepl文件生成的代码中所包含的头文件所链接的库（ CommonAPI lib files + CommonAPI someip/d-bus lib files ）。

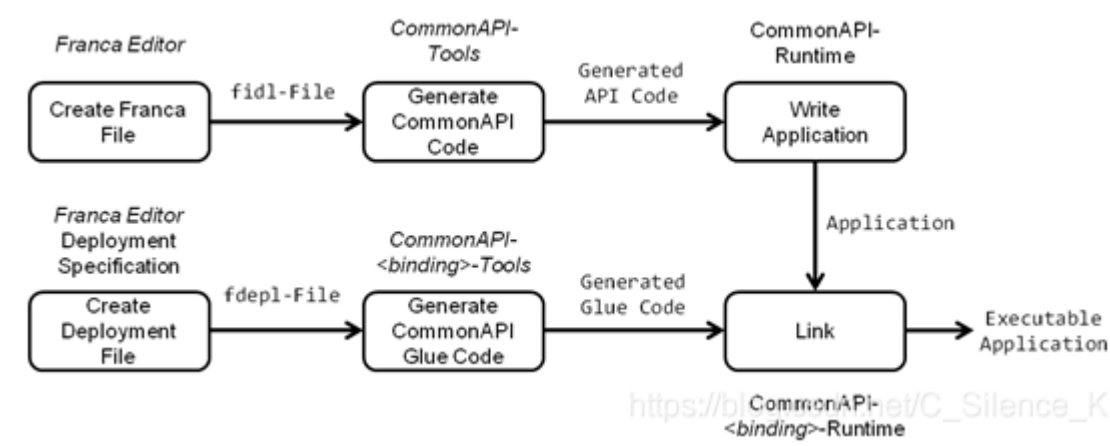


这张图片更详细地显示了CommonAPI C++的元素如何组合在一起。注意：

- 用户API的绝大多数是CommonAPI的生成部分。
- CommonAPI Core和IPC堆栈之间没有直接关系。
- CommonAPI binding的生成代码具有CommonAPI其他部分的所有接口。

4.CommonAPI基本的工作流程

CommonAPI需要基本的工作流程才能创建可执行的应用程序。



应用程序开发人员的工作流程如下：

- 创建具有方法和属性的接口规范的FrancaIDL文件。
- 通过启动CommonAPI代码生成器为客户端和服务端生成代码。
- 通过实现所生成框架中的方法来实现服务；或设置为默认实现。
- 在应用程序中通过创建proxy并使用proxy调用这些方法来实现客户端。

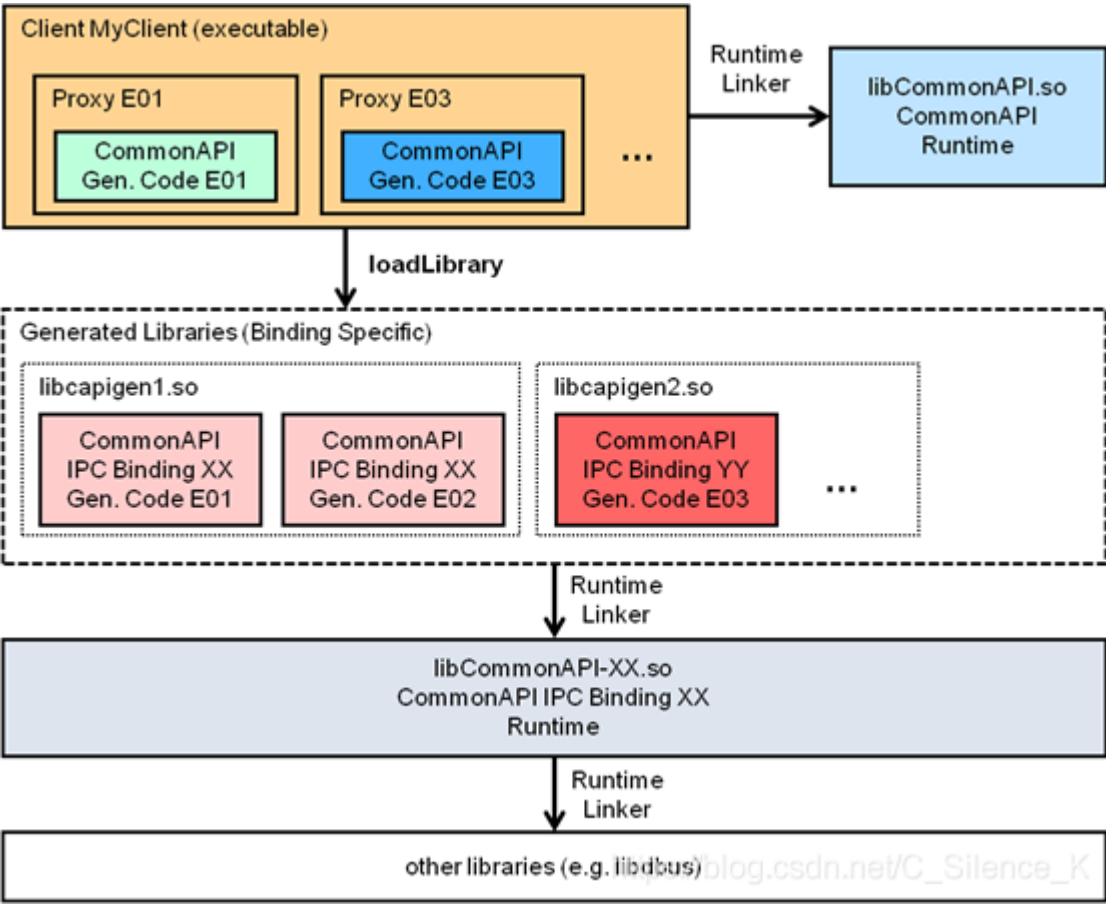
具体的工作流程参考例子：[https://at.projects.genivi.org/wiki/pages/viewpage.action?pageId=5472316#CommonAPIC++D-Busin10minutes\(fromscratch\)-Step1:Preparation/Prerequisites](https://at.projects.genivi.org/wiki/pages/viewpage.action?pageId=5472316#CommonAPIC++D-Busin10minutes(fromscratch)-Step1:Preparation/Prerequisites)

5.构建CommonAPI项目库

CommonAPI可执行文件通常由6部分组成：

- 应用程序代码本身是由开发人员手动编写的；
- 生成的CommonAPI(绑定独立)代码。根据*.fidl文件生成的代码。
在客户端，这段代码包含proxy函数，由应用程序调用;在服务中，它包含生成的函数，这些函数必须由开发人员手动实现(也可以生成默认实现)。
- CommonAPI运行时库。
- 生成的绑定特定代码(所谓的粘合代码)。根据*.fdepl文件生成的代码。
- 绑定的运行时库。
- 使用的中间件的通用库(例如libdbus/vsomeip)。

可以将这6部分划分为共享库或静态库并将它们集中到目标平台上。



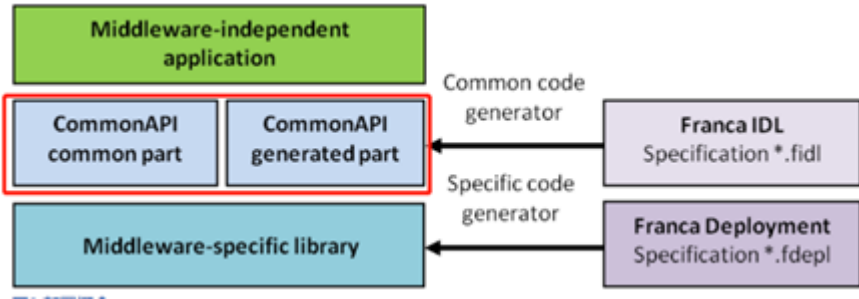
现在将在创建proxy的确切时间加载粘合代码库。通过CommonAPI配置文件可以找到正确的库，该配置文件包含CommonAPI地址和粘合代码库之间的关联。如果配置文件中没有条目，则使用默认设置。

胶水代码库是binding特定的；这意味着所需的运行库由运行时链接程序自动加载。

二、CommonAPI C++规范

1.CommonAPI的基础部分

Common API可以分为两部分：



- 第一部分是由Common API代码生成器生成的基于Franca的部分，也就是根据*.fidl文件生成的部分。那是接口的一部分，它是根据Franca IDL文件中的规范生成的，指数数据类型，数组，枚举和接口等基础知识，包含属性，方法，回调，错误处理，广播等方面。

- 第二个固定部分，是基于CommonAPI Runtime的功能，且独立于接口的规范。它们主要与基于中间件（someip/d-bus）提供的运行时环境有关。此外，此部分包含常见的类型定义和基类。也就是CommonAPI源代码库与所选中间件（someip/d-bus）的源代码库，供代码生成器生成的代码调用。
- 通用API运行时是从其开始所有类加载的基类。通用API运行时访问配置文件，以确定应加载哪个特定的中间件运行时库。中间件库是静态链接的，或者是作为共享库（文件扩展名.so）提供的，因此可用动态加载它们。

2.Franca 基础部分（*.fidl文件）

```
package commonapi.method

interface Methods {
    version {major 1 minor 0}

    method foo {
        in {
            Int32 x1
            String x2
        }
        out {
            Int32 y1
            String y2
        }
        error {
            stdErrorTypeEnum
        }
    }

    broadcast myStatus {
        out {
            Int32 myCurrentValue
        }
    }

    enumeration stdErrorTypeEnum {
        NO_FAULT
        MY_FAULT
    }
}
```

```
package commonapi.unions

interface Unions {
    version {major 0 minor 0}

    attribute CommonTypes.SettingsUnion u
    attribute CommonTypes.SettingsStruct x
}

typeCollection CommonTypes {
    version {major 0 minor 0}

    typedef MyTypedef is Int32

    enumeration MyEnum {
        DEFAULT
        ON
        OFF
    }

    union SettingsUnion {
        MyTypedef id
        MyEnum status
        UInt8 channel
        String name
    }

    struct SettingsStructMyTypedef extends SettingsStruct {
        MyTypedef id
    }
}
```

2.1*.fild文件的基本构成

类别1	类别2	说明	举例
package		限定包名，与命名空间（namespace）有关，必须包含	Package commonapi.examples
	interface	接口，提供一个接口名称，用于包含接口函数等，当定义接口时需要	Interface Methods
	version	版本号，同命名空间（namespace）有关，必须包含	version{major 0 minor 1}
	method	方法，定义供应用程序所调用的输入输出接口函数，具有in，out，error三种参数，这三种参数都是可选的，可以只有in和out，或只有in等多种组合。	method foo { in { Int32 x1 String x2 } out { Int32 y1 String y2} error{ stdErrorTypeEnum } }
	broadcast	广播，定义供应用程序调用的广播类接口函数；只有out参数。	broadcast myStatus { out { Int32 myCurrentValue } }
	attribute	属性，对各种属性进行定义，可用于获取或设置属性值	attribute Int32 x
	typedef	定义类型别名	typedef MyType is Int32
	array	数组	array myArray of UInt16
	enumeration	枚举类型	enumeration stdErrorTypeEnum { NO_FAULT MY_FAULT }
	union	联合类型	union MyUnion {UInt32 MyUIntString MyString}
	struct	结构体	struct a2Struct { Int32 a Boolean b Double d}

类别1	类别2	说明	举例
	map	一种STL关联容器，以一种键值-对的机制存储数据	map MyMap {UInt32 to String}
typeCollection		类型集合，用户根据自己的需要添加的数据类型的集合，各种数据结构在同一个文件中进行扩展等	typeCollection CommonTypes
	version	同interface::version	同interface::version
	typedef	同interface::typedef	同interface::typedef
	array	同interface::array	同interface::array
	enumeration	同interface::enumeration	同interface::enumeration
	union	同interface::union	同interface::union
	struct	同interface::struct	同interface::struct
	map	同interface::map	同interface::map

2.2命名空间

Common API基本函数的名称空间是Common API；Common API应用程序的名称空间取决于Franca IDL中定义接口规范和接口版本的限定包名。

在名称空间的开头添加版本的主要原因是，我们将接口的名称以及包路径（完全限定名称）作为一个单元，这是在Franca文件中指定的。此名称空间不应被中间的附加内容破坏。

Franca IDL

```
package commonapi.helloWorlds

interface HelloWorld {
    version {major 0 minor 1}
}
```

CommonAPI C++

```
namespace v0 {
namespace commonapi {
namespace helloWorlds {

} // namespace helloWorlds
} // namespace commonapi
} // namespace v0

namespace CommonAPI {
}
```

namespace 在每个文件的开头都有，代表这个文件所在的src-gen下面的路径，每个namespace代表一层目录；

如果接口没有版本，则省略名称空间中与版本有关的部分（例如图中的namespace v0）。

1		Version {major 1 minor 0} -- namespace v1 { }
2		Version {major 0 minor 1} -- namespace v0 { }
3		Version {major 0 minor 0} -- 无与版本相关的namespce

2.3接口interface

对应Franca接口名称（称为interfacename），将interfacename生成为一个提供getInterfaceName()方法和getInterfaceVersion()方法的类。版本映射到CommoAPI::Version。

Franca IDL


```
package commonapi.helloWorlds

interface HelloWorld {
    version {major 0 minor 1}
}
```

CommonAPI C++

```
namespace v0 {
namespace commonapi {
namespace helloWorlds {

class HelloWorld {
public:
    virtual ~HelloWorld() {}

    static inline const char* getInterface();
    static inline CommonAPI::Version getInterfaceVersion();
};

const char* HelloWorld::getInterface() {
    return ("commonapi.helloWorlds.HelloWorld");
}

CommonAPI::Version HelloWorld::getInterfaceVersion() {
    return CommonAPI::Version(0, 1);
}

} // namespace helloWorlds
} // namespace commonapi
} // namespace v0
```

函数实现

https://blog.csdn.net/C_Silence_K

版本结构的规范是CommonAPI的一部分：

```
namespace CommonAPI {

struct Version {
    ...Version() = default;
    ...Version(const uint32_t &majorValue, const uint32_t &minorValue)
    ...    : Major(majorValue), Minor(minorValue) {
    ...}

    ...uint32_t Major;
    ...uint32_t Minor;
};

} // namespace CommonAPI
```

2.4类型集合Type Collection

在Franca中，可以将一组用户定义的类型定义为type collection。类型集合的名称称为typecollectionname，可以为空。CommonAPI为空类型集合使用默认名称Anonymous。CommonAPI代码生成器生成头文件Anonymous.hpp，并为类型集合创建C++ struct。如果类型集合的名称不为空（例如：CommonTypes），则CommonAPI代码生成器生成头文件CommonTypes.hpp。

生成的函数类似于接口interface。

Franca IDL

```
typeCollection CommonTypes {
    version {major 1 minor 0}

    typedef MyTypedef is Int32
}
```

CommonAPI C++

```
namespace v1 {
namespace commonapi {
namespace helloWorlds {

struct CommonTypes {
... typedef int32_t MyTypeDef;

static inline const char* getTypeCollectionName() {
... static const char* typeCollectionName = "commonapi.helloWorlds.CommonTypes";
... return typeCollectionName;
}

inline CommonAPI::Version getTypeCollectionVersion() {
... return CommonAPI::Version(1, 0);
}

}; // struct CommonTypes
} // namespace helloWorlds
} // namespace commonapi
} // namespace v1
```

https://blog.csdn.net/C_Silence_K

注意：在内部Franca模型中，类型集合是接口的基类。类型集合也可以有一个版本。在这种情况下，名称空间就像生成的版本名称一样被扩展。

2.5方法Method

方法具有in和out参数，并且可以返回可选的应用程序错误error。如果指定了附加标志fireAndForget，则不允许使用out参数，它指示既不返回值也不表示调用状态。没有fireAndForget标志的方可能返回error，可以在Franca IDL中将其指定为枚举。

对于没有fireAndForget标志的方法，提供了一个附加的返回值CallStatus，它被定义为枚举：
在CallStatus定义了呼叫的传输层的结果，即它返回：

```
namespace CommonAPI {
enum class CallStatus {
... SUCCESS,
... OUT_OF_MEMORY,
... NOT_AVAILABLE,
... CONNECTION_FAILED,
... REMOTE_ERROR,
... UNKNOWN,
... INVALID_VALUE,
... SUBSCRIPTION_REFUSED
};
} // namespace CommonAPI
```

不鼓励在没有定义超时的情况下发送任何方法调用。可以通过将可选参数CallInfo传递给方法调用或在CommonAPI部署文件中配置超时。

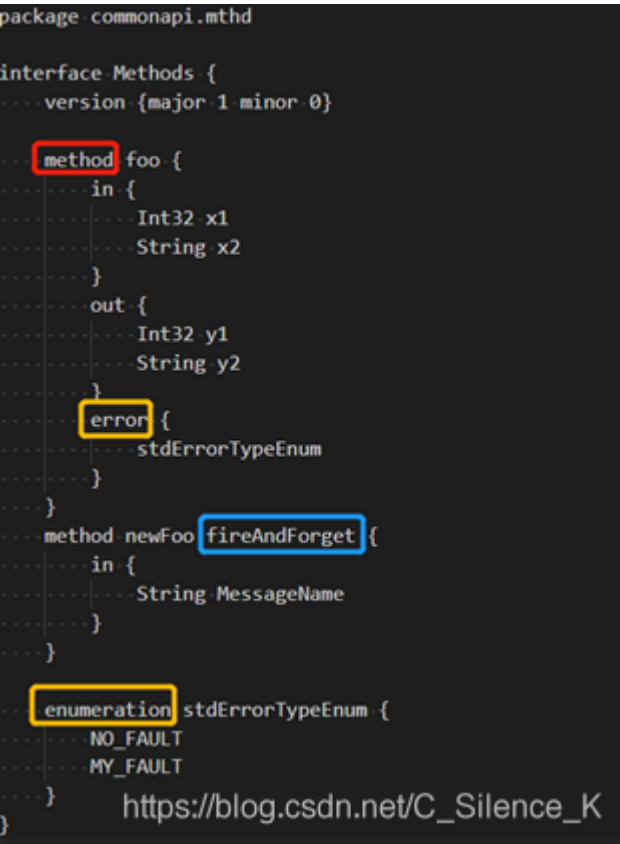
```
struct COMMONAPI_EXPORT CallInfo {
CallInfo()
... : timeout(DEFAULT_SEND_TIMEOUT_MS), sender_(0) {}
CallInfo(timeout_t _timeout)
... : timeout(_timeout), sender_(0) {}
CallInfo(timeout_t _timeout, Sender_t _sender)
... : timeout(_timeout), sender_(_sender) {}

Timeout_t timeout;
Sender_t sender;
};
```

该结构包含一个附加成员sender_，该成员可用于标识此函数的调用方。

标志	参数	说明
None	In + out + error(可选)	提供一个附加的返回值CallStatus
fireAndForget	in	不允许使用out参数，指示既不返回也不表示调用状态

Franca IDL



Method CommonAPI C++

Proxy

返回类型	函数名称	参数
virtual void	foo	const int32_t &_x1, const std::string &_x2, CommonAPI::CallStatus &_internalCallStatus, Methods::fooError &_error, int32_t &_y1, std::string &_y2, const CommonAPI::CallInfo *_info = nullptr
virtual std::futureCommonAPI::CallStatus	fooAsync	const int32_t &_x1, const std::string &_x2, fooAsyncCallback _callback = nullptr, const CommonAPI::CallInfo *_info = nullptr
virtual void	newFoo	const std::string &_MessageName, CommonAPI::CallStatus &_internalCallStatus

在Franca IDL中，方法的异步或同步调用之间没有区别。CommonAPI将同时提供两者。API的用户可以决定他调用哪个变体。含有fireAndForget标志的method，只有同步调用，没有异步调用。

在proxy端生成的函数调用：

- 所有const参数都是该方法的输入参数。
- 所有非const参数将被返回值填充。
- CallStatus将在方法返回时被填满，并指明其中之一

Method CommonAPI C++

Stub

返回类型	函数名称	参数
virtual void	foo	const std::shared_ptrCommonAPI::ClientId _client, int32_t _x1, std::string _x2, fooReply_t _reply
virtual void	newFoo	const std::shared_ptrCommonAPI::ClientId _client, std::string _MessageName

在stub端，生成的函数是生成的stub的一部分，这些功能是纯虚函数的。这意味着必须提供一个实现。此实现的框架可以由代码生成器生成。函数调用的返回值包装在一个函数对象中：

```
1 | typedef std::function<void (Methods::fooError _error, int32_t _y1, std::string _y2)>
```

这允许它将此对象传递给其他函数，以便在stub端实现异步行为。

在stub端，传递了ClientId类型的附加参数。ClientId标识向stub发送调用的客户端。它用于在stub中标识调用

者，并且应该由中间件添加，可以使用==操作符进行比较。

2.6广播Broadcast

广播只能有out参数。对于广播，可以定义一个附加标志selective。该标志指示该消息不应该发送给所有注册的参与者，而是该服务进行选择，表示只有选定的客户端才能在广播中注册。

标志	参数	说明
None	Out	表示发送给所有注册的参与者
Selective	Out	该标志表示只有服务选定的客户端才能在广播中注册

Franca IDL

```
package commonapi.mthd

interface Methods {
  ... version {major 1 minor 0}

  ... broadcast myStatus {
    ... out {
      ... Int32 myCurrentValue
    }
  }

  ... broadcast statusSpecial selective {
    ... out {
      ... Int32 NewCurrentValue
    }
  }
}
```

https://blog.csdn.net/C_Silence_K

Broadcast CommonAPI C++

Proxy

返回类型	函数名称	参数
virtual MyStatusEvent &	getMyStatusEvent	
virtual StatusSpecialSelectiveEvent &	getStatusSpecialSelectiveEvent	

这些方法返回一个事件的包装类，该事件提供对广播MyStatus的访问。包装类提供订阅和取消订阅的方法。Private属性delegate_用于将函数调用转发到特定的绑定（也就是用于与someip协议关联）。

Broadcast CommonAPI C++

Stub

返回类型	函数名称	参数
virtual void	fireMyStatusEvent	const int32_t &_myCurrentVal
virtual void	fireStatusSpecialSelective	const int32_t &_NewCurrentV std::shared_ptrCommonAPI::C = nullptr
virtual std::shared_ptrCommonAPI::ClientIdList const	getSubscribersForStatusSpecialSelective	
virtual void	onStatusSpecialSelectiveSubScriptonChanged	conststd::shared_ptrCommon. const CommonAPI::SelectiveBroadc _event

返回类型	函数名称	参数
virtual bool	onStatusSpecialSelectiveSubscriptionRequest	const std::shared_ptrCommor

2.7属性Attributes

接口的属性由名称和类型定义。另外，属性的规范可以具有两个标志：

- noSubscriptions
- readonly

标志的组合有四种可能：

标志	说明
none	没有附加任何标志的标准属性，默认允许所有内容
readonly	只读属性
noSubscriptions	不可观察的属性
readonly onSubscriptions	不可观察和不可写的属性

可观察的属性提供了一个ChangedEvent，可用于订阅对该属性的更新。此事件与所有其他事件完全一样。头文件Attribute.h中定义了这四种类型的每种属性的模板类。

Franca IDL

```
package commonapi.attr
interface Attributes {
    version {major 1 minor 0}
    attribute Int32 x
    attribute Int32 y readonly
    attribute Int32 w noSubscriptions
    attribute Int32 z readonly noSubscriptions
}
```

CommonAPI C++

Proxy

返回类型	函数名称	参数
virtual XAttribute&	getXAttribute	

仅考虑属性时，stub端属性的CommonAPI如上，该get函数必须由应用程序实现。

Stub

返回类型	函数名称	参数
virtual const int32_t &	getXAttribute	const std::shared_ptrCommonAPI::ClientId _client

此外，CommonAPI定义了必要的回调来处理与IDL描述中为interface定义的属性相关的远程设置事件。对于每个属性，在类AttributesStubRemoteEvent中定义了两个回调：

- 一个验证回调，允许验证请求的值并防止设置，例如无效的值
- 在属性值更改后执行本地工作的操作回调

StubRemoteEvent

返回类型	函数名称	参数	说明
virtual bool	onRemoteSetXAttribute	const std::shared_ptrCommonAPI::ClientId _client, int32_t _value	验证回调
virtual void	onRemoteXAttributeChanged		操作回调

类AttributesStubAdapter提供了一个用于发送广播和可观察属性的更改通知的API，这个API可以在设置属性时调用：

StubAdapter

返回类型	函数名称	参数	说明
virtual void	fireXAttributeChanged	const int32_t& x	用于发送广播和可观察属性的更改通知

CommonAPI代码生成器生成的stub的默认实现将属性定义为stub类的私有属性。可以通过getter和setter函数从stub实现中访问此属性。此外，用于stub实现的API提供了一些回调：

StubDefault

返回类型	函数名称	参数	说明
virtual const int32_t&	getXAttribute		获取属性值
virtual const int32_t&	getXAttribute	const std::shared_ptrCommonAPI::ClientId _client	获取属性值
virtual void	setXAttribute	int32_t _value	在stub实现时更改属性值
virtual void	setXAttribute	const std::shared_ptrCommonAPI::ClientId _client, int32_t _value	在stub实现时更改属性值
virtual bool	trySetXAttribute	int32_t _value	从客户端更改给定值
virtual bool	validateXAttributeRequestedValue	const int32_t & _value	阻止设置属性的回调函数
virtual void	onRemoteXAttributeChanged		通知该属性已更改的回调函数

CommonAPI提供了属性接口的基本实现和扩展机制。所谓扩展机制，就是因为根据应用程序要求属性个数的不同，而存在的一种通用方案，其中包括单个扩展，以便为属性提供任何其他功能（属性扩展）。这将防止开发人员突然添加属性，而缓存不足的情况。
扩展的基类定义在AttributeExtension.hpp中。

2.8事件events

事件为远程触发的动作提供了一个异步接口。这涵盖了FrancaIDL中的广播，方法和属性的更改，事件每个proxy还提供了-一个可用性事件，可用于通知proxy状态。事件提供了订阅和取消订阅的方法，该方法允许注册和注销回调。

事件类的公共接口的相关部分如下：

```
template<typename... Arguments_>
class Event {
public:
    typedef std::tuple<Arguments_...> ArgumentsTuple;
    typedef std::function<void(const Arguments &...)> Listener;
    typedef uint32_t Subscription;
    typedef std::set<Subscription> SubscriptionsSet;
    typedef std::function<void(const CallStatus)> ErrorListener;
    typedef std::tuple<Listener, ErrorListener> Listeners;
    typedef std::map<Subscription, Listeners> ListenersMap;

    /** ...
    Event() : nextSubscription_(0) {};

    /** ...
    Subscription subscribe(Listener listener, ErrorListener errorListener = nullptr);

    /** ...
    void unsubscribe(Subscription subscription);

    virtual ~Event() {}

protected: ...
private: ...
};
```

订阅后，调用listener进行侦听，然后在出现新事件（例如，属性已更改）的任何时候被调用。

2.9数据类型

2.9.1基本类型

CommonAPI使用的整数数据类型在stdint.h中定义。

Franca Type Name	CommonAPI C++ Type	Notes
UInt8	uint8_t	unsigned 8-bit integer(range 0...255)
Int8	int8_t	signed 8-bit integer(range -128...127)
UInt16	uint16_t	unsigned 16-bit integer(range 0...65535)
Int16	int16_t	signed 16-bit integer(range -32768...32767)
UInt32	uint32_t	unsigned 32-bit integer(range 0...4294967295)
Int32	int32_t	signed 32-bit integer(range -2147483648...2147473647)
UInt64	uint64_t	unsigned 64-bit integer
Int64	int64_t	signed 64-bit integer
Boolean	bool	boolean value, which can take one of two values: false or true
Float	float	Floating point number(4 bytes, range +/-3.4e+/-38, ~7 digits).
Double	double	double precision floating point number(8 bytes, range+/-1.7e+/-308, ~15 digits).
String	std::string	character string.
ByteBuffer	std::vector<uint8_t>	buffer of bytes(aka BLOB).

Franca只有一种字符串数据类型String，并且如有必要，可以通过部署模型（也就是*.fdepl文件）指定线路格式/编码。

```
method sayHello {
  in {
    String name
  }
  out {
    String message
  }
}
```

```
method sayHello {
  SomeIpMethodID = 33000
  SomeIpReliable = true

  in {
    name {
      SomeIpStringEncoding = utf16le
    }
  }
}
```

2.9.2数组Arrays

Franca数组类型（可以以显式和隐式两种方法表示）映射到std::vector。虽然可以使用typedef 将Franca IDL中给出的名称显式定义为数组类型，但隐式版本将仅在需要是生成std::vector。

Franca IDL

```
array uint8_array of UInt8
```

CommonAPI C++

```
typedef std::vector<uint8_t> uint8_array;
```

2.9.3结构体Structures

Franca struct类型映射到C++ struct类型。

Structures映射到从CommonAPI::Struct继承的struct。CommonAPI::Struct将结构化数据保存在tuple中。生成的类为结构成员提供getter和setter方法。

Franca IDL

```
struct a2Struct {
  Int32 a
  Boolean b
  Double d
}
```

CommonAPI C++

```
struct a2Struct : CommonAPI::Struct<int32_t, bool, double> {
  a2Struct() {
    std::get<0>(values_) = false;
  }
  a2Struct(const int32_t &a, const bool &b, const double &d)
  {
    std::get<0>(values_) = a;
    std::get<1>(values_) = b;
    std::get<2>(values_) = d;
  }
  inline const int32_t &getA() const { return std::get<0>(values_); }
  inline void setA(const int32_t &value) { std::get<0>(values_) = value; }
  inline const bool &getB() const { return std::get<1>(values_); }
  inline void setB(const bool &value) { std::get<1>(values_) = value; }
  inline const double &getD() const { return std::get<2>(values_); }
  inline void setD(const double &value) { std::get<2>(values_) = value; }
  inline bool operator==(const a2Struct &other) const {
    return (getA() == other.getA() && getB() == other.getB() && getD() == other.getD());
  }
  inline bool operator!=(const a2Struct &other) const {
    return !((*this) == other);
  }
};
```

https://blog.csdn.net/C_Silence_K

2.9.4枚举Enumerations

Franca枚举将映射到从基类继承的C++结构CommonAPI::Enumeration。默认情况下，Enum支持的数据类型和连接格式为uint32_t。如果需要，可以通过CommonAPI部署文件*.fdepl文件（枚举支持类型）指定连接格式。

Franca IDL

```
enumeration EN {  
    DEFAULT  
    NEW  
}
```

CommonAPI C++

```
struct EN : CommonAPI::Enumeration<int32_t> {  
    enum Literal : int32_t {  
        DEFAULT = 0,  
        NEW = 1  
    };  
  
    EN()  
    EN(Literal _literal)  
  
    inline bool validate() const {  
        switch (value_) {  
            case static_cast<int32_t>(Literal::DEFAULT):  
            case static_cast<int32_t>(Literal::NEW):  
                return true;  
            default:  
                return false;  
        }  
    }  
  
    inline bool operator==(const EN &other) const { return (value_ == _other.value_); }  
    inline bool operator!=(const EN &other) const { return (value_ != _other.value_); }  
    inline bool operator<=(const EN &other) const { return (value_ <= _other.value_); }  
    inline bool operator>=(const EN &other) const { return (value_ >= _other.value_); }  
    inline bool operator<(const EN &other) const { return (value_ < _other.value_); }  
    inline bool operator>(const EN &other) const { return (value_ > _other.value_); }  
  
    inline bool operator==(const Literal &value) const { return (value_ == static_cast<int32_t>(_value)); }  
    inline bool operator!=(const Literal &value) const { return (value_ != static_cast<int32_t>(_value)); }  
    inline bool operator<=(const Literal &value) const { return (value_ <= static_cast<int32_t>(_value)); }  
    inline bool operator>=(const Literal &value) const { return (value_ >= static_cast<int32_t>(_value)); }  
    inline bool operator<(const Literal &value) const { return (value_ < static_cast<int32_t>(_value)); }  
    inline bool operator>(const Literal &value) const { return (value_ > static_cast<int32_t>(_value)); }  
};
```

2.9.5Map

出于效率原因，Franca映射的Common API数据类型为std::unordered_map<K, V>。

Franca IDL

```
map MyMap {  
    UInt32 to String  
}
```

CommonAPI C++

```
typedef std::unordered_map<uint32_t, std::string> MyMap;
```

2.9.6联合Union

Franca 中定义的联合类型被实现为CommonAPI通用模板化C++ variant类的类型定义。

Franca IDL

```
union MyUnion {  
    UInt32 MyUInt  
    String MyString  
}
```

CommonAPI C++

```
typedef CommonAPI::Variant<uint32_t, std::string> MyUnion;
```

2.9.7类型别名Type Aliases

Franca typedef映射到C++ typedef。

Franca IDL

```
typedef MyTypedef is Int32
```

CommonAPI C++

```
typedef int32_t MyTypedef;
```

3.CommonAPI的部署方式（*.fdepl文件）

1. 定义独立于中间件（vSomeIp/D-Bus）的C++ API的一个问题是，需要针对API的各个部分使用不同的配置参数，这部分需要取决于中间件。例如，参数，数组或字符串的最大长度等。
2. Franca IDL可以根据中间件或特定于平台的部署模型（*.fdepl文件）中使用的中间件来指定部署参数。
3. 一个明确的目标是，针对Common API编写的应用程序可以与不同的Common API IPC后端链接，而无需更改应用程序代码。
4. 因此，有一个重要的隐性限制：Franca IDL（**.fidl文件）中定义的接口只与CommonAPI以及用户调用相关。专用于IPC后端的部署模型（*.fdepl）不得影响所生成的API。但是允许使用非特定的部署模型。

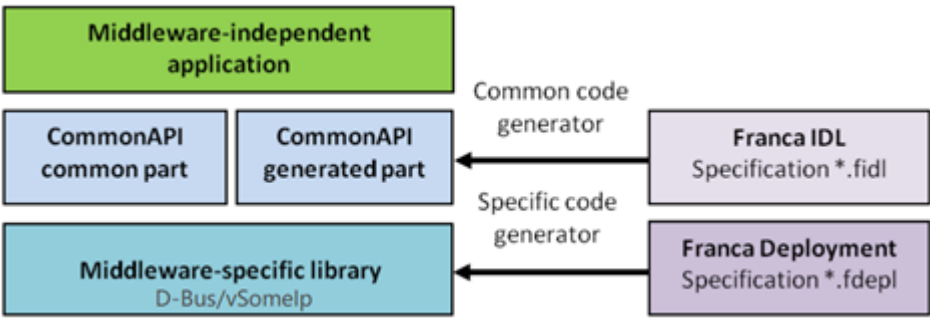


Figure 2. Deployment Concept

3.1CommonAPI部署

CommonAPI代码生成器几乎支持FrancaIDL的全部功能，并且无需任何部署文件即可工作。但是，可以根据以下部署规范，不仅为绑定而且为CommonAPI本身编写接口规范的部署文件。

可以在CommonAPI C++级别上设置枚举支持类型，不仅针对单个枚举，而且通常针对整个接口。请注意，绑定也可能具有与枚举的支持类型有关的部署设置。也可以定义函数调用的超时时间。设置此超时的另一种可能性是在方法调用的可选参数中对CallInfo进行定义。

代码生成器不评估实例和提供者的设置。

有关部署参数的用法，请参见下方，CommonAPI本身的所有部署参数都是可选的。

```
define org.genivi.commonapi.someip.deployment for Interface
SomeIpEventID = 4560 {
  method sayHello {
    SomeIpEventID = 33000
    SomeIpReliable = true
  }
  in {
    name {
      SomeIpStringEncoding = utf16le
    }
  }
}
}

define org.genivi.commonapi.someip.deployment for Provider MyService {
  instance CommonAPIHelloWorld {
    InstanceID = "commonapi.helloWorlds.HelloWorld"
    SomeIpInstanceID = 22136
  }
  SomeIpUnicastAddress = "192.168.0.2"
  SomeIpReliableUnicastPort = 30499
  SomeIpUnreliableUnicastPort = 30499
}

broadcast myStatus {
  SomeIpEventID = 33010
  SomeIpEventGroups = { 33010 }

  out {
  }
}
```

```
enumeration stdErrorTypeEnum {  
    NO_FAULT {  
  
    }  
    MY_FAULT {  
  
    }  
}
```

```
attribute a1 {  
    SomelpGetterID = 3002  
    SomelpServiceID = 3003  
    SomelpNotifierID = 33011  
    SomelpEventGroups = { 33011 }  
  
    SomelpGetterReliable = true  
    SomelpSetterReliable = true  
    SomelpNotifierReliable = true  
}  
..
```

```
struct phoneBookStruct {  
    name {  
    }  
    forename {  
    }  
    organisation {  
    }  
    address {  
    }  
    email {  
    }  
    phoneNumber {  
    }  
}
```

```
union SettingsUnion {  
    id {  
    }  
    status {  
    }  
    channel {  
  
    }  
    name {  
    }  
}
```

3.2*.fdepl文件的基本构成

类别1	类别2	说明	举例
for interface		对接口进行一些部署，设置ServiceID值，也可以设置枚举支持的类型	<i>*Reliable = false表示使用UDP协议，Reliable = true表示使用TCP协议</i>
	attribute	为属性的getter, setter等方法提供ID值	attribute x { SomelpGetterID = 3000 SomelpSetterID = 3001 SomelpNotifierID = 33010 SomelpEventGroups = { 33010 } SomelpGetterReliable = false SomelpSetterReliable = false SomelpNotifierReliable = false}
	method	设置method的ID值，并设置输入输出字符串类型的编码/解码格式；也可以订阅方法调用的超时	method foo { SomelpMethodID = 30000 SomelpReliable = false in { x2 { SomelpStringEncoding = utf16le } } out { y2 {SomelpStringEncoding =utf16le} } }
	broadcast	设置广播事件以及广播事件组的ID值，并设置输出字符串类型的编码/解码格式	broadcast myStatus { SomelpEventID = 33020 SomelpEventGroups = { 33020 } }
	array	定义数组的长度	SomelpArrayLengthWidth = 2
	enumeration	设置枚举的数据类型	EnumBackingType = UInt64
for provider		提供实例	

类别1	类别2	说明	举例
	instance	设置实例名称以及实例ID值，并设置实例的地址和端口号	instance commonapi.mthd.Method { InstanceId = “commonapi.mthd.Method” SomeInstanceId = 22136 SomeUnicastAddress = “192.168.0.2” SomeReliableUnicastPort = 30500 SomeUnreliableUnicastPort = 30501 }
for typeCollection		用户自己定义的各种数据类型 类型的集合	
	array	定义数组的长度	SomeArrayLengthWidth = 2
	enumeration	设置枚举的数据类型	EnumBackingType = UInt64

3.2for interface

for interface，对应于*.fidl文件中的interface，在这里面主要是配置中间件的ServiceID，以及其他method，broadcast，attribute等使用的methodID，eventID值等。

设置ServiceID:

```
1 | SomeIpServiceID = 4660
```

也可以在此定义整个接口的CommonAPI C++级别上定义枚举支持的类型，默认是UInt32：

```
1 | DefaultEnumBackingType: { UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, I
```

例如：

```
1 | DefaultEnumBackingType = UInt8
```

3.2.1attribute

为每个属性所使用的getter, setter等方法设置ID值，并设置这些方法的可靠性。

```
attribute x1 {
-- SomeIpGetterID = 1000
-- SomeIpSetterID = 1001
-- SomeIpNotifierID = 11010 .....
-- SomeIpEventGroups = { 11010 }
--
--
-- SomeIpGetterReliable = true
-- SomeIpSetterReliable = true
-- SomeIpNotifierReliable = true
}
```

3.2.2method

为方法method设置methodID值，并设置可靠性，也可以为in, out的输入输出参数的字符串类型设置编码/解码格式，也可以不设置，从而使用默认设置。

```
method foo {
  .. SomeIpMethodID = 30000
  .. SomeIpReliable = false
  .. in {
    .. x2 {
      .. SomeIpStringEncoding = utf16le
    }
  }
  .. out {
    .. y2 {
      .. SomeIpStringEncoding = utf16le
    }
  }
}
```

https://blog.csdn.net/C_Silence_K

也可以在此处设置超时：

```
1 | Timeout : Integer (default: 0);
```

例如：

```
1 | Timeout = 1
```

3.2.3broadcast

为broadcast事件设置EventID值，Event Groups值（这个是将自己想要划分为一组的事件ID写在一起），设置可靠性，也可以定义out参数里面字符串类型的编码/解码格式。

```
broadcast results {
  .. SomeIpEventID = 41913
  .. SomeIpReliable = true
  .. SomeIpEventGroups = { 17749 }

  .. out {
    ..
  }
}
```

3.3for typeCollection

3.3.1array

SomeIpArrayLengthWidth是决定长度字段的大小，表示数组序列化时在数组前面用于表示数组长度的字节数。

即SomeIpArrayLengthWidth =2表示数组在序列化时，前面需要加2个字节，用于表示数组的长度，允许的值是0、1、2、4。

0表示没有长度字段。

```
1 | array myArray {
2 |   SomeIpArrayLengthWidth = 2
3 | }
```

3.3.2enumeration

可以在此设置枚举使用的数据类型：

```
1 | EnumBackingType : {UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64} (option
```

例如：

```
1 | EnumBackingType = UInt64
```

3.4for provider

在此提供程序所依赖的所有服务实例(如果有)，并为实例设置名称，ID值以及IP地址和端口号。

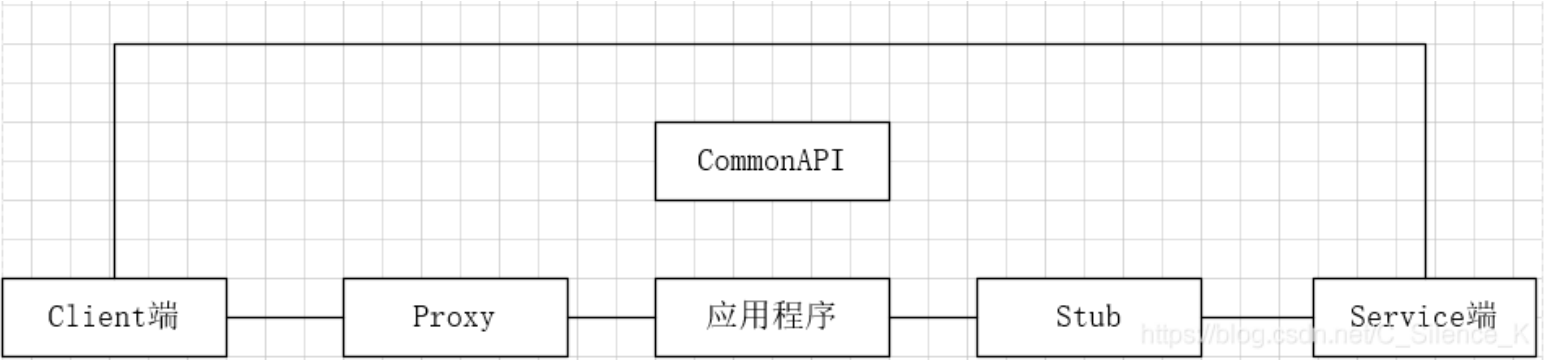
```
define org.genivi.commonapi.someip.deployment for provider MyService {
... instance commonapi.un.Unions {
... .. InstanceId = "commonapi.un.Unions"
... .. SomeIp InstanceID = 22136
... .. SomeIpUnicastAddress = "192.168.0.2"
... .. SomeIpReliableUnicastPort = 30490
... .. SomeIpUnreliableUnicastPort = 30491
... }
}
```

https://blog.csdn.net/C_Silence_K

三、应用程序编写

开发CommonAPI应用程序的第一步可能是客户端将用于与服务器通信的接口的定义。在CommonAPI的上下文中，无论最终打算使用哪种通信机制，此接口的定义始终通过Franca IDL进行。根据*.fidl文件与*.fdepl文件生成的代码文件，具体分为以下几部分，以及各部分的功能。

**fidl文件生成的接口代码		
	HelloWorld.hpp	用于客户端开发：代理是一个提供方法调用的类，该方法调用将导致对服务的远程方法调用，以及服务可以广播的事件的注册方法。
	HelloWorldProxy.hpp	
	HelloWorldProxyBase.hpp	
	HelloWorldStub.hpp	服务器开发：存根是服务的一部分，当来自客户端的远程方法调用到达时，存根将被调用，它还包含将事件（广播）激发到几个或所有客户端的方法。
	HelloWorldStubDefault.hpp	
	HelloWorldStubDefault.cpp	
*fdepl文件生成的粘合代码		名称中具有绑定名称（例如someip）的所有文件都是绑定所需的粘合代码，并且在开发应用程序时不相关，它们仅需与应用程序一起编译
	HelloWorldSomeIPDeployment.hpp	
	HelloWorldSomeIPDeployment.cpp	
	HelloWorldSomeIPProxy.cpp	
	HelloWorldSomeIPProxy.hpp	
	HelloWorldSomeIPStubAdapter.hpp	
	HelloWorldSomeIPStubAdapter.cpp	



这属于代理/存根结构，通过这张流程图，很容易发现CommonAPI实现IPC其实是在原来的C/S框架上加入了代理 / 存根结构。

具体的编写参考例子：https://blog.csdn.net/C_Silence_K/article/details/104674945

四、参考资料

- 1.CommonAPI+someip的配置与使用，操作流程，用HelloWorld演示
<https://at.projects.genivi.org/wiki/pages/viewpage.action?pageId=5472311>
- 2.CommonAPI C++规范
<https://docs.projects.genivi.org/ipc.common-api-tools/3.1.3/html/CommonAPICppSpecification.html>
- 3.CommonAPI C++用户指南
<https://docs.projects.genivi.org/ipc.common-api-tools/3.1.3/html/CommonAPICppUserGuide.html>
- 4.CommonAPI C++的使用例子
<https://github.com/GENIVI/capicxx-core-tools/tree/master/CommonAPI-Examples>



请发表有价值的评论， 博客评论不欢迎灌水，良好的社区氛围需大家一起维护。



评论



绿野耕夫: 很好的资料，谢谢整理。不过SomeIpArrayLengthWidth的解释好像是错误的，这个应该是表示数组序列化时在数组前面用于表示数组长度的字节数。即SomeIpArrayLengthWidth =2表示数组在序列化时，前面需要加2个字节，用于表示数组的长度。 1 年前 回复 ...



C_Silence_K

博主

回复： 谢谢指正，已经修改。

1 年前

回复

...



1