

# bluez系列学习9 - DBUS介绍

---

## 一. 概念

### 1. 概念

### 2. 常见术语

- a. Buses (总线)
- b. Connections (连接)
- c. Object Model (对象模型)
- d. Interfaces (接口)
- e. Method(方法)、属性(Property)
- f. Signals (信号)

### 3. dbus工具

#### a. d-feet

- i . 主要功能
- ii . 使用场景
- iii . 安装与运行

#### b. dbus-monitor

- i . 主要功能
- ii . 使用方法
- iii . 示例

## 二. python使用dbus介绍

### 1. 连接bus

### 2. 获取对象

### 3. 获取interface

### 4. 调用方法

### 5. 获取属性

### 6. 信号处理

### 7. 综合示例

## 三. C语言使用dbus介绍

## 四. C++使用dbus介绍

五 c# Tmds.DBus介绍

- 1. 介绍
- 2. 使用tmds.DBus.Tool生成
- 3. Tmds.DBus代码示例

版本	日期	作者	变更表述
1.0	2024/09/06	于忠军	文档创建

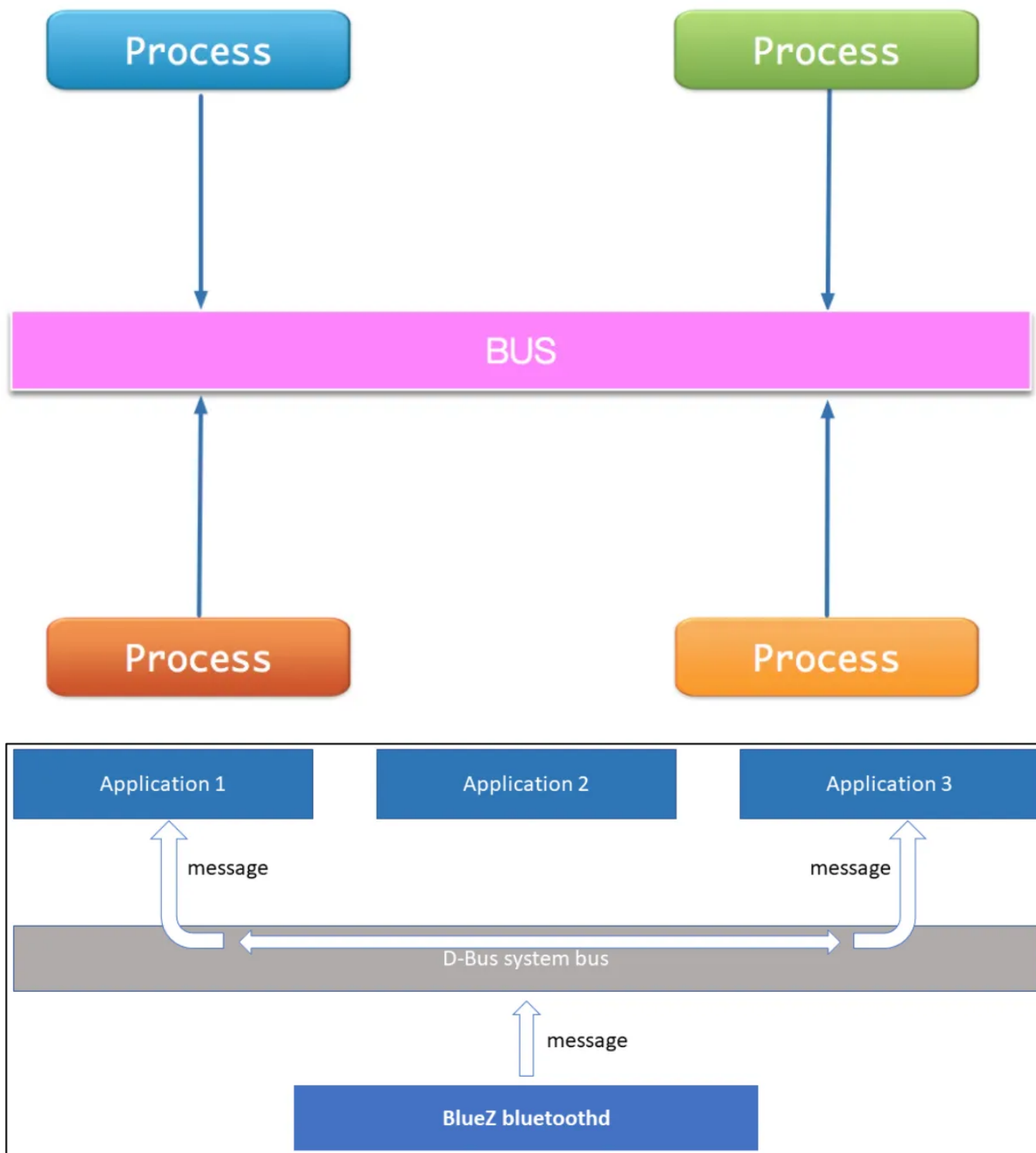
一. 概念

1. 概念

DBUS是一种高级的进程间通信机制(IPC)。DBUS支持进程间一对一和多对多的对等通信，在多对多的通讯时，需要后台进程的角色去分转消息，当一个进程发消息给另外一个进程时，先发消息到后台进程，再通过后台进程将信息转发到目的进程。DBUS后台进程充当着一个路由器的角色。

DBUS中主要概念为总线，连接到总线的进程可通过总线接收或传递消息，总线收到消息时，根据不同的消息类型进行不同的处理。DBUS中消息分为四类：

- Methodcall消息：将触发一个函数调用；
- Methodreturn消息：触发函数调用返回的结果；
- Error消息：触发的函数调用返回一个异常；
- Signal消息：通知，可以看作为事件消息。



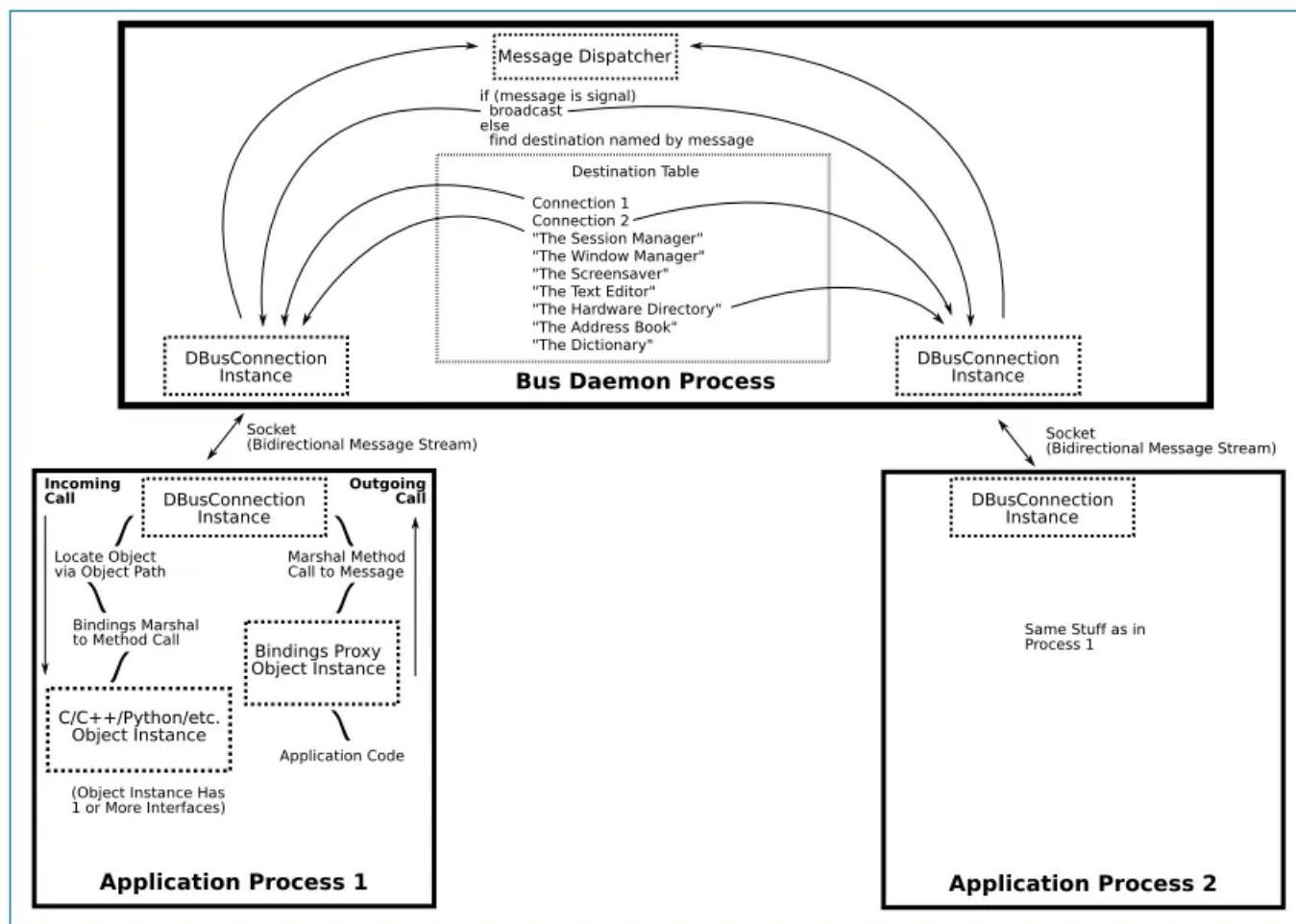
在蓝牙上就是以上图示会用到dbus

## 2. 常见术语

名称	表示方法	看起来就是这样	由谁管理
----	------	---------	------

总线	地址	org.bluez	系统设置
连接	连接地址	:34-907 (唯一名) com.mycompany.TextEditor (公共名)	D-Bus(唯一名) 服务进程 (公共名)
对象	路径	比如 适配器路径: /org/bluez/hci0  设备路径: /org/bluez/hci0/dev_XX_XX_XX_XX_XX_XX	服务进程
接口	接口名	org.bluez.Adapter1	服务进程
方法	函数名	ListNames	服务进程
属性	属性名	PropertyNames	服务进程
信号	信号名	SignalNames	服务进程

整个关系如下：



## a. Buses（总线）

在D-Bus中，“bus”是核心的概念，它是一个通道：不同的程序可以通过这个通道做些操作，比如方法调用、发送信号和监听特定的信号。在一台机器上总线守护有多个实例(instance)。这些总线之间都是相互独立的。支持dbus的系统都有两个标准的消息总线：**系统总线和会话总线**

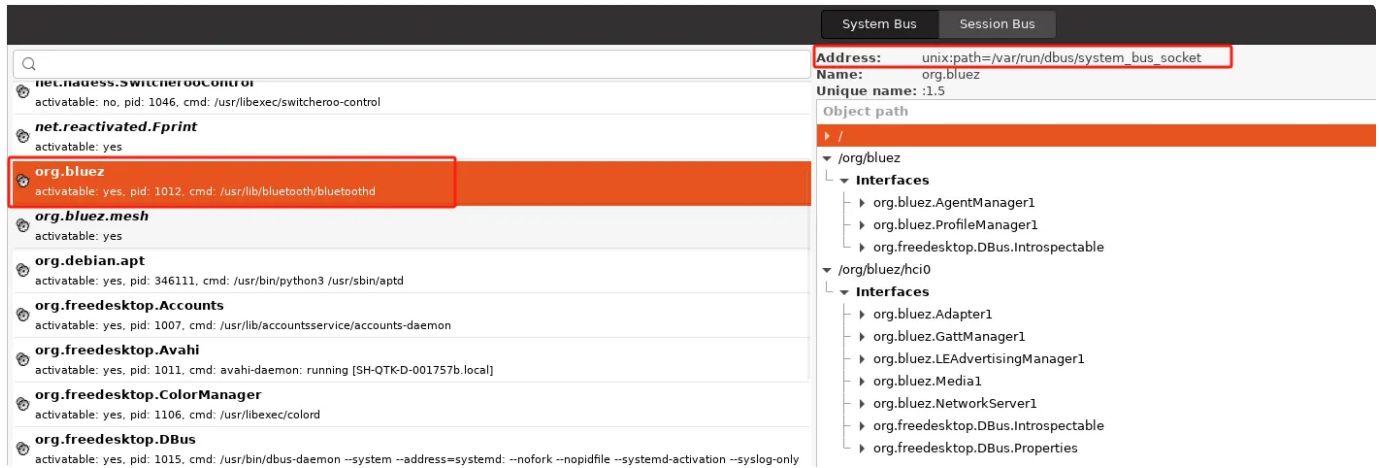
一个持久的系统总线（system bus）：

它在引导时就会启动。这个总线由操作系统和后台进程使用，安全性非常好，以使得任意的应用程序不能欺骗系统事件。它是桌面会话和操作系统的通信，这里操作系统一般而言包括内核和系统守护进程。这种通道的最常用的方面就是发送系统消息，比如：插入一个新的存储设备；有新的网络连接；等等。

还将有很多会话总线（session buses）：

这些总线当用户登录后启动，属于那个用户私有。它是用户的应用程序用来通信的一个会话总线。同一个桌面会话中两个桌面应用程序的通信，可使得桌面会话作为整体集成在一起以解决进程生命周期的相关问题。这在GNOME和KDE桌面中大量使用。

可以看到org.bluez的总线Address是：unix:path=/var/run/dbus/system\_bus\_socket

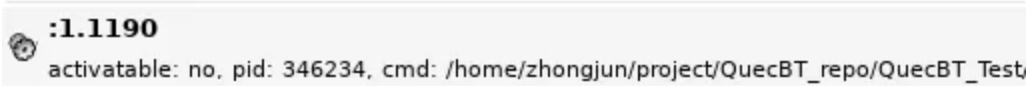


## b. Connections（连接）

总线上的每个连接都有一个或多个名字。这些名字一般叫做连接名（connection’s bus names 或简单地称为 bus names（注：bus name指的是连接名，而不是总线名。））。连接名由”.”分开的字符组成，比如”com.acme.Foo”，中间的字符可以是字母、数字、连接线、下划线。每个连接拥有一个连接名。

当连接建立以后，D-Bus服务会分配一个不可改变的连接名，称为唯一连接名，这个连接名即使在进程结束后也不会再被其他进程所使用。唯一连接名以冒号开头，像是这个样子”:34-907”（后面的数字没有任何意义，仅用来区分不同连接）

此外，连接还可以引证另外的连接名，比如提供服务的进程可以起一个大家都知道的连接名，以便使用其服务的客户进程连接。这个名字必须包含两个以上的“.”，比如：“com.acme.PortableHole”。与唯一名不同的是，这个连接名可以转给其他进程使用。我们随便写了一个程序，连接了DBus，可以看到分配的connection是：



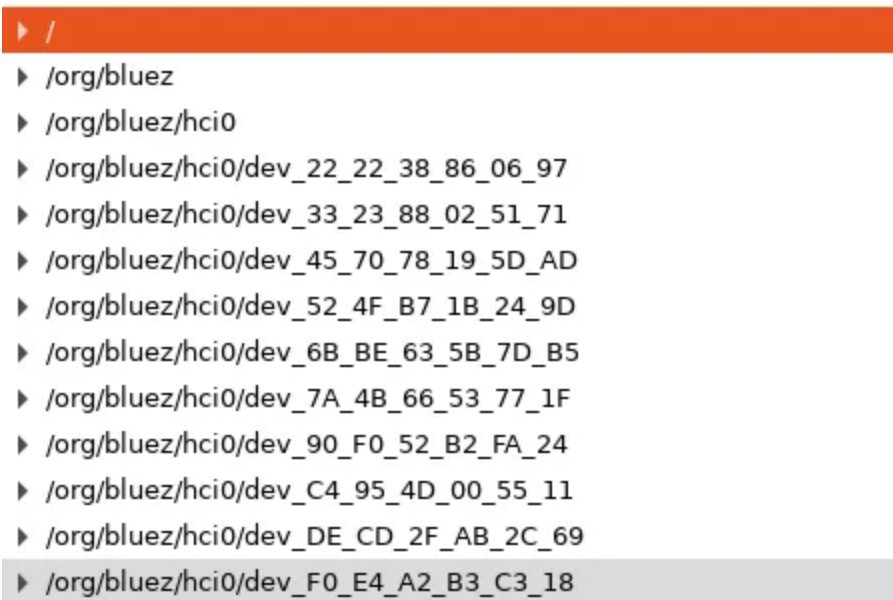
The screenshot shows a terminal window with a title bar that says ":1.1190". Below the title bar, the text "activatable: no, pid: 346234, cmd: /home/zhongjun/project/QuecBT\_repo/QuecBT\_Test" is displayed.

### c. Object Model（对象模型）

对象是一个独立的处理消息的实体。对象有一个或多个接口，在每个接口有一个或多个的方法，每个方法实现了具体的消息处理。在一对一的通讯中，对象通过一个连接直接和另一个客户端应用程序连接起来。在多对多的通讯中，对象通过一个连接和DBus后台进程连接起来。对象有一个路径用于指明该对象的存放位置，消息传递时通过该路径找到该对象。

客户端应用是一个桌面应用程序，是请求消息的发起者。客户端应用通过和自身的相连的一个连接将请求消息发送出去，也通过该连接接收回应的消息、错误消息、系统更新消息等。在一对一的通讯中，请求消息直接到达对象。在多对多的通讯中，请求消息先到达DBus后台，DBus后台将消息转发到目的对象。

我们同样截图下bluez的对象



The screenshot shows a terminal window with a title bar that says "/". Below the title bar, the contents of the /org/bluez directory are listed, including /org/bluez/hci0 and its subdirectories.

这些路径都是对象

当然有的语言不叫做对象，有的访问是通过创建一个对象的代理Proxies（代理）

得到服务代理后，可以在应用程序的各个地方通过对象代理的方法使用函数想对象发出一个方法调用的消息

## d. Interfaces（接口）

对象的所有成员都通过接口来定义。与JAVA类似，接口就是成员声明的集合。接口的实现就等于说这个接口提供了其所有方法及信号。所有成员必须按照接口声明的那样来提供服务。

对象可以都实现同一个接口，这与JAVA中不同的类可以实现同一个接口相似。另一方面，某个对象可以同时实现多个接口。与JAVA有所不同的是，D-Bus中没有实现任何接口的对象是没有意义的。对象所有支持的接口集合被称为对象的类型（object's type）

当客户进程要执行一个对象方法或等待信号时，它必须指明要使用的对象及其对象成员。除此以外，客户可能还要指明对象成员所在的接口，有时这是必须的步骤。比如，当对象同时在其两个接口都实现foo方法时，若客户不指明要执行的是哪个接口上的foo方法，则会产生歧义。D-Bus的实现也可能会直接拒绝这类歧义请求。同样地，如果在订阅信号时不指明接口，则也有可能收到不同接口发来的相同名字的信号，这通常不是预期的情况。D-Bus的较老版本中存在一个当出现歧义时丢失消息的BUG。

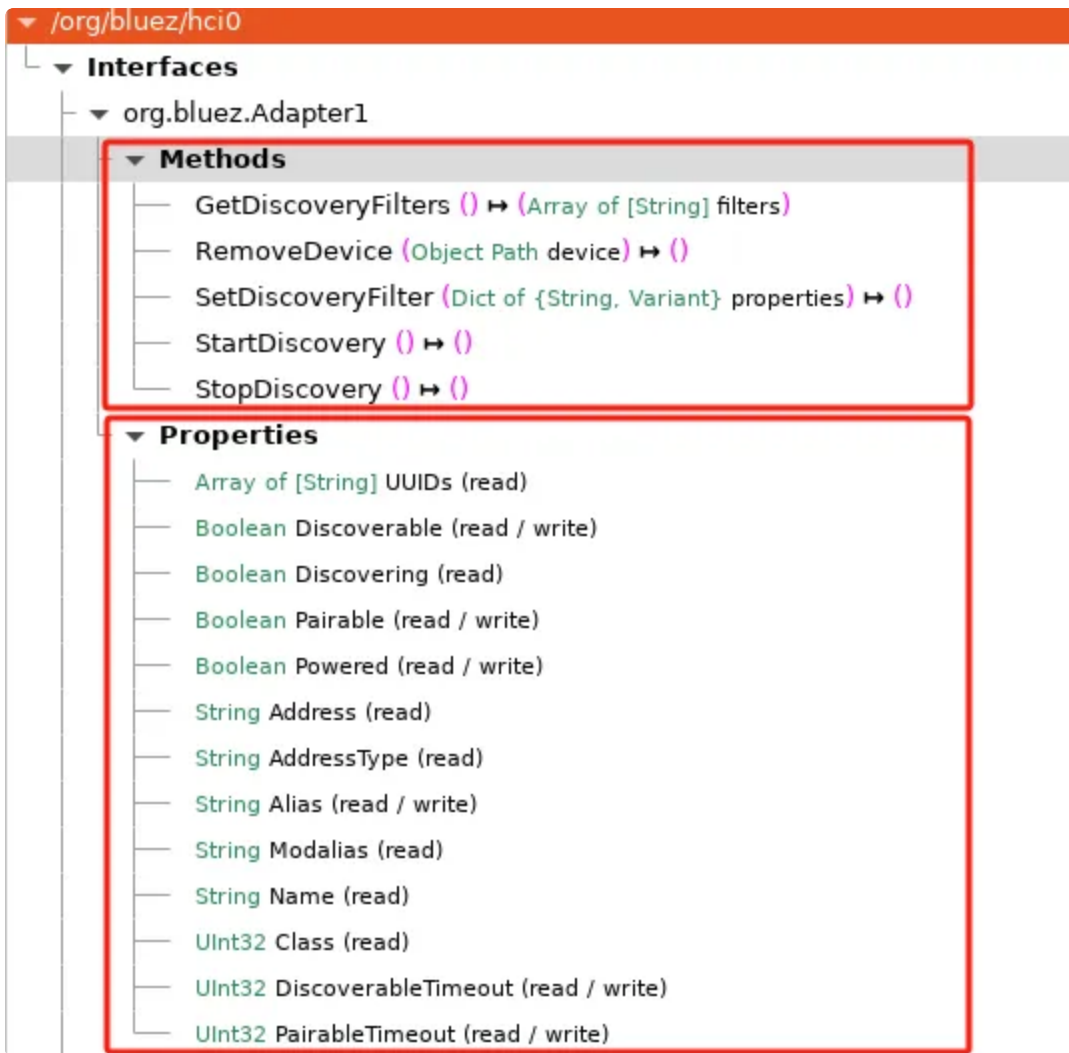
至于同一个接口内的对象是否可以重载，即接口内是否允许相同名字的对象取决于编程语言的实现。

我们来举例说明下接口：



## e. Method(方法)、属性(Property)

就是具体提供的函数功能/属性



## f. Signals (信号)

信号，这种通信形式依然遵从面向对象概念，它是从对象发出但没有特定目的地址的单向数据广播。客户进程可以预先注册其感兴趣的信号，如特定名称的信号或从某个对象发出的信号等。当对象发出信号后，所有订阅了该信号的客户将收到此信号的复本。接收端可能有多种情况出现，或者有一个客户，或者有多个客户，或者根本没有客户对这个信号感兴趣。对于信号来说没有响应消息，发出信号的对象不会知道是不是有客户在接收，有多少客户接收，以及从客户端收到任何反馈。

与方法类似的是，信号可以有参数。由于信号是单向通信，因此其不可能像方法一样具有输入输出参数。D-Bus的最新版本允许客户通过参数比对过滤其需要的信号。

信号一般用来广播一些客户可能会感兴趣的事件，比如某个其他的客户与总线的连接断开等。这些信号来自总线对象，因此从信号中客户可以分辨断线是由于正常退出、被杀掉或者程序崩溃。

## 3. dbus工具



## a. d-feet

**D-Feet** 是一个图形化的 D-Bus 调试工具，允许用户浏览和操作 D-Bus 总线上的服务、对象、接口和方法。它为开发者和系统管理员提供了一个简单易用的界面，用于检查和调试基于 D-Bus 的应用程序和服务。

### i. 主要功能

- **浏览 D-Bus 总线:**
  - **D-Feet** 可以连接到系统总线（System Bus）或会话总线（Session Bus），并显示在这些总线上注册的所有服务。
  - 用户可以浏览每个服务的对象路径、接口和方法。
- **调用 D-Bus 方法:**
  - 用户可以选择一个特定的接口和方法，并通过图形界面调用该方法。
  - 可以输入参数并查看方法的返回值，方便调试和验证 D-Bus 方法的行为。
- **监视信号:**
  - **D-Feet** 允许用户监视 D-Bus 上的信号，这在调试实时系统事件时非常有用。
  - 用户可以订阅感兴趣的信号，并查看它们的内容。
- **查看和编辑属性:**
  - 用户可以查看 D-Bus 对象的属性，并在某些情况下直接修改属性值。
  - 这使得调试和测试属性变化对系统的影响变得更加简单。
- **支持多种 D-Bus 总线:**
  - **D-Feet** 支持同时连接多个 D-Bus 总线，比如系统总线和用户会话总线，并在同一界面中管理它们。

### ii. 使用场景

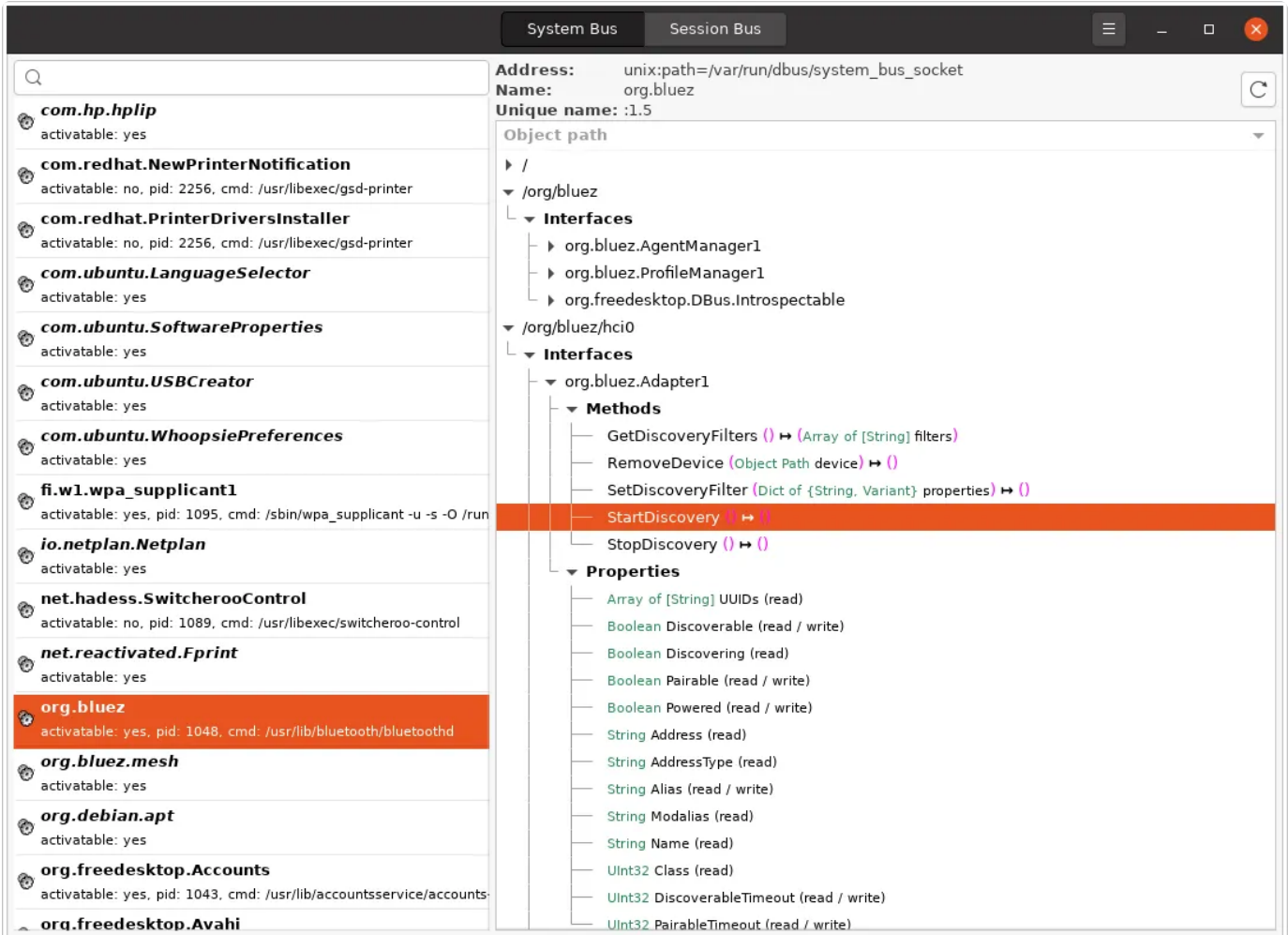
- **开发调试:** 开发者可以使用 **D-Feet** 检查 D-Bus 服务是否按照预期注册，并通过调用方法和监视信号来调试应用程序。
- **系统诊断:** 系统管理员可以使用 **D-Feet** 检查系统服务的运行状态，验证配置，或在服务出现问题时进行故障排查。
- **学习和教学:** 新手可以通过 **D-Feet** 直观地了解 D-Bus 的工作机制，并学习如何与 D-Bus 服务进行交互。

### iii. 安装与运行

在大多数基于 Linux 的系统中，可以通过包管理器安装 **D-Feet**，例如：

```
1 sudo apt install d-feet
```

安装完成后，可以通过命令行输入 **d-feet** 来启动它。



## b. dbus-monitor

**dbus-monitor** 是一个命令行工具，用于监视 D-Bus 消息总线上传输的消息和信号。它可以帮助开发者和系统管理员调试 D-Bus 通信，查看进程间发送的消息，了解系统事件，或者分析应用程序之间的交互。

### i. 主要功能

- 监视 D-Bus 消息：
  - **dbus-monitor** 可以捕获并显示系统总线或会话总线上所有传输的 D-Bus 消息。这些消息

包括方法调用、返回值、信号以及错误。

- 用户可以实时查看哪些进程在发送或接收消息，以及消息的详细内容。

- **过滤消息:**

- 可以指定过滤条件，以只监视特定类型的消息。例如，可以只监视特定服务或特定类型的信号，这在调试时非常有用。
- 通过选择合适的过滤条件，可以减少输出的冗余信息，聚焦于感兴趣的事件。

- **分析系统行为:**

- 通过监视 D-Bus 信号，用户可以了解系统中发生的事件。例如，网络连接的变化、设备的插拔、用户会话的管理等。
- 这对于调试系统级别的问题和了解应用之间的通信机制非常有帮助。

- **调试应用程序:**

- 开发者可以使用 `dbus-monitor` 来检查应用程序是否正确地发送或接收了 D-Bus 消息，并验证消息的格式和内容是否符合预期。
- 通过监视应用程序之间的消息交互，可以更好地理解应用程序的行为，并发现潜在的问题。

## ii. 使用方法

`dbus-monitor` 的基本用法如下：

▼ Plain Text |

```
1  dbus-monitor [选项]
```

常用选项包括：

- **--system:** 监视系统总线上的 D-Bus 消息。系统总线用于系统服务之间的通信，例如 `org.free desktop.NetworkManager`。

▼ Plain Text |

```
1  dbus-monitor --system
```

- **--session:** 监视用户会话总线上的 D-Bus 消息。会话总线通常用于桌面环境中的应用程序之间的通信。

▼ Plain Text |

```
1  dbus-monitor --session
```

- **--address=ADDRESS:** 监视指定 D-Bus 地址上的消息，通常用于自定义或非标准的 D-Bus 总

线。

- **--profile**: 以更简洁的方式显示 D-Bus 消息的概要信息，而不是详细的消息内容。这个模式适合分析系统的消息流量和性能。

### iii. 示例

比如我们在d-feet上点击一个搜索，那么我们用dbus-monitor --system会返回如下信息：

```
signal time=1725516169.192222 sender=:1.5 -> destination=(null destination) serial=1417 path=/org/bluez/hci0/dev_1A_3A_C3_BF_59_E2; interface=org.freedesktop.DBus.Properties; member=PropertiesChanged
string "org.bluez.Device1"
array [
  dict_entry(
    string "RSSI"
    variant
      int16 -55
  )
] array [
]
signal time=1725516169.771475 sender=:1.5 -> destination=(null destination) serial=1418 path=/org/bluez/hci0/dev_1A_3A_C3_BF_59_E2; interface=org.freedesktop.DBus.Properties; member=PropertiesChanged
string "org.bluez.Device1"
array [
  dict_entry(
    string "RSSI"
    variant
      int16 -63
  )
] array [
]
```

## 二. python使用dbus介绍

以下链接是用各种语言来操作dbus的

<https://www.freedesktop.org/wiki/Software/DBusBindings/>

至于python的使用我们直接采用dbus-python，这个是(based on DBus-GLib)

### 1. 连接bus

要与 D-Bus 进行交互，首先需要连接到总线。D-Bus 中有两种主要的总线：

- **系统总线 ( System Bus )**: 用于系统范围内的通信，比如硬件管理器、系统服务等。它通常在 `/var/run/dbus/system_bus_socket` 运行。
- **会话总线 ( Session Bus )**: 用于单个用户会话内的通信，通常用于桌面应用程序之间的通信。

在 `dbus-python` 中，可以通过以下方式连接到这些总线：

```
1 import dbus
2
3 session_bus = dbus.SessionBus()
4 system_bus = dbus.SystemBus()
```

Python

## 2. 获取对象

`bus.get_object` 是 `dbus-python` 库中的一个方法，用于获取一个 D-Bus 对象的代理 (proxy)。这个方法需要两个关键参数：**服务名称** 和 **对象路径**，并且可以接收一些可选参数。

### 参数介绍

- **bus\_name** (服务名称)

- 类型: `str`
- 说明: 这是 D-Bus 上注册的服务名称，用于指定你要与哪个 D-Bus 服务进行通信。例如，在与 `BlueZ` 交互时，服务名称通常为 `"org.bluez"`。

```
1 bus_name = "org.bluez"
```

- **object\_path** (对象路径)

- 类型: `str`
- 说明: 这是 D-Bus 对象的路径，表示你要获取的具体对象。例如，在与 `BlueZ` 交互时，蓝牙适配器的对象路径通常为 `"/org/bluez/hci0"`。这个路径是 D-Bus 服务内部的对象表示方式，类似于文件系统中的路径。

```
1 object_path = "/org/bluez/hci0"
```

- **introspect** (可选参数)

- 类型: `bool`
- 默认值: `True`
- 说明: 这个参数决定了是否在对象代理创建时自动执行 introspection (自省)。自省是指从 D-Bus 服务获取对象的接口和方法信息。通常你不需要显式指定这个参数，使用默认的 `True` 即可。

```
1 adapter = bus.get_object("org.bluez", "/org/bluez/hci0", introspect=False)
```

- **follow\_name\_owner\_changes** (可选参数)

- 类型: `bool`
- 默认值: `False`

- 说明: 如果设为 `True` , 对象代理将会在服务的所有权发生变化时自动更新。这在一些服务动态启动或停止的场景下可能有用。

```
Python |  
1 adapter = bus.get_object("org.bluez", "/org/bluez/hci0", follow_name_owner_  
    changes=True)
```

## 返回值

`bus.get_object` 方法返回一个 D-Bus 对象的代理 (proxy) 。这个代理对象可以用来调用该对象支持的 D-Bus 方法、访问属性以及订阅信号。

## 3. 获取interface

在 `dbus-python` 中, `dbus.Interface` 是用于获取特定 D-Bus 接口的代理对象的类。通过这个代理对象, 你可以调用接口上定义的方法, 并处理与该接口相关的 D-Bus 信号。

### 参数介绍

- **object** (对象代理)
  - 类型: `dbus.proxy.ObjectProxy`
  - 说明: 这是通过 `bus.get_object` 获取到的 D-Bus 对象代理。它表示特定服务的特定对象, 通常包括多个接口。你需要这个对象代理来获取某个具体接口的代理对象。

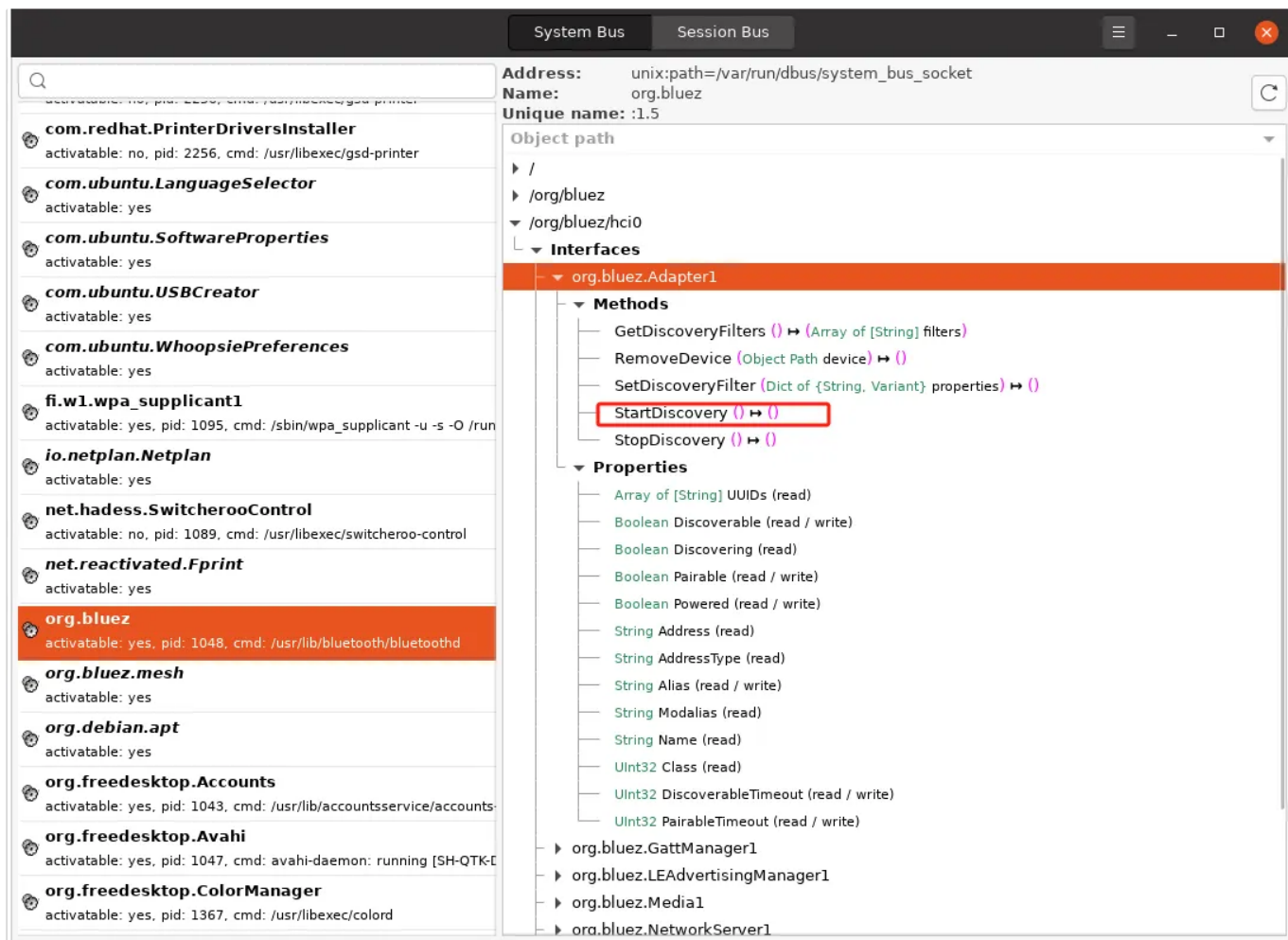
```
Python |  
1 adapter = bus.get_object("org.bluez", "/org/bluez/hci0")
```

- **dbus\_interface** (接口名称)
  - 类型: `str`
  - 说明: 这是 D-Bus 接口的名称, 指定了你要获取的接口。D-Bus 对象可能实现多个接口, 每个接口定义了一组方法和信号。接口名称通常是一个全局唯一的字符串。例如, 在 `BlueZ` 中, 一个常用的接口是 `"org.bluez.Adapter1"` 。

```
Python |  
1 interface = dbus.Interface(adapter, "org.bluez.Adapter1")
```

## 4. 调用方法

在连接bus后，然后获取对象，获取interface后，就可以通过interface来调用特定的方法了，这个方法可以通过d-feet来查看：



比如我们要做下搜索，那么代码示例如下：

```
Python |
1 bus = dbus.SystemBus()
2
3 adapter_path = "/org/bluez/hci0"
4 adapter = dbus.Interface(bus.get_object("org.bluez", adapter_path), "org.bluez.Adapter1")
5 adapter.StartDiscovery()
```

## 5. 获取属性

D-Bus 对象的属性通过 `org.freedesktop.DBus.Properties` 接口来访问。你需要获取该接口的代理对象。

**Address:** unix:path=/var/run/dbus/system\_bus\_socket  
**Name:** org.bluez  
**Unique name:** :1.5

#### Object path

- ▶ /
- ▶ /org/bluez
- ▼ /org/bluez/hci0
  - ▼ Interfaces
    - ▶ org.bluez.Adapter1
    - ▶ org.bluez.GattManager1
    - ▶ org.bluez.LEAdvertisingManager1
    - ▶ org.bluez.Media1
    - ▶ org.bluez.NetworkServer1
    - ▶ org.freedesktop.DBus.Introspectable
    - ▶ org.freedesktop.DBus.Properties
  - ▶ /org/bluez/hci0/dev\_DB\_1C\_05\_24\_B1\_5A
  - ▶ /org/bluez/hci0/dev\_DB\_3C\_E6\_C1\_E9\_2F
  - ▶ /org/bluez/hci0/dev\_DB\_9A\_BD\_7F\_03\_F3

`properties_interface.Get` 是 `org.freedesktop.DBus.Properties` 接口中用于获取 D-Bus 对象属性的标准方法。该方法接受两个参数，分别用于指定要获取的属性所属的接口名称和属性名称。

#### 参数介绍

- **interface\_name** (接口名称)
  - 类型: `str`
  - 说明: 这是属性所属的接口的名称。D-Bus 对象可能实现多个接口，每个接口定义了一组属性。你需要指定你想获取属性的接口名称。对于 `BlueZ` 适配器，接口名称通常是 `"org.bluez.Adapter1"`。

```
1 interface_name = "org.bluez.Adapter1"
```

- **property\_name** (属性名称)
  - 类型: `str`
  - 说明: 这是你想获取的属性的名称。属性名称通常是一个字符串，表示具体的属性，例如 `Name` 或 `Alias`。这个属性名称必须在你指定的接口中定义。



```
1 property_name = "Name"
```

## 返回值

- `properties_interface.Get` 方法返回指定属性的值，类型取决于该属性的具体数据类型。例如，`Name` 属性可能返回一个字符串，而其他属性可能返回整数、布尔值或其他类型。

举例如下，这个是获取本地蓝牙名称以及蓝牙地址

```
1 import dbus
2
3 # 连接到系统总线
4 bus = dbus.SystemBus()
5
6 # 获取蓝牙适配器对象 (hci0)
7 adapter_path = "/org/bluez/hci0"
8 adapter = bus.get_object("org.bluez", adapter_path)
9
10 # 获取 Properties 接口的代理对象
11 properties_interface = dbus.Interface(adapter, "org.freedesktop.DBus.Properties")
12
13 # 获取适配器的名称
14 adapter_name = properties_interface.Get("org.bluez.Adapter1", "Name")
15 print(f"Adapter Name: {adapter_name}")
16
17 # 获取适配器的别名 (本地蓝牙设备的名字)
18 adapter_alias = properties_interface.Get("org.bluez.Adapter1", "Alias")
19 print(f"Adapter Alias: {adapter_alias}")
20
21 # 获取蓝牙地址 (本地蓝牙地址)
22 address = properties_interface.Get("org.bluez.Adapter1", "Address")
23 print(f"address: {address}")
```

## 6. 信号处理

在 `BlueZ` 中，信号 (Signals) 是 D-Bus 用来异步通知事件发生的机制。通过处理信号，你可以响应蓝牙设备连接、断开、属性变化等各种事件。使用 `add_signal_receiver` 方法将信号连接到处理函数。你需要指定信号的名称、发出信号的接口名称，以及可选的对象路径和参数。

**`bus.add_signal_receiver` 参数介绍**

- **handler\_function** (必选)

- 类型: `callable`
- 说明: 信号触发时调用的处理函数。该函数会接收来自信号的数据作为参数。参数的数量和类型取决于发出的信号。

▼

Plain Text |

```
1 def my_signal_handler(*args):
2     print("Signal received with arguments:", args)
```

- **signal\_name** (可选)

- 类型: `str`
- 说明: 要监听的信号名称。如果不指定, 将接收该接口下的所有信号。

▼

Plain Text |

```
1 signal_name="PropertiesChanged"
```

- **dbus\_interface** (可选)

- 类型: `str`
- 说明: 信号所属的 D-Bus 接口名称。如果不指定, 将监听来自所有接口的信号。

▼

Plain Text |

```
1 dbus_interface="org.freedesktop.DBus.Properties"
```

- **bus\_name** (可选)

- 类型: `str`
- 说明: 发出信号的 D-Bus 服务名称。如果不指定, 将监听来自所有服务的信号。

▼

Plain Text |

```
1 bus_name="org.bluez"
```

- **path** (可选)

- 类型: `str`
- 说明: 发出信号的对象路径。如果不指定, 将监听来自所有路径的信号。

▼

Plain Text |

```
1 path="/org/bluez/hci0"
```

- **path\_keyword** (可选)

- 类型: `str`
- 说明: 如果指定, 将在处理函数中添加一个额外的关键字参数, 其值为信号来源的对象路径。这个参数的名称由 `path_keyword` 指定。



Plain Text

```
1 path_keyword="object_path"
```

- **interface\_keyword** (可选)

- 类型: `str`
- 说明: 如果指定, 将在处理函数中添加一个额外的关键字参数, 其值为信号来源的接口名称。这个参数的名称由 `interface_keyword` 指定。



Plain Text

```
1 interface_keyword="interface"
```

- **sender\_keyword** (可选)

- 类型: `str`
- 说明: 如果指定, 将在处理函数中添加一个额外的关键字参数, 其值为信号来源的 D-Bus 名称。这个参数的名称由 `sender_keyword` 指定。



Plain Text

```
1 sender_keyword="sender"
```

- **destination\_keyword** (可选)

- 类型: `str`
- 说明: 如果指定, 将在处理函数中添加一个额外的关键字参数, 其值为信号目标的 D-Bus 名称。这个参数的名称由 `destination_keyword` 指定。



Plain Text

```
1 destination_keyword="destination"
```

- **arg0** (可选)

- 类型: `Any`
- 说明: 用于指定信号的第一个参数的值来过滤信号。如果信号的第一个参数与此值不匹配, 则信号处理函数不会被调用。

```
1 arg0="hci0"
```

- **\*\*kwargs** (可选)
  - 类型: `dict`
  - 说明: 可以传递其他可选参数来进一步过滤信号或自定义行为。

## 7. 综合示例

这个示例就是来进行蓝牙BR/EDR的inquiry，并且搜索到设备，打印蓝牙地址，名称

```

1  import dbus
2  import dbus.mainloop.glib
3  from gi.repository import GLib
4
5  bus = None
6
7  def device_found(interface, changed, invalidated, path):
8      global bus
9      # 获取 D-Bus 对象的接口列表
10     obj = bus.get_object("org.bluez", path)
11     introspect = dbus.Interface(obj, "org.freedesktop.DBus.Introspectable"
12 )
13     xml = introspect.Introspect()
14
15     if "org.bluez.Device1" in xml:
16         # 只有在对象中存在 'org.bluez.Device1' 接口时才继续
17         device = dbus.Interface(obj, "org.freedesktop.DBus.Properties")
18
19         try:
20             address = device.Get("org.bluez.Device1", "Address")
21             try:
22                 name = device.Get("org.bluez.Device1", "Name")
23             except dbus.exceptions.DBusException:
24                 name = "Unknown"
25             print(f"Found device: {name} [{address}]")
26         except dbus.exceptions.DBusException as e:
27             print(f"Error accessing device properties: {e}")
28     else:
29         print(f"No 'org.bluez.Device1' interface at {path}")
30
31 def scan_for_devices():
32     global bus
33     dbus.mainloop.glib.DBusGMainLoop(set_as_default=True)
34     bus = dbus.SystemBus()
35
36     adapter_path = "/org/bluez/hci0"
37     adapter = dbus.Interface(bus.get_object("org.bluez", adapter_path), "o
38 rg.bluez.Adapter1")
39
40     bus.add_signal_receiver(device_found, dbus_interface="org.freedesktop.
41 DBus.Properties", signal_name="PropertiesChanged", path_keyword="path")
42
43     adapter.StartDiscovery()
44     print("Scanning for devices...")
45
46

```

```

43     mainloop = GLib.MainLoop()
44     try:
45         mainloop.run()
46     except KeyboardInterrupt:
47         print("Stopping Bluetooth scan...")
48         adapter.StopDiscovery()
49         mainloop.quit()
50
51 if __name__ == "__main__":
52     scan_for_devices()
53

```

## 三. C语言使用dbus介绍

## 四. C++使用dbus介绍

## 五 c# Tmds.DBus介绍

### 1. 介绍

视频介绍: [https://www.youtube.com/watch?v=fPsyXV8\\_kRg](https://www.youtube.com/watch?v=fPsyXV8_kRg)

TMDS.DBus 和 dbus-sharp 是两个用于与 D-Bus 通信的 C# 库, 它们有一些关键的区别:

#### 项目维护与活跃度

- **TMDS.DBus:** 由 Tmds 维护, 更新较为频繁, 项目较为活跃。Tmds 是一个活跃的开源贡献者, 维护了多个与 Linux 和 .NET 相关的项目。
- **dbus-sharp:** 这是一个较老的库, 项目更新不频繁。最初是为了支持 .NET 框架, 但对于现代 .NET Core 和 .NET 5+ 的支持不如 TMDS.DBus 那么好。

#### API 设计

- **TMDS.DBus:** 提供了更现代化、更简洁的 API。它利用了 C# 的最新特性, 如异步编程 (async/await), 以及自动生成的代理和消息协议解析。

- **dbus-sharp**: API 设计较为传统, 适用于较老的 C# 版本。异步编程支持有限, 使用起来可能比较繁琐。

### 性能

- **TMDs.DBus**: 通常性能较好, 因为它是为现代 .NET 环境设计的, 利用了 .NET Core/.NET 5+ 的性能优化。
- **dbus-sharp**: 性能较为一般, 设计时未充分考虑现代 .NET 环境的优化。

### 易用性和文档

- **TMDs.DBus**: 文档较为丰富, 示例代码较多。API 更加直观和易用, 适合现代 .NET 开发者。
- **dbus-sharp**: 文档较为陈旧, 示例代码较少, 使用起来需要更多的调试和理解。

### 兼容性

- **TMDs.DBus**: 兼容 .NET Core 和 .NET 5+, 适合跨平台开发。
- **dbus-sharp**: 主要设计针对 .NET Framework, 尽管可以在 .NET Core 上运行, 但不如 TMDs.DBus 那么优化。

tmds.dbus的github的链接为: <https://github.com/tmds/Tmds.DBus?tab=readme-ov-file>

## 2. 使用tmds.DBus.Tool生成

在有c#的工程中

安装Tmds.DBus.Tool

```
dotnet tool install --global Tmds.DBus.Tool --version 0.19.0
```

```
zhongjun@SH-QTK-D-001757b:~/project/QuecBT_repo/QuecBT_Test/linux/Fuzzy/btstack/test/dbus_test2$ dotnet tool install --global Tmds.DBus.Tool --version 0.19.0
You can invoke the tool using the following command: dotnet-dbus
Tool 'tmds.dbus.tool' (version '0.19.0') was successfully installed.
```

列举服务

```
dotnet-dbus list services // 列举session bus的服务
```

```
dotnet-dbus list --bus system services // 列举system bus的服务
```

列举对象

```
dotnet-dbus list --bus system --service org.bluez objects
```

列举interface

```
dotnet-dbus list --bus system --service org.bluez interfaces
```

生成代码

```
dotnet-dbus codegen --bus system --service org.bluez
```

### 3. Tmds.DBus代码示例



```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Threading.Tasks;
5  using Tmds.DBus;
6
7  [assembly: InternalsVisibleTo(Tmds.DBus.Connection.DynamicAssemblyName)]
8  namespace bluez.DBus
9  {
10     [DBusInterface("org.bluez.Adapter1")]
11     interface IAdapter1 : IDBusObject
12     {
13         Task StartDiscoveryAsync();
14         Task SetDiscoveryFilterAsync(IDictionary<string, object> Properties);
15         Task StopDiscoveryAsync();
16         Task RemoveDeviceAsync(ObjectPath Device);
17         Task<string[]> GetDiscoveryFiltersAsync();
18         Task<T> GetAsync<T>(string prop);
19         Task<Adapter1Properties> GetAllAsync();
20         Task SetAsync(string prop, object val);
21         Task<IDisposable> WatchPropertiesAsync(Action<PropertyChanges> handler);
22     }
23
24     [Dictionary]
25     class Adapter1Properties
26     {
27         private string _Address = default(string);
28         public string Address
29         {
30             get
31             {
32                 return _Address;
33             }
34             set
35             {
36                 _Address = (value);
37             }
38         }
39     }
40
41     private string _AddressType = default(string);
42     public string AddressType
43     {

```

```

44         get
45     {
46         return _AddressType;
47     }
48
49     set
50 {
51     _AddressType = (value);
52 }
53 }
54
55 private string _Name = default(string);
56 public string Name
57 {
58     get
59     {
60         return _Name;
61     }
62
63     set
64     {
65         _Name = (value);
66     }
67 }
68
69 private string _Alias = default(string);
70 public string Alias
71 {
72     get
73     {
74         return _Alias;
75     }
76
77     set
78     {
79         _Alias = (value);
80     }
81 }
82
83 private uint _Class = default(uint);
84 public uint Class
85 {
86     get
87     {
88         return _Class;
89     }
90
91     set

```

```

92     {
93         _Class = (value);
94     }
95 }
96
97 private bool _Powered = default(bool);
98 public bool Powered
99 {
100     get
101     {
102         return _Powered;
103     }
104     set
105     {
106         _Powered = (value);
107     }
108 }
109
110 private bool _Discoverable = default(bool);
111 public bool Discoverable
112 {
113     get
114     {
115         return _Discoverable;
116     }
117     set
118     {
119         _Discoverable = (value);
120     }
121 }
122
123 private uint _DiscoverableTimeout = default(uint);
124 public uint DiscoverableTimeout
125 {
126     get
127     {
128         return _DiscoverableTimeout;
129     }
130     set
131     {
132         _DiscoverableTimeout = (value);
133     }
134 }
135
136 private bool _Pairable = default(bool);

```

```

140     public bool Pairable
141     {
142     get
143     {
144         return _Pairable;
145     }
146
147     set
148     {
149         _Pairable = (value);
150     }
151     }
152
153     private uint _PairableTimeout = default(uint);
154     public uint PairableTimeout
155     {
156     get
157     {
158         return _PairableTimeout;
159     }
160
161     set
162     {
163         _PairableTimeout = (value);
164     }
165     }
166
167     private bool _Discovering = default(bool);
168     public bool Discovering
169     {
170     get
171     {
172         return _Discovering;
173     }
174
175     set
176     {
177         _Discovering = (value);
178     }
179     }
180
181     private string[] _UUIDs = default(string[]);
182     public string[] UUIDs
183     {
184     get
185     {
186         return _UUIDs;
187     }

```

```

188
189         set
190     {
191         _UUIDs = (value);
192     }
193     }
194
195     private string _Modalias = default(string);
196     public string Modalias
197     {
198         get
199         {
200             return _Modalias;
201         }
202
203         set
204         {
205             _Modalias = (value);
206         }
207     }
208 }
209
210 static class Adapter1Extensions
211 {
212     public static Task<string> GetAddressAsync(this IAdapter1 o) => o
213     .GetAsync<string>("Address");
214     public static Task<string> GetAddressTypeAsync(this IAdapter1 o)
215     => o.GetAsync<string>("AddressType");
216     public static Task<string> GetNameAsync(this IAdapter1 o) => o.Ge
217     tAsync<string>("Name");
218     public static Task<string> GetAliasAsync(this IAdapter1 o) => o.G
219     etAsync<string>("Alias");
220     public static Task<uint> GetClassAsync(this IAdapter1 o) => o.Get
221     Async<uint>("Class");
222     public static Task<bool> GetPoweredAsync(this IAdapter1 o) => o.G
223     etAsync<bool>("Powered");
224     public static Task<bool> GetDiscoverableAsync(this IAdapter1 o) =
225     > o.GetAsync<bool>("Discoverable");
226     public static Task<uint> GetDiscoverableTimeoutAsync(this IAdapte
227     r1 o) => o.GetAsync<uint>("DiscoverableTimeout");
228     public static Task<bool> GetPairableAsync(this IAdapter1 o) => o.
229     GetAsync<bool>("Pairable");
230     public static Task<uint> GetPairableTimeoutAsync(this IAdapter1 o
231     ) => o.GetAsync<uint>("PairableTimeout");
232     public static Task<bool> GetDiscoveringAsync(this IAdapter1 o) =>
233     o.GetAsync<bool>("Discovering");
234     public static Task<string[]> GetUUIDsAsync(this IAdapter1 o) => o
235     .GetAsync<string[]>("UUIDs");

```

```

224         public static Task<string> GetModaliasAsync(this IAdapter1 o) =>
225         o.GetAsync<string>("Modalias");
226         public static Task SetAliasAsync(this IAdapter1 o, string val) =>
227         o.SetAsync("Alias", val);
228         public static Task SetPoweredAsync(this IAdapter1 o, bool val) =>
229         o.SetAsync("Powered", val);
230         public static Task SetDiscoverableAsync(this IAdapter1 o, bool val) =>
231         o.SetAsync("Discoverable", val);
232         public static Task SetDiscoverableTimeoutAsync(this IAdapter1 o,
233         uint val) => o.SetAsync("DiscoverableTimeout", val);
234         public static Task SetPairableAsync(this IAdapter1 o, bool val) =>
235         o.SetAsync("Pairable", val);
236         public static Task SetPairableTimeoutAsync(this IAdapter1 o, uint
237         val) => o.SetAsync("PairableTimeout", val);
238     }
239 }

```

调用

```
1 // Program.cs
2 using System;
3 using System.Threading.Tasks;
4 using Tmds.DBus;
5 using bluez.DBus;
6
7 class Program
8 {
9     static async Task Main(string[] args)
10    {
11        try
12        {
13            // 创建与系统总线的连接
14            var connection = new Connection(Address.System);
15            await connection.ConnectAsync();
16
17            // 创建 Adapter1 接口的代理对象
18            var blueZService = connection.CreateProxy<IAdapter1>("org.bluez", "/org/bluez/hci0");
19
20
21            // 使用扩展方法获取属性
22            var address = await blueZService.GetAddressAsync();
23            var name = await blueZService.GetNameAsync();
24            var alias = await blueZService.GetAliasAsync();
25            var powered = await blueZService.GetPoweredAsync();
26            var discovering = await blueZService.GetDiscoveringAsync();
27            var uuids = await blueZService.GetUUIDsAsync();
28
29            // 打印属性值
30            Console.WriteLine("Adapter Properties:");
31            Console.WriteLine($"Address: {address}");
32            Console.WriteLine($"Name: {name}");
33            Console.WriteLine($"Alias: {alias}");
34            Console.WriteLine($"Powered: {powered}");
35            Console.WriteLine($"Discovering: {discovering}");
36            Console.WriteLine($"UUIDs: {string.Join(", ", uuids)}");
37
38            // 开始发现设备
39            await blueZService.StartDiscoveryAsync();
40            Console.WriteLine("Device discovery started...");
41
42            Console.WriteLine("Press Enter to stop discovery...");
43            Console.ReadLine();
44
```

```
45         // 停止发现设备
46         await blueZService.StopDiscoveryAsync();
47         Console.WriteLine("Device discovery stopped.");
48
49         // 断开连接
50         //await connection.DisposeAsync();
51     }
52     catch (Exception ex)
53     {
54         Console.WriteLine($"An error occurred: {ex.Message}");
55     }
56 }
57 }
58 }
```