4

RedOffice 高级开发支持

描述 RedOffice 支持二次开发的系统原理与核心技术，以及语言语言绑定、脚本转接等高级开发支持。

4.1 核心技术 UNO 介绍

UNO（通用网络对象）的目标是为跨编程语言和跨平台边界的网络对象提供环境。UNO 对象可在任何地方运行和通信。UNO 通过提供以下基础框架达到此目标：

UNO 对象在一种称为 UNOIDL（UNO 接口定义语言）的抽象元语言中指定，这种语言与 CORBA IDL 或 MIDL 类似。利用 UNOIDL 规范，可以生成与语言有关的头文件和程序库，用于在目标语言中实现 UNO 对象。在 UNO 对象中，经过编译和绑定程序库的那些对象称为组件。组件必须支持某些基接口才能够在 UNO 环境中运行。

为了在目标环境中实例化组件，UNO 使用了工厂概念。该工厂称为服务管理器，它维护一个注册组件数据库，这些组件可通过名称识别，并可按名称创建。服务管理器可能会要求 Linux 加载和实例化用 C++ 编写的共享对象，也可能会调用本地 Java VM 以实例化 Java 类。这对于开发者来说是透明的，无需考虑组件的实现语言。通信是以独占方式通过 UNOIDL 中指定的接口调用来进行的。

UNO 提供桥，用于在用不同实现语言编写的进程之间以及对象之间发送方法调用和接收返回值。为此，远程桥使用一种特殊的 UNO 远程协议（URP）来支持套接字和管道。桥的两端都必须是 UNO 环境，因此，需要一种特定于语言的 UNO 运行时环境来连接任何受支持语言中的另一个 UNO 进程。这些运行时环境是作为语言绑定提供的。

RedOffice 的大多数对象都能够在 UNO 环境中进行通信。RedOffice 的可编程功能规范称为 RedOffice API。

API 概念 RedOffice API 与语言无关，可以用来指定 RedOffice 的功能。其主要目标是提供访问 RedOffice 功能的 API，使用户能够通过自己的解决方案和新功能来扩展功能，以及使 RedOffice 的内部实现可交换。

RedOffice 的长期目标是将现有的 RedOffice 拆分为若干个小组件，这些小组件可以组合起来提供 RedOffice 的完整功能。这些组件易于管理，它们通过彼此交互来提供高级功能，而且可与提供同样功能的其他实现进行交换，即使这些新实现是通过不同编程语言实现的。达到这一目标后，API、组件和基本概念将提供一个构造工具包，这使 RedOffice 可适应于多种专门解决方案，而不仅仅是一个具有预定义和静态功能的办公套件。本节全面介绍 RedOffice API 背后的概念。API 引用中有若干种其他地方没有的 UNOIDL 数据类型。该引用提供一些抽象规范，有时可能想知道如何将它们映射成可以实际使用的实现。

4.1.1 数据类型

API 引用中的数据类型属于 UNO 类型，需要将这些数据类型映射成可与 RedOffice API 配合使用的任何编程语言中的类型。前面介绍了最重要的 UNO 类型，但没有对 UNO 中的简单类型、接口、属性和服务进行详细介绍。这些类型之间存在多种特殊标志、条件和关系，如果要以专业水平使用 UNO，就需要了解这些标志、条件和关系。

本节从一个想要使用 RedOffice API 的开发者的角度出发，介绍 API 引用中的各种类型。如果有兴趣编写自己的组件，且需要定义新的接口和类型。

4.1.2 简单类型

UNO 提供了一组预定义的简单类型，如下所示：

void　空类型，仅在 any 中用作方法返回类型。

boolean　可以是 true 或 false。

byte　有符号的 8 位整数类型（范围从 -128 到 127，包括上下限）。

short　有符号的 16 位整数类型（范围从 -32768 到 32767，包括上下限）。

unsigned short　无符号的 16 位整数类型（已不再使用）。

long　有符号的 32 位整数类型（范围从 -2147483648 到 2147483647，包括上下限）。

unsigned long　无符号的 32 位整数类型（已不再使用）。

hyper　有符号的 64 位整数类型（范围从 -9223372036854775808 到 9223372036854775807，包括上下限）。

unsigned hyper　无符号的 64 位整数类型（已不再使用）。

float　IEC 60559 单精度浮点类型。

double　IEC 60559 双精度浮点类型。

char　表示单个的 Unicode 字符（更确切地说是单个的 UTF-16 代码单元）。

string　表示 Unicode 字符串（更确切地说是 Unicode 标量值的字符串）。

type　说明所有 UNO 类型的元类型。

any　能够表示其他所有类型值的特殊类型。

4.1.3 Any 类型

特殊类型 Any 可以表示其他所有 UNO 类型的值。在目标语言中，Any 类型需要特殊对待。Java 中提供了 AnyConverter，而 C++中提供了特殊的运算符。

4.1.4 接口

UNO 对象之间的通信基于对象接口。接口分对象外部接口和对象内部接口。

从对象外部看，接口提供对象的一种功能或某个特殊方面。通过发布一组有关对象某个特定方面的操作，接口提供对对象的访问，而无需给出对象的任何内部信息。

接口是一个十分合乎自然的概念，日常生活中经常用到它。接口允许建立彼此匹配的对象，而无须了解对象的内部细节。与标准插座匹配的电源插头，或可以适合于各种尺寸的工作手套，就是一些简单示例。这些对象的正常工作是通过标准化配合作用时所要求的最低条件来实现的。

一个较为高级的示例是简单电视系统的“遥控功能”。遥控是电视系统的功能之一，可以用 XPower 和 XChannel 接口来说明遥控功能。

XPower 接口包含控制电源的函数 turnOn()和 turnOff()，而 XChannel 接口包含控制当前通道的函数 select()、next()和 previous()。这些接口的用户不关心是使用电视机附带的原始遥控，还是某种通用遥控，只要遥控可以实现这些功能即可。只有当遥控无法实现接口承诺的某些功能时，用户才会不满意。

从对象内部或从 UNO 对象实现者的角度来看，接口是抽象规范。RedOffice API 中所有接口的抽象规范都具有一个优点：用户和实现者可签订同意遵守接口规范的合同。一个严格按照规范使用 RedOffice API 的程序将会始终有效，而对于实现者来说，只要遵守合同，就可以对其对象进行任何操作。

UNO 使用 interface 类型来说明 UNO 对象的这些方面。按照约定，所有接口名称都以字母 X 开头，以将接口类型与其他类型区分开来。所有接口类型都必须继承 com.sun.star.uno.XInterface 根接口，可以直接继承，也可以按层次继承结构继承。

接口通过封装对象数据的专用方法（成员函数）来访问该对象的内部数据。方法通常具有一个参数列表和一个返回值，而且可以定义异常以进行智能错误处理。

RedOffice API 中的异常概念与 Java 或 C++中的异常概念类似。没有明确规范，所有操作都可抛出 com.sun.star.uno.RuntimeException，但必须指定其他所有异常。

请看以下两个示例，了解 UNOIDL 表示法中的接口定义。UNOIDL 接口与 Java 接口类似，方法看起来与 Java 方法签名类似。但是，请注意下面示例中方括号内的标志：

```
// base interface for all UNO interfaces
interface XInterface
{
any queryInterface( [in] type aType );
[oneway] void acquire();
[oneway] void release();

};
// fragment of the Interface com.sun.star.io.XInputStream
interface XInputStream: com::sun::star::uno::XInterface
{
      long readBytes( [out] sequence<byte> aData,
                              [in] long nBytesToRead )
```

```
                raises( com::sun::star::io::NotConnectedException,
                        com::sun::star::io::BufferSizeExceededException,
                        com::sun::star::io::IOException);
        ...
};
```

[oneway]标志表示如果基本方法调用系统不支持此功能，则可以以异步方式执行操作。例如，UNO 远程协议（URP）桥是一个支持单向调用的系统。

尽管 UNO oneway 功能的规范和实现没有出现常规问题，但在几种 API 远程使用方案中，oneway 调用会导致 RedOffice 中发生死锁。因此，请不要使用新的 RedOffice UNO API 引入新的 oneway 方法。还存在参数标记，每个参数定义都以方向标志 in、out 或 inout 开头，用来指定参数用途：

- in 指定参数仅用作输入参数。
- out 指定参数仅用作输出参数。
- inout 指定参数可以用作输入和输出参数。

这些参数标记不在 API 引用中出现。方法细节中说明了一个参数实际上是[out]参数还是[inout]参数。包含方法的接口形成服务规范的基础。

4.1.5 服务

我们知道一个单继承接口仅说明对象的一个方面。但是，对象通常会包含多个方面。UNO 使用多继承接口和服务来指定包含多个方面的完整对象。

在第一步中，对象的所有各个方面（通常用单继承接口表示）被组合到一个多继承接口类型中。如果通过调用特定的工厂方法可获得这样的对象，则此步骤即所需的全部操作。指定工厂方法以返回给定的多继承接口类型的值。但是，如果这样的对象在全局组件上下文中可用作常规服务，则在第二步中必须有服务说明。此服务说明将为新样式，能够将服务名称（在组件上下文中通过它可获得服务）映射成给定的多继承接口类型。

为了实现向后兼容，还存在旧式服务，其中包括一组支持某一功能所需的单继承接口和属性。这样的服务还可以包括其他旧式服务。旧式服务的主要缺点在于，它无法明确指出其说明的是通过特定的工厂方法获得的对象（因此将不存在新式服务说明），还是说明在全局组件上下文中可以获得的常规服务（因此将存在新式服务说明）。

从 UNO 对象用户的角度来看，对象提供 API 引用中所述的一项服务，有时甚至提供多项独立的、多继承接口或旧式的服务。可以通过接口中分组的方法调用以及也是经过特殊接口处理的属性来使用服务。由于对功能的访问仅由接口提供，因此，希望使用某个对象的用户无需关心如何实现。从 UNO 对象实现者的角度来看，多继承接口和旧式服务用于定义某项与编程语言无关的功能，且无需提供有关对象内部实现的说明。实现某个对象意味着必须支持所有指定的接口和属性。一个 UNO 对象可以实现多项独立的、多继承接口或旧式服务。有时，实现两项或更多独立的、多继承接口或服务非常有用，因为它们具有相关的功能，或者支持对象的不同视图。说明了接口和服务之间的关系。具有多个接口的旧式服务的与语言无关规范用于实现符合此规范的 UNO 对象。这样的

UNO 对象有时被称为“组件”，尽管该术语用于说明 UNO 环境内的配置实体更为准确。图 7 使用的是直接支持多个接口的旧式服务说明，对于新式服务说明，唯一的区别就是它仅支持一个多继承接口，该接口将继承其他接口。

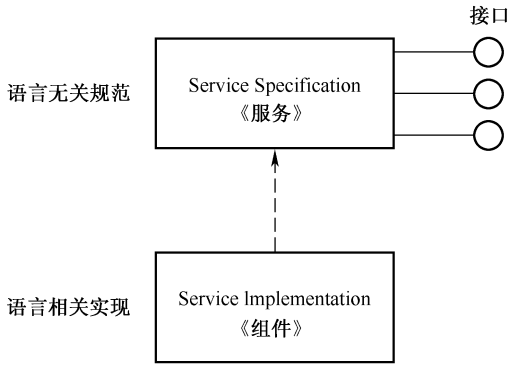

图 7：接口、服务和实现

可以按照服务规范来说明包含电视机和遥控的电视系统的功能。前面所述的 XPower 和 XChannel 接口是服务规范 RemoteControl 中的一部分。新服务 TVSet 包含三个接口：XPower、XChannel 和 XStandby，用于控制电源、频道选择、附加电源功能 standby()以及 timer()功能，如图 8 所示。

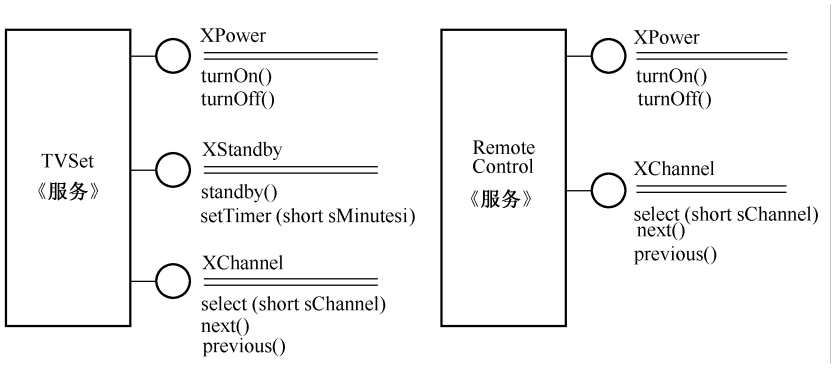

图 8：电视系统规范

4.1.6 引用接口

对某项服务定义中接口的引用意味着要实现此服务必须提供指定的接口。此外，还可以提供可选接口。如果某个多继承接口继承可选接口，或者某个旧式服务包含可选接口，则任何给定的 UNO 对象可以支持此接口，也可以不支持。如果使用某个 UNO 对象的可选接口，通常需要检查 queryIn-terface()的结果是否为 null，并作出相应的反应，否则，您的代码将与不包含可选接口的实现不兼容，并会因为出现空指针异常而结束。以下 UNOIDL 代码段说明了 RedOffice API 中 com.sun.star.text.TextDocument 的旧式服务的规范片段。请注意方括号中的标志 optional，此标志表示接口 XFootnotesSupplier 和 XEndnotesSupplier 为可选接口。

```
// com.sun.star.text.TextDocument
service TextDocument
{
        ...
        interface com::sun::star::text::XTextDocument;
        interface com::sun::star::util::XSearchable;
        interface com::sun::star::util::XRefreshable;
        [optional] interface com::sun::star::text::XFootnotesSupplier;
        [optional] interface com::sun::star::text::XEndnotesSupplier;
        ...
};
```

4.1.7 服务构造函数

新式服务可以拥有构造函数，与接口方法类似：

```
service SomeService: XSomeInterface {
        create1();
        create2([in] long arg1, [in] string arg2);
        create3([in] any... rest);
};
```

在上面的示例中，有三个显式构造函数，名为 create1、create2 和 create3。第一个构造函数没有参数，第二个有两个常规参数，第三个有一个特殊的 rest 参数，该参数可以接受任意数目的 any 值。构造函数参数只能为[in]，rest 参数必须是构造函数的唯一参数，并且必须为 any 类型；另外，与接口方法不同，服务构造函数不指定返回类型。

各个语言绑定将 UNO 构造函数映射成特定语言结构，这些结构可用于客户机代码，在给定组件上下文的情况下可得到服务实例。常规约定（例如，后跟 Java 和 C++语言绑定）将每个构造函数映射成同名的静态方法（resp 函数），将 XComponentContext 作为第一个参数，后跟构造函数中指定的所有参数，并返回一个（适当键入的）服务实例。如果无法得到实例，则会抛出 com.sun.star.uno.DeploymentException。上面的 SomeService 将映射成以下 Java 1.5 类，例如：

```
public class SomeService {
        public static XSomeInterface create1(
                com.sun.star.uno.XComponentContext context) { ... }
        public static XSomeInterface create2(
                com.sun.star.uno.XComponentContext context, int arg1, String arg2) { ... }
        public static XSomeInterface create3(
                com.sun.star.uno.XComponentContext context, Object... rest) { ... }
}
```

服务构造函数还拥有异常规范（"raises (Exception1, ...)"），其处理方法与接口方法的异常规范相同。（如果构造函数没有异常规范，则只能抛出运行时异常，尤其是 com.sun.star.uno.DeploymentException）。

如果新式服务以简写形式 service SomeService: XSomeInterface;来编写，则会有一个隐式构造函数。隐式构造函数的准确行为是特定于语言绑定的，但其名称通常为 create，除 XComponentContext 之外不接受任何参数，并且只能抛出运行时异常。

4.1.8 包含属性

建立 RedOffice API 结构时，设计者发现办公软件环境中的对象具有大量不属于对象结构的属性，更恰当地说，它们似乎是对底层对象的表面更改。同时还发现，并非某种具体类型中的任何对象都具有所有属性。因此，没有为每种属性定义一系列复杂的可选和非可选接口，而是引入了属性的概念。属性是对象中的数据，在普通接口中按名称提供以进行属性访问，接口包含 getPropertyValue()和 setPropertyValue()访问方法。属性这一概念还具有其他优点，有许多信息值得了解。

旧式服务可直接在 UNOIDL 规范中列出支持的属性。property 定义特定类型的成员变量，可以在实现组件时按照特定名称进行访问。可以通过附加标记加入对某个 property 的进一步限制。下面的旧式服务引用了一个接口和三个可选属性。所有已知 API 类型都可以是有效的属性类型：

```
// com.sun.star.text.TextContent
service TextContent
{
    interface com::sun::star::text::XTextContent;
    [optional, property] com::sun::star::text::TextContentAnchorType AnchorType;
    [optional, readonly, property] sequence<com::sun::star::text::TextContentAnchorType> AnchorTypes;
    [optional, property] com::sun::star::text::WrapTextMode TextWrap;
};
```

下面是可能的属性标志：

- optional。实现组件时可以不支持对应的属性。
- readonly。不能使用 com.sun.star.beans.XPropertySet 更改对应的属性值。
- bound。如果任何属性值通过 com.sun.star.beans.XPropertySet 注册，则属性值的更改将通知 com.sun.star.beans.XPropertyChangeListener。
- constrained。属性在其值被更改之前广播一个事件。侦听器有权禁止更改。
- maybeambiguous。某些情况下，可能无法确定属性值，例如，具有不同值的多项选择。
- maybedefault。值可能存储在某个样式工作表中，也可能存储在非对象本身的环境中。
- maybevoid。除属性类型的范围以外，值也可以为空。它与数据库中的空值类似。
- removable。对应的属性是可删除的，用于动态属性。
- transient。如果序列化对象，将不存储对应的属性。

4.1.9 引用其他服务

旧式服务可以包括其他旧式服务。此类引用是可选的。一项服务被另一项服务所包含与实现继

承无关，而仅仅是合并规范。由实现者决定是继承还是授权必需的功能，或者决定是否从头实现必需的功能。

以下 UNOIDL 示例中的旧式服务 com.sun.star.text.Paragraph 包含一项必需服务 com.sun.star.text.TextContent 和五项可选服务。每个 Paragraph 都必须是 TextContent。它同时可以是 TextTable，而且可用于支持段落和字符的格式化属性：

```
// com.sun.star.text.Paragraph
service Paragraph
{
    service com::sun::star::text::TextContent;
    [optional] service com::sun::star::text::TextTable;
    [optional] service com::sun::star::style::ParagraphProperties;
    [optional] service com::sun::star::style::CharacterProperties;
    [optional] service com::sun::star::style::CharacterPropertiesAsian;
    [optional] service com::sun::star::style::CharacterPropertiesComplex;
    ...
};
```

如果上面示例中的所有旧式服务都使用多继承接口类型代替，则结构类似：多继承接口类型 Paragraph 将继承强制接口 TextContent 和可选接口 TextTable、ParagraphProperties 等。

4.1.10 组件中的服务实现

组件是一个共享库或 Java 存档文件，其中包含用 UNO 支持的某种目标编程语言实现的一项或多项服务。这样的组件必须满足基本要求（目标语言不同，通常要求也不同），而且必须支持已实现的服务的规范。这意味着必须实现所有指定的接口和属性。需要在 UNO 运行时系统中注册组件。注册之后，通过在适当的服务工厂对服务实例进行排序以及通过接口访问功能，可以使用所有已实现的服务。

根据 TVSet 和 RemoteControl 服务的示例规范，组件 RemoteTVImpl 可以模拟远程电视系统：

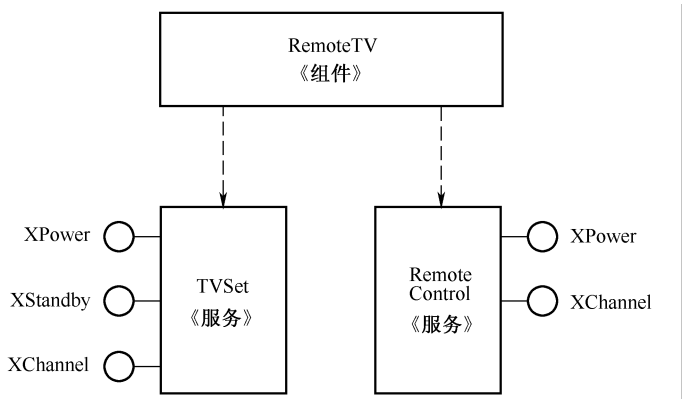

图 9：Remote TVImpl 组件

这样的 RemoteTV 组件可以是 jar 文件或共享库。它包含两个服务实现：TVSet 和 RemoteControl。用全局服务管理器注册 RemoteTV 组件后，用户就可以调用服务管理器的工厂方

法，并可以请求 TVSet 或 RemoteControl 服务。然后，可以通过接口 XPower、XChannel 和 XStandby 使用其功能。以后重新实现具有更好性能或新功能的这些服务时，如果通过加入接口引入新功能，就可以在不破坏现有代码的情况下更换旧组件。

4.1.11 结构

struct 类型定义一条记录中的多个元素。struct 中的元素是结构中具有唯一名称的 UNO 类型。结构的缺点是不封装数据，但缺少 get ()和 set()方法，有助于避免通过 UNO 桥进行方法调用所产生的开销。UNO 支持 struct 类型的单继承。派生的 struct 以递归方式继承父对象以及父对象的父对象中的所有元素。

```
// com.sun.star.lang.EventObject
/** specifies the base for all event objects and identifies the
source of the event.
 */
struct EventObject
{
/** refers to the object that fired the event.
 */
com::sun::star::uno::XInterface Source;

};
// com.sun.star.beans.PropertyChangeEvent
struct PropertyChangeEvent : com::sun::star::lang::EventObject {
      string PropertyName;
      boolean Further;
      long PropertyHandle;
      any OldValue;
      any NewValue;
};
```

RedOffice [OO2.0]的一项新功能是多态结构类型。多态结构类型模板与普通结构类型类似，但它有一个或多个类型参数，并且其成员可以将这些参数作为类型。多态结构类型模板本身不是 UNO 类型，它必须使用实际的类型参数来实例化才能作为一个类型使用。

```
// A polymorphic struct type template with two type parameters:
struct Poly<T,U> {
      T member1;
      T member2;
      U member3;
      long member4;
};
// Using an instantiation of Poly as a UNO type:
interface XIfc { Poly<boolean, any> fn(); };
```

在本例中，Poly<boolean, any>将是一个具有与普通结构类型相同形式的实例化多态结构类型

```
struct PolyBooleanAny {
    boolean member1;
    boolean member2;
    any member3;
    long member4;
};
```

添加多态结构类型主要用于支持丰富的接口类型属性，这些属性与 maybeambiguous、maybedefault 或 maybevoid 属性一样富有表达力（请参阅 com.sun.star.beans.Ambiguous、com.sun.star.beans.Defaulted、com.sun.star.beans.Optional），但这些类型也可能用于其他环境中。

4.1.12 预定义值

API 提供许多预定义值，用作方法参数，或作为方法的返回值。在 UNO IDL 中，预定义值有两种不同的数据类型：常数和枚举。

1．const

const 定义有效 UNO IDL 类型的命名值。值取决于指定的类型，可以是常值（整数、浮点数或字符），另一个 const 类型的标识符或包含以下运算符的算术项：+、-、*、/、~、&、|、%、^、<<、>>。

由于可以在 const 中广泛选择类型和值，因此，const 有时用于生成对合并的值进行编码的位矢量。

```
const short ID = 23;
const boolean ERROR = true;
const double PI = 3.1415;
```

通常情况下，const 定义是常数组的一部分。

2．constants

constants 类型定义的是包含多个 const 值的一个命名组。constants 组中的 const 用组名称加上 const 名称表示。在下面的 UNO IDL 示例中，ImageAlign.RIGHT 指的是值 2：

```
constants ImageAlign {
    const short LEFT = 0;
    const short TOP = 1;
    const short RIGHT = 2;
    const short BOTTOM = 3;
};
```

4.1.13 enum

enum 类型与 C++中的枚举类型相当。它包含一个已排序列表，列表中有一个或多个标识符，代表 enum 类型的各个值。默认情况下，值按顺序编号，以 0 开头，每增加一个新值就增加 1。如果给某个 enum 值指定了一个值，则没有预定义值的所有后续 enum 值都以此指定值为基准来获得值。

```
// com.sun.star.uno.TypeClass
enum TypeClass {
    VOID,
```

```
        CHAR,
        BOOLEAN,
        BYTE,
        SHORT,
        ...
};
enum Error {
        SYSTEM = 10,    // value 10
        RUNTIME,        // value 11
        FATAL,          // value 12
        USER = 30,      // value 30
        SOFT            // value 31
};
```

如果调试中使用了 enum，就应该能够通过操作一个 enum 在 API 引用中的位置来获得该 enum 的数值。但是，不要使用文字数值取代程序中的 enum。

指定和发布某个 enum 类型后，可以确定以后不会扩展该类型，因为这样做会破坏现有的代码。但是，可以将新的 const 值添加到一个常数组。

4.1.14 序列

sequence 类型是一组相同类型的元素，元素的数量可变。在 UNO IDL 中，使用的元素始终引用某个现有的已知类型或者另一个 sequence 类型。sequence 可作为其他所有类型定义中的一个一般类型出现。

```
sequence< com::sun::star::uno::XInterface >
sequence< string > getNamesOfIndex( sequence< long > indexes );
```

模块是命名空间，与 C++中的命名空间或 Java 中的包类似。模块将服务、接口、结构、异常、枚举、类型定义、常数组以及子模块与相关的功能内容或性能组合在一起。使用它们在 API 中指定一致的块，这样可以生成结构良好的 API。例如，模块 com.sun.star.text 包含接口以及其他类型，用于进行文本处理。其他一些典型模块包括 com.sun.star.uno、com.sun.star.drawing、com.sun.star.sheet 和 com.sun.star.table。一个模块内的标识符不会与其他模块内的标识符相冲突，因此，同一名称可能多次出现。API 引用的全局索引说明了确实存在这种情况。

尽管模块似乎与 RedOffice 的各个部分相对应，但是 API 模块与 RedOffice 应用程序 Writer、Calc 和 Draw 之间不存在直接的关系。Calc 和 Draw 中使用模块 com.sun.star.text 的接口。像 com.sun.star.style 或 com.sun.star.document 等模块提供的普通服务和接口不是针对 RedOffice 任何一个部分的。

在 API 引用中看到的模块是通过在模块说明中嵌套 UNO IDL 类型进行定义的。例如，模块 com.sun.star.uno 包含接口 XInterface：

```
module com {
    module sun {
        module star {
            module uno {
                interface XInterface {
```

```
                    ...
                };
            };
        };
    };
};
```

4.1.15 异常

exception 类型表示函数调用程序发生错误。异常类型给出发生的错误类型的基本说明。另外，UNO IDL exception 类型包含的元素给出精确规范和错误详细说明。exception 类型支持继承，这常用于定义错误分层。异常仅用于指出错误，不作为方法参数或返回类型。

UNO IDL 要求所有异常都必须从 com.sun.star.uno.Exception 继承。这是 UNO 运行时的一个前提。

```
// com.sun.star.uno.Exception is the base exception for all exceptions
exception Exception {
    string Message;
    Xinterface Context;
};
// com.sun.star.uno.RuntimeException is the base exception for serious problems
// occuring at runtime, usually programming errors or problems in the runtime environment
exception RuntimeException : com::sun::star::uno::Exception {
};
// com.sun.star.uno.SecurityException is a more specific RuntimeException
exception SecurityException : com::sun::star::uno::RuntimeException {
};
```

异常只能由指定抛出异常的操作来抛出，而 com.sun.star.uno.RuntimeException 则可以随时发生。

UNO 基接口 com.sun.star.uno.XInterface 的方法 acquire()和 release 是上述规则的例外。它们是唯一不会抛出运行时异常的操作。但在 Java 和 C++程序中，不直接使用这些方法，而是由各自的语言绑定进行处理。

4.1.16 Singleton

Singleton 用于指定已命名对象，在一个 UNO 组件上下文的生存期中恰好可以存在一个实例。Singleton 引用一个接口类型，并指定只能在组件上下文中通过使用 Singleton 名称来访问 Singleton 唯一存在的实例。如果该 Singleton 不存在实例，组件上下文将实例化一个新的实例。这种新式 Singleton 的一个示例如下：

```
module com { module sun { module star { module deployment {
singleton thePackageManagerFactory: XPackageManagerFactory;
}; }; }; };
```

各个语言绑定提供了在给定组件上下文的情况下，可得到新式 Singleton 实例的特定语言方法。例如，在 Java 和 C++中，有一个名为 get 的静态方法（resp.函数），该方法将 XComponentContext 作为其唯一的参数，并返回（适当键入的）Singleton 实例。如果无法得到实例，则会抛出

com.sun.star.uno.DeploymentException。

此外，还有旧式 Singleton，这些 Singleton 引用的是（旧式）服务而不是接口。但是，对于旧式服务，语言绑定不提供 get 功能。

4.2 了解 API 引用

4.2.1 规范、实现和实例

API 引用中包含抽象的 API 规范。API 引用的服务说明不是关于已经存在于某处的类。首先有了规范，然后按照规范创建 UNO 实现。甚至对 UNO 必须采用的传统实现也是如此。而且，由于组件开发者可以根据需要自由地实现服务和接口，因而某个特定服务规范与一个真实对象之间不一定要有一对一的关系。真实对象具有的功能可以比服务定义中指定的功能多。例如，如果您在工厂订购了一项服务，或从一个 getter 或 getPropertyValue()方法接收一个对象，指定的功能就会出现，但还可以有其他功能。例如，文本文档模型中有几个接口未包括在 com.sun.star.text.TextDocument 规范中。

由于存在可选的接口和属性，因而不可能通过 API 引用完全理解 RedOffice 中某个对象的给定实例具有什么功能。可选接口和属性对于一个抽象规范来说是恰当的，但它意味着当您离开必需的接口和属性的范围时，引用仅定义允许事物如何工作，而不是它们实际如何工作。

另一点重要的是存在对象实现实际可用的若干入口点。无法通过全局服务管理器实例化 API 引用中存在的每项旧式服务。原因如下：

有些旧式服务需要特定的环境。例如，独立于一个现有的文本文档或其他任何有用的环境来实例化 com.sun.star.text.TextFrame 是没有意义的。这样的服务通常不是由全局服务管理器建立，而是由文档工厂建立，这些文档工厂具有建立在某种具体环境中工作的对象所必需的知识。

这并不意味着永远无法从要插入的全局服务管理器中获取文字框。因此，如果您希望在 API 引用中使用某项服务，就要自问从哪里获取一个支持此服务的实例，并考虑要使用此服务的环境。如果环境是文档，则文档工厂即可创建该对象。

旧式服务不仅仅用于指定可能的类实现。有时用来指定可由其他旧式服务引用的若干组属性。也就是说，有些服务根本没有接口。无法在服务管理器中创建此类服务。少数旧式服务需要特殊对待。例如，不能让服务管理器创建 com.sun.star.text.TextDocument 的实例。必须使用桌面的 com.sun.star.frame.XComponentLoader 接口中的 loadComponent-FromUrl()方法来加载它。

在上面的第一种和最后一种情况中，使用多继承接口类型而不使用旧式服务将是最佳的设计选择，但是提到的服务在 UNO 中多继承接口类型之前即可用。

因此，有时在 API 引用中查找一个需要的功能非常麻烦，因为在实际使用引用之前，需要基本了解功能的工作原理、包含哪些服务、可以从何处获得服务等。本指南的目的在于让您了解 RedOffice 文档模型、数据库集成以及 RedOffice 应用程序本身。

4.2.2 对象复合

接口支持单继承和多继承，而且它们都基于 com.sun.star.uno.XInterface。在 API 引用中，任何接口规范的基本层次结构区域中反映了这一点。如果您查找一个接口，通常需要检查基本层次结构

区域，了解各种支持的方法。例如，如果查找 com.sun.star.text.XText，您会看到两个方法 insertTextContent()和 removeTextContent()，但是，继承的接口提供了另外 9 个方法。同一情况也适用于异常，有时还适用于结构，它们也支持单继承。

API 引用中的服务规范可以包含一个区域，即包含的服务，此区域与上一个区域类似，一项包含的旧式服务可能包括全部服务。但是，包含一项服务这一事实与类继承没有关系。根本没有定义服务实现通过什么方式从技术上包含其他服务（对于 UNO 接口继承也是如此），比如说，通过从基本实现继承，通过聚合，通过其他形式的授权，或只是通过重新实现所有内容。而且，这对于一个 API 用户来说没有什么意义，因为该用户可以依赖于说明功能的可用性，但永远无需依赖于实现的内部细节，例如，哪些类提供此功能、这些类从何处继承以及它们将什么内容授权给其他类。

4.2.3 UNO 概念

现在，您已经深入了解了 RedOffice API 概念，而且了解了 UNO 对象的规范。接下来，我们将探究 UNO，即看看 UNO 对象彼此之间是如何连接和进行通信的。

4.2.4 UNO 进程间连接

不同环境中的 UNO 对象通过进程间的桥进行连接。您可以调用不同进程中的 UNO 对象实例。这是通过以下过程完成的：将方法名称和参数转换成为字节流形式，并将此信息包发送到远程进程，例如，通过套接字连接。本手册中的大多数示例使用进程间的桥来与 RedOffice 进行通信。

本节介绍如何使用 UNO API 来建立 UNO 进程间连接。

4.2.5 侦听模式

本开发者指南中的大多数示例连接到正在运行的 RedOffice，然后在 RedOffice 中执行 API 调用。默认情况下，出于安全考虑，办公软件不对资源进行侦听。这样，就有必要使 RedOffice 对进程间连接资源（例如套接字）进行侦听。目前，侦听可通过两种方法来完成：

通过附加参数启动办公软件：soffice -accept=socket,host=0,port=2002;urp;

在 UNIX shell 上，必须用引号将该字符串引起来，因为 shell 会解释分号“;”。

将上面的同一字符串去掉'-accept='写入配置文件中。您可以编辑文件<OfficePath>/share/registry/data/org/openoffice/Setup.xcu 且可以将标记<prop oor:name="ooSetupConnectionURL"/> 替换为 <prop oor:name="ooSetupConnectionURL"><value>socket,host=localhost,port=2002;urp;StarOffice.ServiceManager </value> </prop>

如果此标记不存在，请在以下标记<node oor:name="Office"/>内添加此标记，此更改会影响整个安装。如果要为网络安装中某个具体用户配置此标记，请在节点 <node oor:name="Office/> 内添加同一标记到用户相关配置目录<OfficePath>/user/registry/data/org/openoffice/的文件 Setup.xcu 中选择所需的过程，并在侦听模式下立即启动 RedOffice。通过在命令行中调用 netstat -a 或-na 来检查是否正在侦听。如果输出与下面显示的结果类似，则表示办公软件正在侦听：

CP <Hostname>:8100 <Fully qualified hostname>: 0 Listening

如果使用-n 选项，netstat 将以数字形式显示地址和端口号。这对于 UNIX 系统有时会很有用，因为 UNIX 中可能会将逻辑名称指定给端口。

如果办公软件没有侦听，可能是由于启动办公软件的连接 URL 参数不正确。检查 Setup.xcu 文件或命令行中是否存在拼写错误，然后重试。

4.2.6 导入 UNO 对象

进程间连接的最常用情况是从导出服务器导入对 UNO 对象的引用。例如，本手册中介绍的大多数 Java 示例获取对 RedOffice ComponentContext 的引用。执行此操作的正确方法是使用 com.sun.star.bridge.UnoUrlResolver 服务。其主要接口 com.sun.star.bridge.XUnoUrlResolver 用以下方法定义：

```
interface XUnoUrlResolver: com::sun::star::uno::XInterface
{
    /** resolves an object on the UNO URL */
    com::sun::star::uno::XInterface resolve( [in] string sUnoUrl )
        raises (com::sun::star::connection::NoConnectException,
                com::sun::star::connection::ConnectionSetupException,
                com::sun::star::lang::IllegalArgumentException);
};
```

传送到 resolve()方法的字符串称为 UNO URL。它必须具有以下格式：

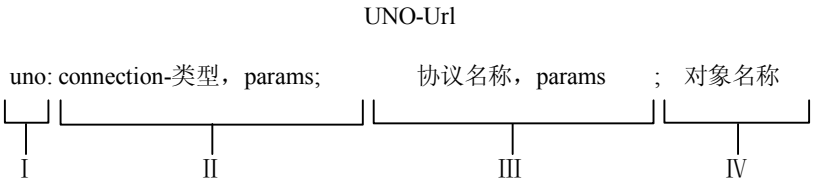

图 10

示例 URL：uno:socket,host=localhost,port=2002;urp;StarOffice.ServiceManager，此 URL 的各个部分如下：

I. URL 模式 uno:。它将 URL 标识为 UNO URL，并将它与其他 URL（例如，http:或 ftp: URL）区分开来。

II. 一个表示用于访问其他进程连接类型的字符串。在此字符串后面可以直接跟一个逗号分隔的名称值对列表，其中，名称和值用一个“=”分隔。连接类型指定传送字节流时使用的传输机制，例如，TCP/IP 套接字或命名管道。

III. 一个表示用于通过建立的字节流连接进行通信的协议类型的字符串。此字符串后面可以跟一个逗号分隔的名称值对列表，用于根据具体需要自定义协议。建议使用的协议为 urp（UNO 远程协议）。下面解释了一些有用的参数。有关完整的规范信息，请参阅 udk.openoffice.org 上名为 UNO-URL 的文档。

IV. 进程必须按照明确名称明确地导出具体对象。不能访问任意的 UNO 对象（但在 CORBA 中，则可通过 IOR 执行此操作）。

以下示例说明如何使用 UnoUrlResolver 导入对象：

```
(ProfUNO/InterprocessConn/UrlResolver.java)
    XComponentContext xLocalContext =
            com.sun.star.comp.helper.Bootstrap.createInitialComponentContext(null);
                // initial serviceManager
     XMultiComponentFactory xLocalServiceManager = xLocalContext.getServiceManager();
              // create a URL resolver
     Object urlResolver = xLocalServiceManager.createInstanceWithContext(
            "com.sun.star.bridge.UnoUrlResolver", xLocalContext);
     // query for the XUnoUrlResolver interface
     XUnoUrlResolver xUrlResolver =
            (XUnoUrlResolver) UnoRuntime.queryInterface(XUnoUrlResolver.class, urlResolver);
     // Import the object
     Object rInitialObject = xUrlResolver.resolve(
            "uno:socket,host=localhost,port=2002;urp;StarOffice.ServiceManager");
      // XComponentContext
     if (null != rInitialObject) {
            System.out.println("initial object successfully retrieved");
     } else {
            System.out.println("given initial-object name unknown at server side");
     }
```

使用 UnoUrlResolver 时有一些限制，您不能：

- 在进程间桥由于任何原因终止时得到通知。
- 关闭底层进程间连接。
- 将一个本地对象作为初始对象提供给远程进程。

这些问题可通过底层 API 得到解决。

4.2.7 进程间桥的属性

整个桥是线程安全的，并且允许多个线程执行远程调用。桥内的分发线程不会阻塞，因为它从不执行调用，而是将请求传送到工作线程。

同步调用通过连接发送请求，并使请求的线程等待应答。所有具有返回值（即 out 参数）或抛出非 RuntimeException 异常的调用必须是同步的。

异步（或 oneway）调用通过连接发送请求并立即返回，而不等待应答。目前在 IDL 接口中，使用[oneway]修饰符来指定一个请求是同步还是异步。

同步请求可以保证线程标识。当进程 A 调用进程 B，而进程 B 又调用进程 A 时，进程 A 中等待的同一线程将接管新的请求。这就避免了再次锁定同一互斥体时出现的死锁。对于异步请求，不可能发生这样的情况，因为进程 A 中没有等待的线程。这类请求在新的线程中执行。因而保证了两个进程之间的一系列调用。如果将来自进程 A 的两个异步请求发送到进程 B，第二个请求将会等待，直到完成第一个请求。

尽管远程桥支持异步调用，但此功能在默认情况下被禁用。每个调用都是同步执行。UNO 接口方法的单向标志将被忽略。但是，桥可以在启用单向功能的模式下启动，因此可以像异步调用那

样执行标有[oneway]修饰符的调用。为此，必须通过',Negotiate=0,ForceSynchronous=0'扩展远程桥两端连接字符串中的协议部分。例如：

soffice "-accept=socket,host=0,port=2002;urp,Negotiate=0,ForceSynchronous=0;"

用于启动办公软件，而

"uno:socket,host=localhost,port=2002;urp,Negotiate=0,ForceSynchronous=0;StarOffic e.ServiceManager"

作为连接到办公软件的 UNO URL。

4.2.8 打开连接

如前面章节所述，使用 UnoUrlResolver 导入 UNO 对象这一方法有些不足之处。

UnoUrlResolver 的下一层为处理进程间连接提供了非常大的灵活性。

UNO 进程间桥建立在 com.sun.star.connection.XConnection 接口上，此接口封装一个可靠的双向字节流连接（如 TCP/IP 连接）。

```
interface XConnection: com::sun::star::uno::XInterface
{
    long read( [out] sequence < byte > aReadBytes , [in] long nBytesToRead )
        raises( com::sun::star::io::IOException );
    void write( [in] sequence < byte > aData )
        raises( com::sun::star::io::IOException );
    void flush() raises( com::sun::star::io::IOException );
    void close() raises( com::sun::star::io::IOException );
    string getDescription();
};
```

建立进程间连接有多种不同的机制，其中多数机制遵循类似的模式，一个进程对资源进行侦听，并等待一个或多个进程连接到此资源。

此模式是通过导出 com.sun.star.connection.XAcceptor 接口的 com.sun.star.connection.Acceptor 和导出 com.sun.star.connection.XConnector 接口的 com.sun.star.connection.Connector 服务抽出概念。

```
interface XAcceptor: com::sun::star::uno::XInterface
{
    XConnection accept( [in] string sConnectionDescription )
        raises( AlreadyAcceptingException,
                ConnectionSetupException,
                com::sun::star::lang::IllegalArgumentException);

    void stopAccepting();
};
interface XConnector: com::sun::star::uno::XInterface
{
    XConnection connect( [in] string sConnectionDescription )
        raises( NoConnectException,ConnectionSetupException );
};
```

侦听进程使用 Acceptor 服务，而主动连接服务使用 Connector 服务。方法 accept()和 connect()以参数的方式获取连接字符串。它是 UNO URL 中的连接部分（位于 uno:和;urp 之间）。

连接字符串包含一个连接类型，后跟一个逗号分隔的名称值对列表。表 4-1 所示为默认情况下支持的连接类型。

表 4-1

连接类型		
套接字	可靠的 TCP/IP 套接字连接	
	参数	说明
	host	要侦听/连接资源的主机名或 IP 编号。可以是本地主机。在 Acceptor 字符串中，它可以是 0（'host=0'），这意味着接受所有可用网络接口
	port	要侦听/连接的 TCP/IP 端口号
	tcpNoDelay	对应于套接字选项 tcpNoDelay。对于 UNO 连接，应将此参数设置为 1（这是不默认值，必须明确添加此参数）。如果使用默认值（0），在某些调用组合中可能会发生 200 毫秒的延迟
管道	命名管道（使用共享内存）。此进程间连接类型比套接字连接稍微快一些，并仅适用于两个进程位于同一台计算机的情况。默认情况下，此连接类型不适用于 Java，因为 Java 不直接支持命名管道	
	参数	说明
	name	命名管道的名称。一台计算机一次只能接受一个进程名称

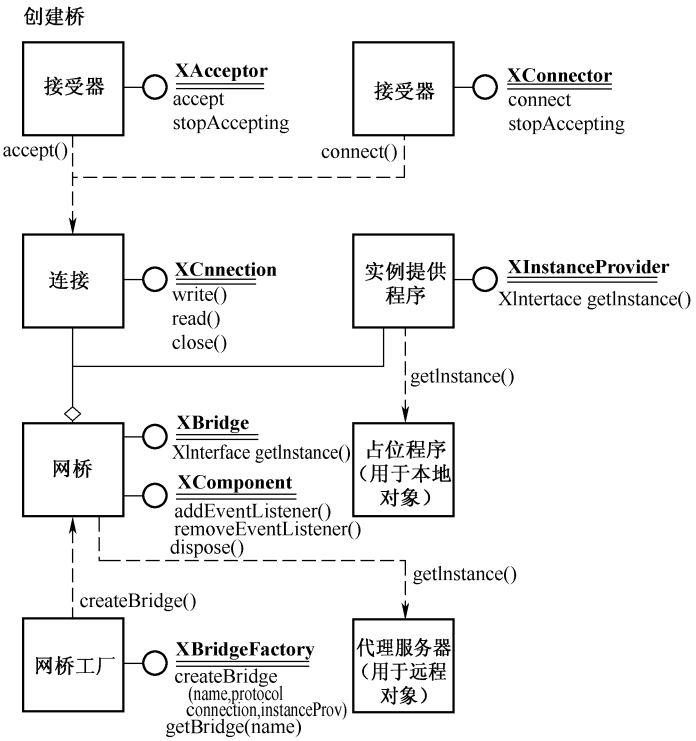

图 10：启动 UNO 进程间桥所需的服务交互（接口已经被简化）

XConnection 实例现在可以用来在连接上建立 UNO 进程间桥，而不管连接是使用 Connector 还是使用 Acceptor 服务（或别的方法）建立的。为此，必须实例化服务 com.sun.star.bridge.BridgeFactory。它支持 com.sun.star.bridge.XBridgeFactory 接口。

```
interface XBridgeFactory: com::sun::star::uno::XInterface
{
    XBridge createBridge(
                [in] string sName,
                [in] string sProtocol ,
                [in] com::sun::star::connection::XConnection aConnection ,
                [in] XInstanceProvider anInstanceProvider )
            raises ( BridgeExistsException , com::sun::star::lang::IllegalArgumentException );
    XBridge getBridge( [in] string    sName );
    sequence < XBridge > getExistingBridges( );
};
```

BridgeFactory 服务管理所有 UNO 进程间连接。createBridge()方法创建一个新桥：可以使用 sName 参数给桥指定一个明确的名称。然后可以使用此名称通过 getBridge()方法来获取桥。这使两个独立的代码段可以共享同一进程间桥。如果您使用已经存在的进程间桥名称来调用 createBridge()，就会抛出 BridgeExistsException。如果传送的是一个空字符串，通常将建立一个新的匿名桥，通过 getBridge()不会获取到此桥，而且不会抛出 BridgeExistsException。

第二个参数指定连接使用的协议。目前，仅支持'urp'协议。在 UNO URL 中，此字符串由两个“;”分隔。urp 字符串后面可以跟一个逗号分隔的名称值对列表，这些名称值对说明桥协议的属性。udk.openoffice.org 上提供了 urp 规范。

第三个参数是 XConnection 接口，因为它是通过 Connector/Acceptor 服务获取的。

第四个参数是一个 UNO 对象，该对象支持 com.sun.star.bridge.XInstanceProvider 接口。

如果不想将一个本地对象导出到远程进程，此参数可以是一个空引用。

```
interface XInstanceProvider: com::sun::star::uno::XInterface
{
    com::sun::star::uno::XInterface getInstance( [in] string sInstanceName )
            raises ( com::sun::star::container::NoSuchElementException );
};
BridgeFactory 返回一个 com.sun.star.bridge.XBridge 接口。
interface XBridge: com::sun::star::uno::XInterface
{
    XInterface getInstance( [in] string sInstanceName );
    string getName();
    string getDescription();
};
```

XBridge.getInstance()方法通过远程桥获取一个初始对象。本地 XBridge.getInstance()调用作为 XInstanceProvider.getInstance()调用到达远程进程。返回的对象可由字符串 sInstanceName 控制。它完全取决于 XInstanceProvider 的实现，即 XInstanceProvider 返回的对象。

可以从 XBridge 接口中查询 com.sun.star.lang.XComponent 接口，该接口将一个 com.sun.star.lang.XEventListener 添加到桥。当底层连接关闭时，就会终止此侦听器（请参阅前面的内容）。您也可以明确地调用 XComponent 接口中的 dispose()，来关闭底层连接并启动桥关闭过程。

4.2.9 关闭连接

如果出现以下原因，进程间连接将会关闭：桥不再使用。释放远程对象的所有代理服务器以及本地对象的所有占位程序后，进程间桥将关闭连接。这是远程桥自我析构的一般方法。进程间桥的用户不需要直接关闭进程间连接，它是自动进行的。如果在 Java 中实现一个通信进程，VM 结束最后的代理服务器/占位程序后才会关闭桥。因此，没有指定关闭进程间桥的时间。

通过调用 dispose()方法来直接废止进程间桥。

由于进程间桥实现中发生故障导致封送处理/取消封送处理发生错误，或者某个进程中的 IDL 类型不可用。除第一种原因之外，其他所有连接关闭都会启动一个进程间桥关闭过程。所有暂挂同步请求终止时都会出现 com.sun.star.lang.DisposedException，它是从 com.sun.star.uno.RuntimeException 派生的。已废止的代理服务器上启动的每个调用都会抛出 DisposedException。所有线程都不再使用桥后（原来远程进程中可能存在一个同步调用），桥将明确释放到本地进程中原始对象的所有占位程序，这些占位程序以前由原来的远程进程所有。然后，桥使用 com.sun.star.lang.XEventListener 将已废止状态通知给所有已注册的侦听器。下面的示例代码适用于识别连接的客户端，该代码显示了如何使用此机制。释放最后一个代理服务器后，桥自身被析构。遗憾的是，无法区分列出的各种错误状况。

4.3 服务管理器与组件上下文

本节讨论用于与 RedOffice（以及与任何 UNO 应用程序）连接的根对象——服务管理器。根对象用作每个 UNO 应用程序的入口点，并在实例化过程中被传送到每个 UNO 组件。

目前存在两种不同的获取根对象的概念。RedOffice 6.0 和 OpenOffice.org 1.0 使用旧概念。较新的版本或产品修补程序使用新概念，并仅针对兼容性问题提供旧概念。首先，我们来了解旧概念，即本指南底层 RedOffice 实现的主要部分中使用的服务管理器。其次，我们将介绍组件上下文（它是新概念），并解释移植路径。

4.3.1 服务管理器

com.sun.star.lang.ServiceManager 是每个 UNO 应用程序中的主要工厂。它按服务名称实例化服务，以枚举某项具体服务的所有实现，并在运行时添加或删除某项具体服务的工厂。实例化时，服务管理器被传送到每个 UNO 组件。

1. XMultiServiceFactory 接口

服务管理器的主要接口是 com.sun.star.lang.XMultiServiceFactory 接口。它提供三个方法：createInstance()、createInstanceWithArguments()和 getAvailableServiceNames()。

```
interface XMultiServiceFactory: com::sun::star::uno::XInterface
{
    com::sun::star::uno::XInterface createInstance( [in] string aServiceSpecifier )
        raises( com::sun::star::uno::Exception );
    com::sun::star::uno::XInterface createInstanceWithArguments(
```

```
                [in] string ServiceSpecifier,
                [in] sequence<any> Arguments )
            raises( com::sun::star::uno::Exception );
    sequence<string> getAvailableServiceNames();
    };
```

createInstance()返回一个默认构造的服务实例。返回的服务保证至少支持所有接口，这些接口是通过请求的服务名称来指定的。现在，可以从返回的 XInterface 引用中查询服务说明中指定的接口。

使用服务名称时，调用程序不会对实例化哪个具体实现产生任何影响。如果某项服务存在多个实现，服务管理器就可以自由决定采用哪个实现。这一般不会对调用程序产生影响，因为每个实现都确实履行服务合同。性能或其他细节可能会有所不同。因此，也可以传送实现名称，而不是服务名称；但是，不建议这样做，因为实现名称可能会更改。

如果服务管理器没有为某个请求提供实现，将会返回一个空引用，因此必须进行检查。实例化时，可能会抛出所有 UNO 异常。有些可能会在要实例化的服务的规范中说明，例如，由于具体实现的配置不正确。另一个原因可能是缺少某个具体桥，例如，Java-C++桥，以防从 C++ 代码实例化 Java 组件。

createInstanceWithArguments()使用附加参数实例化服务。一项服务表示在通过支持 com.sun.star.lang.XInitialization 接口进行实例化时需要参数。服务定义应该说明序列中每个元素的意义。可能有些服务必须使用参数来实例化。

getAvailableServiceNames()返回服务管理器确实支持的所有服务名称。

2. XContentEnumerationAccess 接口

com.sun.star.container.XContentEnumerationAccess 接口允许创建具体服务名称的所有实现的枚举。

```
interface XContentEnumerationAccess: com::sun::star::uno::XInterface
{
    com::sun::star::container::XEnumeration createContentEnumeration( [in] string aServiceName );

    sequence<string> getAvailableServiceNames();

};
```

createContentEnumeration()方法返回一个 com.sun.star.container.XEnumeration 接口。请注意，如果枚举为空，此方法可能会返回空引用。

```
interface XEnumeration: com::sun::star::uno::XInterface
{
    boolean hasMoreElements();

    any nextElement()
        raises( com::sun::star::container::NoSuchElementException,
                com::sun::star::lang::WrappedTargetException );

};
```

在上面的示例中，方法 Xenumeration.nextElement()返回的 any 中，包含一个与此特定服务的每个实现对应的 com.sun.star.lang.XSingleServiceFactory 接口。例如，您可以遍历某项具体服务的所有实现，并检查附加的已实现服务的每个实现，XSingleServiceFactory 接口就提供这样的方法。利用此方法，可以实例化一个服务丰富的功能实现。

4.3.2 XSet 接口

com.sun.star.container.XSet 接口允许在运行时将 com.sun.star.lang.XSingleServiceFactory 或 com.sun.star.lang.XSingleComponentFactory 实现插入到服务管理器或从中删除而不保存这些更改。办公软件应用程序终止时，所有更改将失效。对象还必须支持 com.sun.star.lang.XServiceInfo 接口，用于提供有关组件实现的实现名称和所支持服务的信息。

在开发阶段可能会对此功能特别感兴趣。例如，您可以连接到正在运行的办公软件，在服务管理器中插入新的工厂，以及在不需要提前注册的情况下直接实例化新服务。

前面将服务管理器描述为主要工厂，它被传送到每个新实例化的组件。部署应用程序后，一个组件通常需要更多可以交换的功能或信息。在这种环境中，服务管理器方法具有局限性。

因此，创建了组件上下文的概念。将来，此概念将会成为每个 UNO 应用程序的主要对象。它基本上是一个提供命名值的只读容器，其中一个命名值就是服务管理器。实例化时，组件上下文被传送到一个组件。这可以理解为一种组件生存环境（关系类似于 shell 环境变量与可执行程序）。

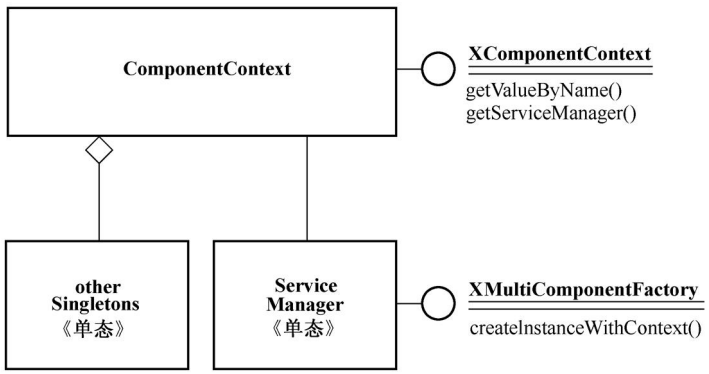

图 11：ComponentContext 与 ServiceManager

组件上下文仅支持 com.sun.star.uno.XComponentContext 接口。

```
// module com::sun::star::uno
interface XComponentContext : XInterface
{
    any getValueByName( [in] string Name );
    com::sun::star::lang::XMultiComponentFactory getServiceManager();
};
```

getValueByName() 方法返回一个命名值。getServiceManager() 是一种获取命名值/singleton/com.sun.star.lang.theServiceManager 的便捷方法。它返回 ServiceManagersingleton，因为多数组件需要访问服务管理器。组件上下文至少提供三种命名值：在 RedOffice 6.0 PP2 中，只有 ServiceManager singleton。从 RedOffice 7 开始，增加了一个 singleton/singletons/com.sun.star.util.theMacroExpander，此 singleton 可用于在配置文件中扩展宏。在 IDL 引用中可以找到其他可能的 singleton。

- 实现属性（尚未定义）

这些属性自定义某个具体实现，并且在每个组件的模块说明中指定。模块说明是某个模块（DLL 或 jar 文件）基于 XML 的说明，其中包含一个或多个组件的一般说明。

- 服务属性（尚未定义）

这些属性可以自定义与实现无关的某项具体服务，而且是在一项服务的 IDL 规范中指定的。请注意，服务环境属性不同于服务属性。服务环境属性无法更改，而且对于所有共享同一组件上下文的服务实例都是相同的。每个实例可以具有不同的服务属性，而且在运行时可以通过 XPropertySet 接口更改服务属性。

请注意，在上述模式中，ComponentContext 引用了服务管理器，而不是相反的情况。

除了前面讨论的接口以外，ServiceManager 还支持 com.sun.star.lang.XMultiComponentFactory 接口。

```
interface XMultiComponentFactory : com::sun::star::uno::XInterface
{
com::sun::star::uno::XInterface createInstanceWithContext(
            [in] string aServiceSpecifier,
            [in] com::sun::star::uno::XComponentContext Context )
            raises (com::sun::star::uno::Exception);
    com::sun::star::uno::XInterface createInstanceWithArgumentsAndContext(
            [in] string ServiceSpecifier,
            [in] sequence<any> Arguments,
            [in] com::sun::star::uno::XComponentContext Context )
            raises (com::sun::star::uno::Exception);
    sequence< string > getAvailableServiceNames();
};
```

它替代 XMultiServiceFactory 接口。对于两种对象建立方法，它提供了一个附加的 XComponentContext 参数。此参数使调用程序能够定义组件新实例可以接收的组件上下文。多数组件使用其初始组件上下文来实例化新组件，这样就可以进行环境传播。

图 12 所示为环境传播。用户可能需要一个专用组件来获得自定义的环境。这样，用户只需包装一个现有的环境，即可建立一个新环境。用户覆盖所需的值，并将自己不感兴趣的属性授权给原始 C1 环境。用户定义实例 A 和 B 收到哪个环境。实例 A 和 B 将其环境传播到它们创建的每个新对象。这样，用户建立了两棵实例树，第一棵树完全使用环境 Ctx C1，而第二棵树则使用 Ctx C2。

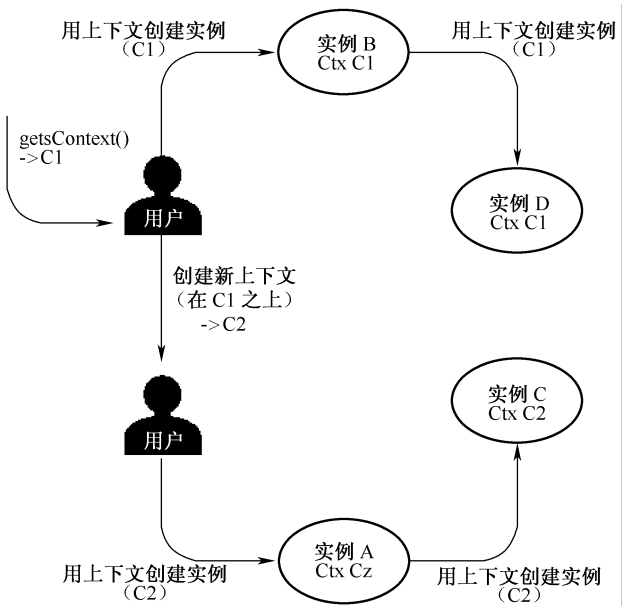

图 12：环境传播

4.4 可用性

可以在 RedOffice 4.5 中使用组件上下文的最终 API。使用此 API 取代服务管理器部分中说明的 API。目前，组件上下文无法进行永久存储，因此，无法将命名值添加到一个已部署 RedOffice 的环境中。在发布未来版本之前，新 API 目前没有其他优点。

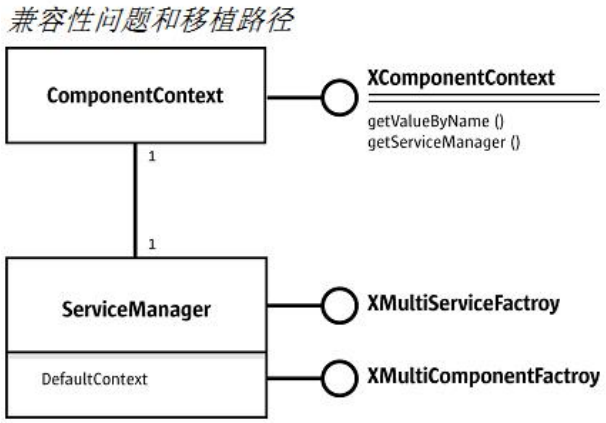

图 13：纯服务管理器与组件上下文概念之间的折中

如前所述，办公软件内目前同时使用了这两个概念。ServiceManager 支持 com.sun.star.lang.XMultiServiceFactory 和 com.sun.star.lang.XMultiComponentFactory 接口。将 XMultiServiceFactory 接口调用授权给 XMultiComponentFactory 接口。服务管理器使用自己的 XComponentContext 引用来填充未设定的参数。可以用'DefaultContext'通过 XProper-tySet 接口来获

取 ServiceManager 的组件上下文。

```
// Query for the XPropertySet interface.
// Note xOfficeServiceManager is the object retrieved by the
// UNO URL resolver
XPropertySet xPropertySet = (XPropertySet)
UnoRuntime.queryInterface(XPropertySet.class, xOfficeServiceManager);

// Get the default context from the office server.
Object oDefaultContext = xpropertysetMultiComponentFactory.getPropertyValue("DefaultContext");

// Query for the interface XComponentContext.
xComponentContext = (XComponentContext) UnoRuntime.queryInterface(
XComponentContext.class, objectDefaultContext);
```

此解决方案允许使用同一服务管理器实例，不管此实例使用旧式 API 还是新式 API。将来，所有 RedOffice 代码将仅使用新 API。但是，还将保留旧 API，目的是为了确保兼容性。

4.5 使用 UNO 接口

每个 UNO 对象都必须从接口 com.sun.star.uno.XInterface 继承。使用一个对象之前，需要了解其使用方式以及生存期。通过将 XInterface 指定为每个 UNO 接口的基接口，UNO 为对象通信打下了基础。由于历史原因，XInterface 的 UNOIDL 说明中列出了与 C++（或二进制 UNO）语言绑定中与 XInterface 有关的功能；其他语言绑定根据不同的机制提供类似的功能：

```
// module com::sun::star::uno
interface XInterface
{
    any queryInterface( [in] type aType );
    [oneway] void acquire();
    [oneway] void release();
};
```

acquire()和 release()方法通过引用计数来处理 UNO 对象的生存期。无论何时引用 UNO 对象，所有当前语言绑定都内部处理 acquire()和 release()。

queryInterface()方法获取该对象导出的其他接口。如果该对象支持类型参数指定的接口，调用程序就会请求实现该对象。type 参数必须表示一个 UNO 接口类型。调用可能返回请求类型的接口引用，或返回一个空 any。在 C++或 Java 中，只测试结果是否为 null。

当请求服务管理器创建一个服务实例时，我们无意中遇到了 Xinterface：

```
XComponentContext xLocalContext =
        com.sun.star.comp.helper.Bootstrap.createInitialComponentContext(null);

// initial serviceManager
XMultiComponentFactory xLocalServiceManager = xLocalContext.getServiceManager();

// create a urlresolver
```

```
        Object urlResolver   = xLocalServiceManager.createInstanceWithContext(
            "com.sun.star.bridge.UnoUrlResolver", xLocalContext);
```

XmultiComponentFactory 的 IDL 规范显示：

```
// module com::sun::star::lang
interface XMultiComponentFactory : com::sun::star::uno::XInterface
{
    com::sun::star::uno::XInterface createInstanceWithContext(
            [in] string aServiceSpecifier,
            [in] com::sun::star::uno::XComponentContext Context )
        raises (com::sun::star::uno::Exception);
    ...
}
```

上面的代码说明的是 createInstanceWithContext()提供给定服务的一个实例，但它仅返回一个 com.sun.star.uno.XInterface。然后，通过 Java UNO 绑定将其映射成 java.lang.Object。

要访问某项服务，需要知道该服务导出哪些接口，可从 IDL 引用中获得此信息。例如，对于 com.sun.star.bridge.UnoUrlResolver 服务，您会了解到：

```
// module com::sun::star::bridge
service UnoUrlResolver: XUnoUrlResolver;
```

这意味着您在服务管理器上订购的服务必须支持 com.sun.star.bridge.XUnoUrlResolver。接下来，查询此接口的返回对象：

```
// query urlResolver for its com.sun.star.bridge.XUnoUrlResolver interface
XUnoUrlResolver xUrlResolver = (XUnoUrlResolver)
    UnoRuntime.queryInterface(UnoUrlResolver.class, urlResolver);
// test if the interface was available
if (null == xUrlResolver) {
    throw new java.lang.Exception(
        "Error: UrlResolver service does not export XUnoUrlResolver interface");
}
// use the interface
Object remoteObject = xUrlResolver.resolve(
    "uno:socket,host=0,port=2002;urp;StarOffice.ServiceManager");
```

该对象决定是否返回接口。如果该对象不返回某项服务中指定的必需接口，则会出现一个错误。获取接口引用时，根据接口规范调用此引用。在服务管理器上实例化每项服务时，可以遵循此策略，以取得成功。利用此方法，不仅可以通过服务管理器来获取 UNO 对象，而且可以通过一般接口调用来获取 UNO 对象：

```
// Module com::sun::star::text
interface XTextRange: com::sun::star::uno::XInterface
{
    XText getText();
    XTextRange getStart();
    ....
};
```

返回的接口类型是在操作中指定的，因此可以直接在返回的接口上启动调用。通常，返回的是

一个实现多个接口的对象，而不是实现某个具体接口的对象。

然后，就可以查询在给定的旧式服务中指定的其他接口的返回对象，在这里，给定的旧式服务为 com.sun.star.drawing.Text。

UNO 有许多普通接口。例如，接口 com.sun.star.frame.XComponentLoader：

```
// module com::sun::star::frame
interface XComponentLoader: com::sun::star::uno::XInterface
{
    com::sun::star::lang::XComponent loadComponentFromURL( [in] string aURL,
                [in] string aTargetFrameName,
                [in] long nSearchFlags,
                [in] sequence<com::sun::star::beans::PropertyValue> aArgs )
            raises( com::sun::star::io::IOException,
                    com::sun::star::lang::IllegalArgumentException );
};
```

4.6 属性

属性是属于某项服务的名称值对，用于确定服务实例中某个对象的属性。属性通常用于非结构属性，如对象的字体、大小或颜色，而 get 和 set 方法则用于像父对象或子对象这样的结构属性。

几乎在所有情况下，com.sun.star.beans.XPropertySet 都用于按名称访问的属性。其他接口，例如 com.sun.star.beans.XPropertyAccess 或 com.sun.star.beans.XMultiPropertySet，前者用于同时设置和获取所有属性，后者用于同时访问多个指定属性。这对远程连接非常有用。另外，还有用于按数字 ID 访问属性的接口，如 com.sun.star.beans.XFastPropertySet。

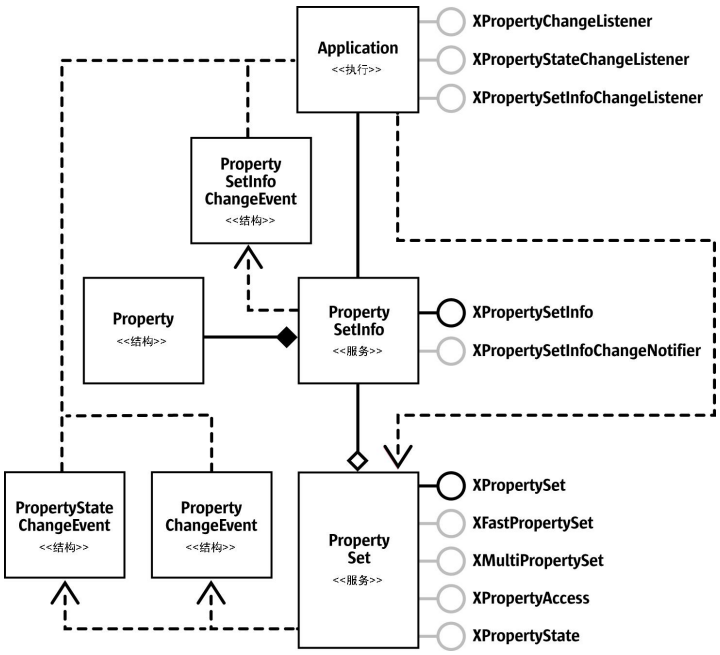

图 13：属性

4.7 UNO 语言绑定

本节介绍如何将 UNO 映射到各种编程语言或组件模型。此语言绑定有时称为 UNO 运行时环境（URE）。每个 URE 都需要符合前面各章节中所述的规范。本节还将说明 UNO 服务和接口的使用。

以下各节提供了下列主题的详细信息：

- 将所有 UNO 类型映射成编程语言类型。
- 将 UNO 异常处理映射到编程语言。
- 映射基本的对象功能（查询接口、对象生存期、对象标识）。
- 引导服务管理器。其他编程语言特有的材料（如 C++ UNO 中的核心程序库）。

目前支持 Java、C++、RedOffice Basic 以及 Windows 32 平台上支持 MS OLE Automation 或公共语言基础结构（CLI）的所有语言，将来可能会扩大支持的语言绑定数量。

4.7.1 Java 语言绑定

Java 语言绑定使开发者可以将 Java 或 UNO 组件用于客户机程序。由于可以无缝地与 UNO 桥进行交互，Java 程序可以访问用其他语言编写并用不同编译器构建的组件以及远程对象。

Java 提供了可以在客户机程序或组件实现中使用的多个类。但是，当需要与其他 UNO 对象交互时，请使用 UNO 接口，因为桥仅识别这些接口，并且可以将这些接口映射到其他环境。

要从客户机程序控制办公软件，客户机需要安装 Java 1.3（或更高版本）、可用的套接字端口以及以下 jar 文件：juh.jar、jurt.jar、ridl.jar 和 unoil.jar。服务器端无需安装 Java 程序。

使用 Java 组件时，办公软件已安装 Java 支持。同时确保已启用 Java：可以通过设置工具－选项－RedOffice－安全性对话框中的一个开关选项来达到此目的。安装 RedOffice 时应该已经安装了所有必需的 jar 文件。

“Java UNO 引用”对“Java UNO 运行时”进行了说明，可以在 RedOffice 软件开发工具包（SDK）中或在 api.openoffice.org 上找到“Java UNO 引用”。

- 序列类型的映射

带有给定组件类型的 UNO 序列类型被映射成带有对应组件类型的 Java 数组类型。

- UNO sequence<long>被映射成 Java int[]。
- UNO sequence< sequence<long> >被映射成 Java int[][]。

只有对这些 Java 数组类型的非空引用才是有效的。通常情况下，也可以使用对其他 Java 数组类型（与给定的数组类型是指定兼容的）的非空引用，但这样做会造成 java.lang.ArrayStoreException。在 Java 中，一个数组的最大长度是有限的，因此，如果在 Java 语言绑定的环境中使用的 UNO 序列太长将会出错。

1. 枚举类型的映射

UNO 枚举类型被映射为同名的公共最终 Java 类，它从 com.sun.star.uno.Enum 类派生而来。只有对生成的最终类的非空引用才是有效的。

UNO	Java
void	void
boolean	boolean
byte	byte
short	short
unsigned short	short
long	int
unsigned long	int
hyper	long
unsigned hyper	long
float	float
double	double
char	char
string	java.lang.String
type	com.sun.star.uno.Type
any	java.lang.Object/com.sun.star.uno.Any

图 14

此基类 com.sun.star.uno.Enum 声明存储实际值的保护成员、初始化值的保护构造函数以及获取实际值的公共 getValue()方法。生成的最终类拥有一个保护构造函数以及一个公共方法 getDefault()，后者返回一个实例，此实例以第一个枚举成员的值为默认值。对于 UNO 枚举类型的各成员，对应的 Java 类声明给定 Java 类型的一个公共静态成员，该类型用 UNO 枚举成员的值进行初始化。枚举类型的 Java 类有一个附加的公共方法 fromInt()，此方法返回包含指定值的实例，如以下 IDL 定义用于 com.sun.star.uno.TypeClass：

```
module com { module sun { module star { module uno {
enum TypeClass {
    INTERFACE,
    SERVICE,
    IMPLEMENTATION,
    STRUCT,
    TYPEDEF,
    ...
};
}; }; }; };
```

被映射成：

```
package com.sun.star.uno;
public final class TypeClass extends com.sun.star.uno.Enum {
    private TypeClass(int value) {
        super(value);
    }
    public static TypeClass getDefault() {
        return INTERFACE;
```

```
        }
        public static final TypeClass INTERFACE = new TypeClass(0);
        public static final TypeClass SERVICE = new TypeClass(1);
        public static final TypeClass IMPLEMENTATION = new TypeClass(2);
        public static final TypeClass STRUCT = new TypeClass(3);
        public static final TypeClass TYPEDEF = new TypeClass(4);
        ...
        public static TypeClass fromInt(int value) {
            switch (value) {
            case 0:
                return INTERFACE;
            case 1:
                return SERVICE;
            case 2:
                return IMPLEMENTATION;
            case 3:
                return STRUCT;
            case 4:
                return TYPEDEF;
            ...
            }
        }
    }
```

2. 结构类型的映射

普通的UNO结构类型被映射成一个同名的公共Java类。只有对这种类的非空引用才是有效的。UNO 结构类型的每个成员都被映射成相同名称、对应类型的公共字段。该类提供一个用默认值初始化所有成员的默认构造函数，以及一个获取所有结构成员确切值的构造函数。如果一个普通结构类型继承另一个结构类型，则生成的类是被继承结构类型的类的子类。

```
module com { module sun { module star { module chart {
struct ChartDataChangeEvent: com::sun::star::lang::EventObject {
    ChartDataChangeType Type;
    short StartColumn;
    short EndColumn;
    short StartRow;
    short EndRow;
};
}; }; }; };
```

被映射成：

```
package com.sun.star.chart;
public class ChartDataChangeEvent extends com.sun.star.lang.EventObject {
    public ChartDataChangeType Type;
    public short StartColumn;
    public short EndColumn;
```

```
        public short StartRow;
        public short EndRow;
        public ChartDataChangeEvent() {
            Type = ChartDataChangeType.getDefault();
        }
        public ChartDataChangeEvent(
            Object Source, ChartDataChangeType Type,
    short StartColumn, short EndColumn, short StartRow, short EndRow)
        {
            super(Source);
            this.Type = Type;
            this.StartColumn = StartColumn;
            this.EndColumn = EndColumn;
            this.StartRow = StartRow;
            this.EndRow = EndRow;
        }
    }
```

与普通结构类型类似，UNO 多态结构类型模板也被映射成 Java 类。唯一区别在于带有参数类型的结构成员的处理方法，反过来，此处理在 Java 1.5 和旧版本之间也有所不同。

以多态结构类型模板为例：

```
module com { module sun { module star { module beans {
struct Optional<T> {
    boolean IsPresent;
    T Value;
};
}; }; }; };
```

在 Java 1.5 中，带有一系列类型参数的多态结构类型模板被映射成带有对应系列无约束类型参数的一般 Java 类。对于 com.sun.star.beans.Optional，这意味着对应的 Java 1.5 类与以下示例类似：

```
package com.sun.star.beans;
public class Optional<T> {
    public boolean IsPresent;
    public T Value;
    public Optional() {}
    public Optional(boolean IsPresent, T Value) {
        this.IsPresent = IsPresent;
        this.Value = Value;
    }
}
```

这种多态结构类型模板的实例很自然地映射成 Java 1.5。例如，UNO Optional<string>映射成 Java Optional<String>。但是，通常映射成原始 Java 类型的 UNO 类型参数映射成相应的 Java 包装类型：

- boolean 映射成 java.lang.Boolean。

- byte 映射成 java.lang.Byte。
- short 和 unsigned short 映射成 java.lang.Short。
- long 和 unsigned long 映射成 java.lang.Integer。
- hyper 和 unsigned hyper 映射成 java.lang.Long。
- float 映射成 java.lang.Float。
- double 映射成 java.lang.Double。
- char 映射成 java.lang.Character。

例如，UNO Optional<long> 映射成 Java Optional<Integer>。还要注意，any 和 com.sun.star.uno.XInterface 的 UNO 类型参数都映射成 java.lang.Object，因此，Optional<any>和 Optional<XInterface> 都映射成 Java Optional<Object>。

处理默认构造的多态结构类型实例的参数成员时，仍然存在少数问题和缺陷。一个问题是默认构造函数将这种成员初始化为 null，但是，除 any 的值或接口类型以外，null 在 Java UNO 的环境中通常不是合法值。例如，new Optional<PropertyValue>().Value 的类型为 com.sun.star.beans.PropertyValue（结构类型），但它却是一个空引用。同样，newOptional<Boolean>().Value 也是一个空引用（而不是对 Boolean.FALSE 的引用）。所选的解决方案通常允许将空引用作为 Java 类字段的值，这些值与 UNO 多态结构类型的参数成员相对应。然而，为了避免出现任何问题，这种情况下最好不要依赖默认构造函数，而是要明确初始化所有有问题的字段。请注意，这并非真正是 Optional 的问题，如同 Optional< T >() 一样。对于默认构造的实例，IsPresent 始终为 false，由于 com.sun.star.beans.Optional 的文档约定，Value 的实际内容应被忽略，其他情况下，应忽略 com.sun.star.beans.Ambiguous，然而，这是一个实际问题）。

另一个缺陷是默认构造的多态结构类型实例的类型 any 的参数成员（在 Java 1.5 中为 newOptional<Object>().Value，在 C++中为 Optional<Any> o; o.Value）在 Java 语言绑定和 C++ 语言绑定中的值不同。在 Java 中，它包含类型 XInterface 的一个空引用（即 Java 值 null），而在 C++ 中，它却包含 void。此外，为了避免出现任何问题，这种情况下最好不要依赖默认构造函数。

在 1.5 以前的 Java 版本中（这些版本不支持一般类型），多态结构类型模板以这样一种方法被映射成普通的 Java 类：任何参数成员都被映射成类 java.lang.Object 的类字段。这通常没有使用一般类型可取，因为它降低了类型安全性，但它具有与 Java 1.5 兼容的优点（实际上，单个 Java 类文件是为给定的 UNO 结构类型模板生成的，该类文件适用于 Java 1.5 和旧版本）。在 1.5 以前的 Java 版本中，Optional 示例如下：

```
package com.sun.star.beans;
public class Optional {
    public boolean IsPresent;
    public Object Value;
    public Optional() {}
    public Optional(boolean IsPresent, Object Value) {
        this.IsPresent = IsPresent;
        this.Value = Value;
    }
}
```

如何使用 java.lang.Object 表示任意 UNO 类型的值，其详细介绍如下：

- UNO 类型 boolean、byte、short、long、hyper、float、double 和 char 的值通过对应的 Java 类型 java.lang.Boolean、java.lang.Byte、java.lang.Short、java.lang.Integer、java.lang.Long、java.lang.Float、java.lang.Double 和 java.lang.Character 的非空引用表示。
- UNO 类型 unsigned short、unsigned long 和 unsigned hyper 的值通过对应的 Java 类型 java.lang.Short、java.lang.Integer 和 java.lang.Long 的非空引用表示。这样的 Java 类型的值是对应于有符号的 UNO 类型还是无符号的 UNO 类型必须从环境推算。
- UNO 类型 string、type、any 和 UNO 序列、枚举、结构和接口类型（全部映射成 Java 引用类型）的值用其标准的 Java 映射来表示。
- UNO 类型 void 和 UNO 异常类型不能用作实例化多态结构类型的类型参数。

这与如何使用 java.lang.Object 表示 UNO any 类型的值类似。这里区别之处只有 com.sun.star.uno.Any，它可以用来消除不同的 UNO 类型映射成 java.lang.Object 的同一个子类的情况。相反，在此，它必须始终是从给定的 Java 引用表示的 UNO 类型的环境中推算。

对于默认构造的多态结构类型实例的参数成员，Java 1.5 中所提到的问题和缺陷也适用于 Java 旧版本。

3. 异常类型的映射

UNO 异常类型被映射成同名的公共 Java 类。只有对这种类的非空引用才是有效的。有两个 UNO 异常是其他所有异常的基异常。它们是 com.sun.star.uno.Exception 和 com.sun.star.uno.RuntimeException，其他所有异常都是从它们继承的。Java 中的相应异常继承 Java 异常：

```
module com { module sun { module star { module uno {
exception Exception {
    string Message;
    XInterface Context;
};
exception RuntimeException {
    string Message;
    XInterface Context;
};
}; }; }; };
Java 中的 com.sun.star.uno.Exception：
package com.sun.star.uno;
public class Exception extends java.lang.Exception {
    public Object Context;
    public Exception() {}
    public Exception(String Message) {
        super(Message);
    }
    public Exception(String Message, Object Context) {
        super(Message);
        this.Context = Context;
    }
}
```

Java 中的 com.sun.star.uno.RuntimeException：

```
package com.sun.star.uno;
public class RuntimeException extends java.lang.RuntimeException {
    public Object Context;
    public RuntimeException() {}
    public RuntimeException(String Message) {
        super(Message);
    }
    public RuntimeException(String Message, Object Context) {
        super(Message);
        this.Context = Context;
    }
}
```

如示例所示，Message 成员不存在相应的 Java 类。而是使用构造函数参数 Message 来初始化 Java 异常的基类。可通过继承的 getMessage()方法访问 Message。UNO 异常类型的其他所有成员都被 121 映射成同名称、相应 Java 类型的公共字段。生成的 Java 异常类通常具有一个用默认值初始化所有成员的默认构造函数，以及一个获取所有成员值的构造函数。如果一个异常继承另一个异常，则生成的类是被继承异常的类的子类。

4. 接口类型的映射

UNO 接口类型被映射成一个同名的公共 Java 接口。与表示 UNO 序列、枚举、结构和异常类型的 Java 类不同，空引用对表示 UNO 接口类型的 Java 接口实际上是合法值，即 Java 空引用表示 UNO 空引用。

如果一个 UNO 接口类型继承一个或多个其他接口类型，则 Java 接口是对应的 Java 接口的子接口。UNO 接口类型 com.sun.star.uno.XInterface 很特殊：只有当该类型用作另一个接口类型的基类型时，它才被映射成 Java 类型 com.sun.star.uno.XInterface。在其他所有情况下（用作序列类型的组件类型、结构或异常类型的成员或者接口方法的参数或返回类型时），它都被映射成 java.lang.Object。然而，该类型的有效 Java 值只是 Java 空引用和对实现 com.sun.star.uno.XInterface 的 java.lang.Object 这些实例的引用。

如下形式的 UNO 接口属性：

```
[attribute] Type Name {
    get raises (ExceptionG1, ..., ExceptionGM);
    set raises (ExceptionS1, ..., ExceptionSM);
};
```

用两种 Java 接口方法表示：

Type getName() throws ExceptionG1, ..., ExceptionGM;

void setName(Type value) throws ExceptionS1, ..., ExceptionSM;

如果属性被标记为 readonly，则没有设置方法。属性是否被标记为 bound 对生成的 Java 方法的签名没有影响。

如下形式的 UNO 接口方法：

Type0 name([in] Type1 arg1, [out] Type2 arg2, [inout] Type3 arg3) raises (Exception1, ..., ExceptionN);

用 Java 接口方法表示：

Type0 name(Type1 arg1, Type2[] arg2, Type3[] arg3) throws Exception1, ..., ExceptionN;

UNO 方法是否被标记为 oneway 对生成 Java 方法的签名没有影响。可以看出，out 和 inout 参数要特殊处理。为了便于解释，以 UNOIDL 定义为例。

```
struct FooStruct {
    long nval;
    string strval;
};
interface XFoo {
    string funcOne([in] string value);
    FooStruct funcTwo([inout] FooStruct value);
    sequence<byte> funcThree([out] sequence<byte> value);
};
```

UNO 方法调用的语义是将任意 in 或 inout 参数的值从调用程序传递到被调用程序，如果方法没有标记为 oneway 并且执行成功终止，被调用程序将向调用程序传递回返回值和任意 out 或 inout 参数的值。因此，in 参数和返回值的处理很自然地映射成 Java 方法调用的语义。但是，UNO out 和 inout 参数则被映射成对应的 Java 类型的数组。每个这样的数组都必须至少有一个元素（即其长度至少必须为 1；实际上，其长度没有必要更大。）因此，与 UNO 接口 XFoo 对应的 Java 接口如下：

```
public interface XFoo extends com.sun.star.uno.XInterface {
    String funcOne(String value);
    FooStruct funcTwo(FooStruct[] value);
byte[] funcThree(byte[][] value);
}
```

下面说明如何将 FooStruct 映射到 Java：

```
public class FooStruct {
    public int nval;
    public String strval;
    public FooStruct() {
        strval="";
    }
    public FooStruct(int nval, String strval) {
        this.nval = nval;
        this.strval = strval;
    }
}
```

将一个值作为 inout 参数提供时，调用程序必须将输入值写到数组的索引 0 对应的元素中。当函数返回成功时，索引 0 对应的值反映输出值，该值可以是未修改的输入值、输入值的已修改副本或一个全新的值。对象 obj 实现 XFoo：

```
// calling the interface in Java
obj.funcOne(null);                    // error, String value is null
obj.funcOne("");                      // OK
```

```
FooStruct[] inoutstruct= new FooStruct[1];
obj.funcTwo(inoutstruct);                    // error, inoutstruct[0] is null
inoutstruct[0]= new FooStruct();             // now we initialize inoutstruct[0]
obj.funcTwo(inoutstruct);                    // OK
```

当方法接受作为 out 参数的一个自变量时，必须提供一个值，而且必须放在数组的索引 null 处。

```
// method implementations of interface XFoo
public String funcOne(/*in*/ String value) {
   assert value != null;                     // otherwise, it is a bug of the caller
   return null;                              // error; instead use: return "";
}
public FooStruct funcTwo(/*inout*/ FooStruct[] value) {
   assert value != null && value.length >= 1 && value[0] != null;
   value[0] = null;                          // error; instead use: value[0] = new FooStruct();
   return null;                              // error; instead use: return new FooStruct();
}
public byte[] funcThree(/*out*/ byte[][] value) {
   assert value != null && value.length >= 1;
   value[0] = null;                          // error; instead use: value[0] = new byte[0];
   return null;                              // error; instead use: return new byte[0];
}
```

5. UNOIDL 类型定义的映射

UNOIDL 类型定义在 Java 语言绑定中不可见。从 UNOIDL 映射到 Java 时，出现的每个类型定义都用别名类型替换。

6. 个别 UNOIDL 常数的映射

个别 UNOIDL 常数：

```
module example {
      const long USERFLAG = 1;
};
```

被映射成同名的公共 Java 接口：

```
123package example;
public interface USERFLAG {
      int value = 1;
}
```

请注意，个别常数已不再使用。

7. UNOIDL 常数组的映射

UNOIDL 常数组：

```
module example {
      constants User {
            const long FLAG1 = 1;
            const long FLAG2 = 2;
            const long FLAG3 = 3;
```

```
    };
};
```

被映射成同名的公共 Java 接口：

```
package example;
public interface User {
    int FLAG1 = 1;
    int FLAG2 = 2;
    int FLAG3 = 3;
}
```

该组中定义的每个常数都被映射成相同同名称、对应类型和值的接口字段。

8. UNOIDL 模块的映射

UNOIDL 模块被映射成同名的 Java 软件包。实际上，每个名为 UNO 和 UNOIDL 的实体都被映射成同名的 Java 类。（UNOIDL 将“::”用在“com::sun::star::uno”中来分隔名称内的单独标识符，而 UNO 本身和 Java 则将“.”用在“com.sun.star.uno”中；因此，必须先以明显的方法转换 UNOIDL 实体的名称，然后才能在 Java 中用作名称）。未包括在任何模块中的 UNO 和 UNOIDL 实体（即其名称分别不包含任何“.”或“::”）在未命名软件包中被映射成 Java 类。

9. 服务的映射

新式服务被映射成同名的公共 Java 类。该类有一个或多个公共静态方法，这些方法与服务的显式或隐式构造函数相对应。

对于具有给定接口类型 XIfc 的新式服务，以下形式的显式构造函数

name([in] Type1 arg1, [in] Type2 arg2) raises (Exception1, ..., ExceptionN);

用以下 Java 方法表示：

public static XIfc name(com.sun.star.uno.XComponentContext context, Type1 arg1, Type2 arg2)
    throws Exception1, ..., ExceptionN { ... }

UNO rest 参数 (any...)在 Java 1.5 中被映射成 Java rest 参数 (java.lang.Object...)，在 Java 的旧版本中被映射成 java.lang.Object[]。

如果新式服务有隐式构造函数，则对应的 Java 方法的形式为 public static XIfc create(com.sun.star.uno.XComponentContext context) { ... }

Java 中显式和隐式服务构造函数的语义如下：

- 服务构造函数的第一个参数始终为 com.sun.star.uno.XComponentContext，且不得为空。其他所有参数都用于初始化创建的服务（见下文）。
- 服务构造函数首先使用 com.sun.star.uno.XComponentContext:getServiceManager 从给定的组件上下文获取服务管理器（com.sun.star.lang.XMultiComponentFactory）。然后，服务构造函数使用 com.sun.star.lang.XMultiComponentFactory:createInstance-WithArguments AndContext 创建能够向它传递参数列表（无初始的 XComponentContext）的服务实例。如果服务构造函数有单个 rest 参数，则可以直接使用 any 值的序列，否则，给定的参数将列入 any 值的序列中。如果是隐式服务构造函数，则不传递参数，而是使用 com.sun.star.lang.XmultiComponent Factory:createInstanceWithContext。

如果以上任何步骤因服务构造函数可能抛出（根据其异常规范）的异常而失败，则服务构造函

数也会抛出该异常并以失败告终。否则，如果以上任何步骤因不可能由服务构造函数抛出的异常而失败，则服务构造函数会抛出 com.sun.star.uno.DeploymentException 并以失败告终。最后，如果没有创建任何服务实例（由于给定的组件上下文无服务管理器，或者由于服务管理器不支持请求的服务），则服务构造函数会抛出 com.sun.star.uno.DeploymentException 并以失败告终。实际结果是服务构造函数或者返回所请求服务的非空实例，或者抛出异常；服务构造函数决不会返回空实例。

没有将旧式服务映射成 Java 语言绑定。

10. singleton 的映射

以下形式的新式 singleton：

singleton Name: XIfc;

被映射成同名的公共 Java 类。该类有单一方法：

public static XIfc get(com.sun.star.uno.XComponentContext context) { ... }

在 Java 中，这种 singleton getter 方法的语义如下：

- com.sun.star.uno.XComponentContext 参数不得为空。
- singleton getter 使用 com.sun.star.uno.XComponentContext:getValueByName 获取 singleton 实例（在“/singletons/”命名空间内）。
- 如果未获取 singleton 实例，则 singleton getter 会抛出 com.sun.star.uno.Deployment Exception 并以失败告终。实际结果是 singleton getter 或者返回所请求服务的 singleton 非空实例，或者抛出异常；singleton getter 决不会返回空实例。

没有将旧式 singleton 映射成 Java 语言绑定。

11. UNO 值语义的不精确近似值

在 Java 中，一些 UNO 类型通常被视为映射成引用类型的值类型。名义上，它们是 UNO 类型 string、type、any 及 UNO 序列、枚举、结构和异常类型。问题在于如果将这种类型（Java 对象）的值用作：

- 存储在 any 中的值。
- 序列组件的值。
- 结构或异常成员的值。
- 接口属性的值。
- 接口方法调用中的参数或返回值。
- 服务构造函数调用中的参数。
- 出现的异常。

则 Java 不会创建该对象的克隆，而是通过对它的多重引用共享对象。现在，如果通过任何一个引用来修改对象，则其他所有引用也可以查看所做的修改。这样就违背了计划的值语义。

在 Java 语言绑定中所选的解决方案禁止对任何 Java 对象进行修改，这些对象用来在上面列出的任何情况下表示 UNO 值。请注意，对于表示 UNO 类型 string 或 UNO 枚举类型值的 Java 对象，一般都能保证这一点，因为对应的 Java 类型不变。如果 Java 类 com.sun.star.Type 为 final，这同样适用于 UNO 类型 type。

从此处使用的意义上来说，修改 Java 对象 A 包括修改其他任何满足以下条件的 Java 对象 B，① B 通过一个或多个引用可以到达 A，② B 可用来在上面列出的任何情况下表示 UNO 值。对于表示 UNO any 值的 Java 对象，不对其进行修改的限制仅适用于类型 com.sun.star.uno.Any（实际上

应不变）的包装对象，或者仅适用于表示类型 string 或 type，以及序列、枚举、结构或异常类型的 UNO 值的未包装对象。

请注意，用于将某些具体的 UNO 值表示为 any 值或实例化多态结构类型的参数成员的类型 java.lang.Boolean、java.lang.Byte、java.lang.Short、java.lang.Integer、java.lang.Long、java.lang.Float、java.lang.Double 和 java.lang.Character 始终不变，因此，在此无需进行特殊考虑。

4.7.2 C++ 语言绑定

本节介绍 UNO C++ 语言绑定。它为有经验的 C++程序员提供使用 UNO 的最初步骤：建立与远程 RedOffice 的 UNO 进程间连接以及使用远程 RedOffice 的 UNO 对象。

- 程序库概述

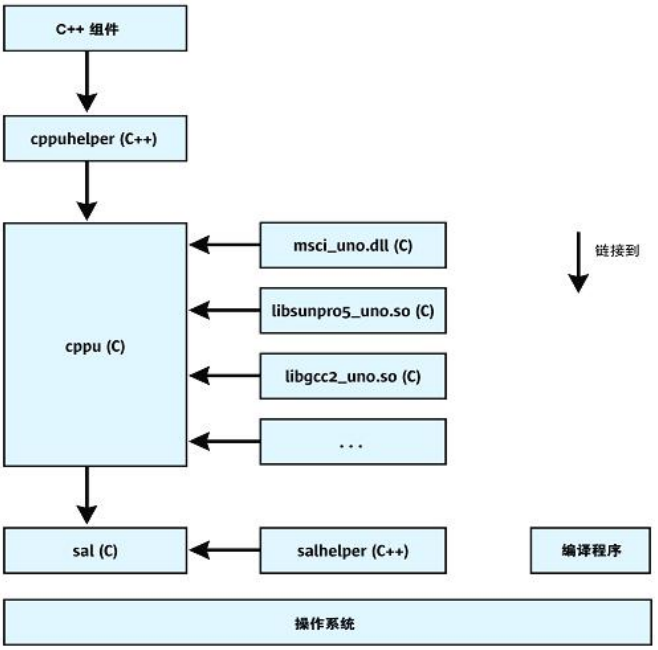

图 15：C++ UNO 的共享库

可以在安装 RedOffice 的<officedir>/program 文件夹中找到这些共享库。上图中的标签 (C) 表示 C 链接，而(C++) 表示 C++ 链接。所有程序库都需要一个 C++ 编译器来进行编译。

所有 UNO 程序库的基础是 sal 程序库。sal 程序库包含系统抽象层（sal）和附加的运行时库功能，但不包含任何 UNO 特有的信息。可以通过 C++ 内联包装类访问 sal 程序库中的常用 C 函数。这样，就可以从任何其他编程语言调用函数，因为多数编程语言都具有某种调用 C 函数的机制。

salhelper 程序库是一个小型 C++程序库，提供无法通过内联方式实现的附加运行时库功能。cppu（C++ UNO）程序库是核心 UNO 程序库。它提供访问 UNO 类型库的方法，并允许以普通方式建立、复制和比较 UNO 数据类型的值。另外，还在此库中管理所有 UNO 桥（=映射+环境）。

示例 msci_uno.dll、libsunpro5_uno.so 和 libgcc2_uno.so 只是某些 C++ 编译器语言绑定库的示例。cppuhelper 程序库是一个 C++程序库，包含 UNO 对象的重要基类以及用于引导 UNO 核心的函数。C++组件和 UNO 程序必须链接 cppuhelper 程序库。

在 UNO 的所有未来版本中，将保持与上面显示的所有程序库的兼容。您将能够一次构建并链接应用程序和组件，并能够使用 RedOffice 的当前和以后版本进行运行。

4.8 类型映射

1. 简单类型的映射

下表所示为 UNO 简单类型对应的 C++类型映射。

UNO	C++
void	void
boolean	sal_Bool
byte	sal_Int8
short	sal_Int16
unsigned short	sal_uInt16
long	sal_Int32
unsigned long	sal_uInt32
hyper	sal_Int64
unsigned hyper	sal_uInt64
float	float
double	double
char	sal_Unicoed
string	rt1::OUString
type	com::sum::star::uno::Type
any	com::sun::star::uno::Any

图 18

2. 字符串的映射

除以下两个细节以外，UNO 的 string 类型和 rtl::OUString 之间的映射很简单：

- 可以用 rtl::OUString 对象表示的字符串长度是有限的。在 C++ 语言绑定的环境中使用 UNO 的较长的 string 值是错误的。
- rtl::OUString 类型的对象可以表示 UTF-16 代码单元的任意序列，而 UNO 的 string 类型值是 Unicode 标量值的任意序列。目前为止，只有当某些个别的 UTF-16 代码单元（名义上是范围为 D800-DFFF 的高低替代码点）没有对应的 Unicode 标量值，并因此在 UNO 的环境中被禁止，这才会有影响。

3. 类型的映射

UNO 类型 type 被映射成 com::sun::star::uno::Type。它包含类型的名称和 com.sun.star.uno.TypeClass。该类型使您可以获取包含 IDL 中定义的所有信息的 com::sun::star::uno::TypeDescription。对于某个给定的类型，对应的 com::sun::star::Type 对象可以使用重载的 getCppuType ()函数来获取，而对于接口类型，则使用 static_type()函数来获取：

```
// get the type of sal_Int32
com::sun::star::uno::Type intType = getCppuType(static_cast< sal_Int32 * >(0));
```

```
// get the type of a string
com::sun::star::uno::Type stringType = getCppuType(static_cast< rtl::OUString * >(0));
// get the type of the XEnumeration interface
Type xenumerationType1 = getCppuType(
    static_cast< com::sun::star::uno::Reference< com::sun::star::container::XEnumeration > * >(0));
Type xenumerationType2 = com::sun::star::container::XEnumeration::static_type();
```

以上代码在编写模板函数时非常有用。有些 getCppuType()函数不太明确。下面是一些专用函数：

getVoidCppuType()、getBooleanCppuType()、getCharCppuType()，以处理不明确的函数。这些函数是以内联方式实现的，并且由类型库中生成的头文件引入。

4. 任意类型的映射

IDL any 被映射成 com::sun::star::uno::Any。它包含一个任意 UNO 类型的实例。UNO 类型只能存储在 any 中，因为处理 any 时需要来自类型库的数据。

默认构造的 Any 包含 void 类型且不包含值。您可以使用运算符 <<=将值指定到 Any，并可以使用运算符 >>= 获取值。

5. 结构类型的映射

普通的 UNO 结构类型被映射成一个同名的公共 C++类。UNO 结构类型的每个成员都被映射成相同名称、对应类型的公共数据成员。C++结构提供一个用默认值初始化所有成员的默认构造函数，以及一个获取所有成员确切值的构造函数。如果一个普通结构类型继承另一个结构类型，则生成的 C++结构是与被继承的 UNO 结构类型对应的 C++结构的子结构。

带有一系列类型参数的 UNO 多态结构类型模板被映射成带有对应系列类型参数的 C++结构模板。

例如，与 com.sun.star.beans.Optional 对应的 C++模板如下：

```
template< typename T > struct Optional {
    sal_Bool IsPresent;
    T Value;
    Optional(): IsPresent(sal_False), Value() {}
    Optional(sal_Bool theIsPresent, T const & theValue): IsPresent(theIsPresent), Value(theValue) {}
};
```

从上面的示例中可以看出，默认构造函数使用默认初始化为任何参数数据成员提供值。

6. 接口类型的映射

UNO 接口类型的值（为空引用或对给定接口类型对象实现的引用）被映射成模板类：

template< class t >

com::sun::star::uno::Reference< t >

模板用于获取类型安全接口引用，因为只可以将正确键入的接口指针指定到该引用。

7. 服务的映射

新式服务被映射成同名的 C++类。该类有一个或多个公共静态成员函数，这些函数与服务的显式或隐式构造函数相对应。

对于具有给定接口类型 XIfc 的新式服务，以下形式的显式构造函数 name([in] Type1 arg1, [in] Type2 arg2) raises (Exception1, ..., ExceptionN);

通过如下的 C++ 成员函数表示：

```
public:
static com::sun::star::uno::Reference< XIfc > name(
    com::sun::star::uno::Reference< com::sun::star::uno::XComponentContext > const & context,
    Type1 arg1, Type2 arg2)
    throw (Exception1, ..., ExceptionN, com::sun::star::uno::RuntimeException) { ... }
```

4.9 脚本连接

1. UNO 和 Basic 类型的映射

Basic 和 UNO 使用不同的类型系统。尽管 RedOffice Basic 与 Visual Basic 及其类型系统兼容，UNO 类型对应于 IDL 规范，因此有必要映射这两种类型系统。本章介绍不同 UNO 类型必须使用的 Basic 类型。

2. 简单类型映射

一般来说，RedOffice Basic 类型系统并不严格。与 C++和 Java 不同，RedOffice Basic 不要求声明变量，除非使用了强制进行声明的 Option Explicit 命令。要声明变量，使用的是 Dim 命令。此外，还可以通过 Dim 命令有选择地指定 RedOffice Basic 类型。一般语法是：

Dim VarName [As Type][, VarName [As Type]]...

在未指定类型的情况下，声明的所有变量都为 Variant 类型。可以将任意 Basic 类型的值指定给类型为 Variant 的变量。未声明的变量为 Variant，除非将类型后缀与其名称一起使用。后缀也可以在 Dim 命令中使用。下表包含 Basic 所支持的类型及其相应后缀的一个完整列表：

类型	后缀	范围
Boolean		True
Integer	%	-32768
Long	&	-2147483648
Single	!	浮点数 负数：-3.402823E38 到-1.401298E-45 正数：1.401298E-45
Double	#	双精度浮点数 负数：-1.79769313486232E308 到-4.94065645841247E-324 正数：4.940656458412474E-324
Currency	@	带有四位小数的固定点数 -922，337，203，685，477.5808 到 922，337，203，685，477.5807
Date		01/01/100
Object		Basic 对象
String	$	字符串
Variant		任意 Basic 类型

图 19

下面的关系表用于将 UNO 中的类型映射成 Basic 中的类型，反过来也可以。

UNO	Basic
void	内部类型
boolean	Boolean
byte	Integer
short	Integer
unsigned short	内部类型
long	Long
unsigned long	内部类型
hyper	内部类型
unsigned hyper	内部类型
float	Single
double	Double
char	内部类型
string	String
type	com.sun.star.reflection.XIdlClass
any	Variant

图 20

3. 映射

可以通过 Dim As New 命令将 UNO 结构类型实例化为单个实例和数组。

4. 枚举和常数组映射

使用符合命名规则的名称对枚举类型的值进行寻址。以下示例假定 oExample 和 oExample2 支持 com.sun.star.beans.XPropertySet，并包含一个枚举类型为 com.sun.star.beans.PropertyState 的属性 Status。