参考：<https://waylau.com/remote-procedure-calls/>

远程过程调用(RPC)详解

# 远程过程调用(RPC)详解

2016-06-30  [RPC](https://waylau.com/categories/" \l "RPC)  [Web](https://waylau.com/tags/#Web)[RPC](https://waylau.com/tags/#RPC)

本文介绍了什么是远程过程调用(RPC)，RPC 有哪些常用的方法，RPC 经历了哪些发展阶段，以及比较了各种 RPC 技术的优劣。

## 什么是 RPC

RPC 是远程过程调用（Remote Procedure Call）的缩写形式，Birrell 和 Nelson 在 1984 发表于 ACM Transactions on Computer Systems 的论文《Implementing remote procedure calls》(实现远程调用)对 RPC 做了经典的诠释。RPC 是指计算机 A 上的进程，调用另外一台计算机 B 上的进程，其中 A 上的调用进程被挂起，而 B 上的被调用进程开始执行，当值返回给 A 时，A 进程继续执行（同步模式：请求-响应）。调用方可以通过使用参数将信息传送给被调用方，而后可以通过传回的结果得到信息。而这一过程，对于开发人员来说是透明的。

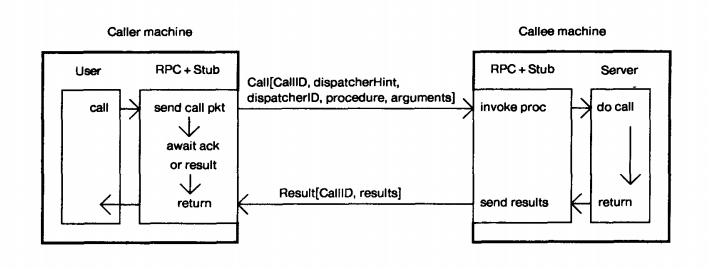


图1 描述了数据报在一个简单的RPC传递的过程（me:ack报文同tcp连接时的acp报文）

注：上述论文，可以在线阅读 [http://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf](https://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf)。

远程过程调用采用客户机/服务器(C/S)模式。请求程序就是一个客户机，而服务提供程序就是一台服务器。和常规或本地过程调用一样，远程过程调用是同步操作(me：A程序在执行远程调用时会挂起，远程调用比http块)，在远程过程结果返回之前，需要暂时中止请求程序。使用相同地址空间的低权进程或低权线程（me：避免权限较大产生安全问题，如sql server的browser服务）允许同时运行多个远程过程调用。

## RPC 的基本操作

让我们看看本地过程调用是如何实现的。考虑下面的 C 语言的调用：

count = read(fd, buf, nbytes);

其中，fd 为一个整型数，表示一个文件。buf 为一个字符数组，用于存储读入的数据。 nbytes 为另一个整型数，用于记录实际读入的字节数。如果该调用位于主程序中，那么在调用之前堆栈的状态如图2(a)所示。为了进行调用，调用方首先把参数反序压入堆栈，即为最后一个参数先压入，如图2(b)所示。在 read 操作运行完毕后，它将返回值放在某个寄存器中，移出返回地址，并将控制权交回给调用方。调用方随后将参数从堆栈中移出，使堆栈还原到最初的状态。

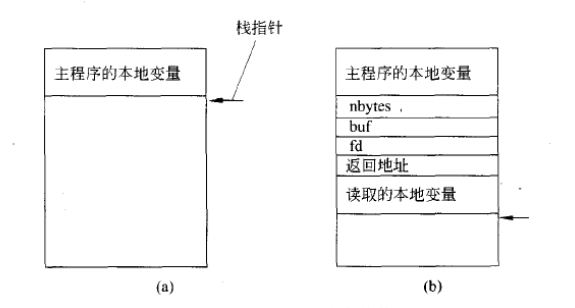


图2 过程调用中的参数传递

RPC 背后的思想是尽量使远程过程调用具有与本地调用相同的形式。假设程序需要从某个文件读取数据，程序员在代码中执行 read 调用来取得数据。在传统的系统中， read 例程由链接器从库中提取出来，然后链接器再将它插入目标程序中。 read 过程是一个短过程，一般通过执行一个等效的 read 系统调用来实现。即，read 过程是一个位于用户代码与本地操作系统之间的接口。

虽然 read 中执行了系统调用，但它本身依然是通过将参数压入堆栈的常规方式调用的。如图2(b)所示，程序员并不知道 read 干了啥。

RPC 是通过类似的途径来获得透明性。当 read 实际上是一个远程过程时（比如在文件服务器所在的机器上运行的过程），库中就放入 read 的另外一个版本，称为客户存根（client stub）。这种版本的 read 过程同样遵循图2(b)的调用次序，这点与原来的 read 过程相同。另一个相同点是其中也执行了本地操作系统调用。唯一不同点是它不要求操作系统提供数据，而是将参数打包成消息，而后请求此消息发送到服务器，如图3所示。在对 send 的调用后，客户存根调用 receive 过程，随即阻塞自己（同步操作），直到收到响应消息。

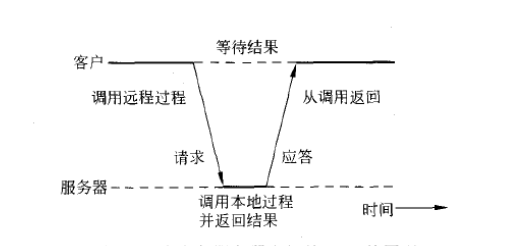


图3 客户与服务器之间的RPC原理

当消息到达服务器时，服务器上的操作系统将它传递给服务器存根（server stub）。服务器存根是客户存根在服务器端的等价物，也是一段代码，用来将通过网络输入的请求转换为本地过程调用。服务器存根一般先调用 receive（me：提供服务） ，然后被阻塞（同步操作），等待消息输入。收到消息后，服务器将参数由消息中提取出来，然后以常规方式调用服务器上的相应过程（如图3所示）。从服务器角度看，过程好像是由客户直接调用的一样：参数和返回地址都位于堆栈中，一切都很正常。服务器执行所要求的操作，随后将得到的结果以常规的方式返回给调用方。以 read 为例，服务器将用数据填充 read 中第二个参数指向的缓冲区，该缓存区是属于服务器存根内部的。

调用完后，服务器存根要将控制权交回给客户发出调用的过程，它将结果（缓冲区）打包成消息（所以传递时要求对象是可序列化的），随后调用 send 将结果返回给客户。事后，服务器存根一般会再次调用 receive，等待下一个输入的请求。

客户机器接收到消息后，客户操作系统发现该消息属于某个客户进程（实际上该进程是客户存根，只是操作系统无法区分二者）。操作系统将消息复制到相应的缓存区中，随后解除对客户进程的阻塞。客户存根检查该消息，将结果提取出来并复制给调用者，而后以通常的方式返回。当调用者在 read 调用进行完毕后重新获得控制权时，它所知道的唯一事就是已经得到了所需的数据。它不知道操作是在本地操作系统进行，还是远程完成。

整个方法，客户方可以简单地忽略不关心的内容。客户所涉及的操作只是执行普通的（本地）过程调用来访问远程服务，它并不需要直接调用 send 和 receive 。消息传递的所有细节都隐藏在双方的库过程中，就像传统库隐藏了执行实际系统调用的细节一样。

概况来说，远程过程调用包含如下步骤：

1. 客户过程以正常的方式调用客户存根；
2. 客户存根生成一个消息，然后调用本地操作系统；
3. 客户端操作系统将消息发送给远程操作系统；
4. 远程操作系统将消息交给服务器存根；
5. 服务器存根调将参数提取出来，而后调用服务器；
6. 服务器执行要求的操作，操作完成后将结果返回给服务器存根；
7. 服务器存根将结果打包成一个消息，而后调用本地操作系统；
8. 服务器操作系统将含有结果的消息发送给客户端操作系统；
9. 客户端操作系统将消息交给客户存根；
10. 客户存根将结果从消息中提取出来，返回给调用它的客户存根。

以上步骤就是将客户过程对客户存根发出的本地调用转换成对服务器过程的本地调用，而客户端和服务器都不会意识到中间步骤的存在。

RPC 的主要好处是双重的。首先,程序员可以使用过程调用语义来调用远程函数并获取响应。其次,简化了编写分布式应用程序的难度,因为 RPC 隐藏了所有的网络代码存根函数。应用程序不必担心一些细节,比如 socket、端口号以及数据的转换和解析。在 OSI 参考模型,RPC 跨越了会话层和表示层。

## 实现远程过程调用

要实现远程过程调用，需考虑以下几个问题。

### 如何传递参数

#### 传递值参数

传递值参数比较简单，下图图展示了一个简单 RPC 进行远程计算的例子。其中，远程过程 add(i,j) 有两个参数 i 和 j， 其结果是返回 i 和 j 的算术和。

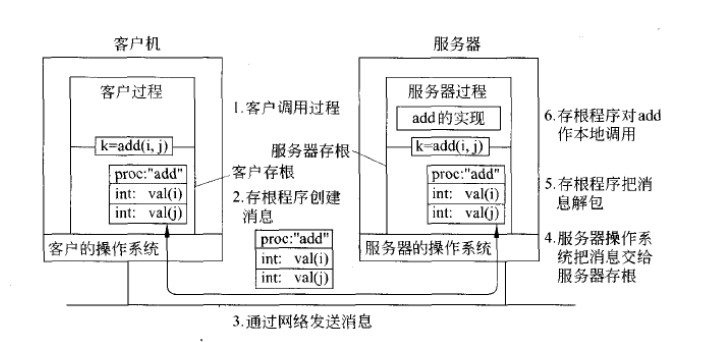


图4 通过RPC进行远程计算的步骤

通过 RPC 进行远程计算的步骤有：

1. 将参数放入消息（me：要求参数可序列化）中，并在消息中添加要调用的过程的名称或者编码。
2. 消息到达服务器后，服务器存根对该消息进行分析，以判明需要调用哪个过程，随后执行相应的调用。
3. 服务器运行完毕后，服务器存根将服务器得到的结果打包成消息（me:要求对象可序列化）送回客户存根，客户存根将结果从消息中提取出来，把结果值返回给客户端。

当然，这里只是做了简单的演示，在实际分布式系统中，还需要考虑其他情况，因为不同的机器对于数字、字符和其他类型的数据项的表示方式常有差异。比如整数型，就有 Big Endian 和 Little Endian 之分。

#### 传递引用参数

传递引用参数相对来说比较困难。单纯传递参数的引用（也包含指针）是完全没有意义的，因为引用地址传递给远程计算机,其指向的内存位置可能跟远程系统上完全不同。如果你想支持传递引用参数,你必须发送参数的副本,将它们放置在远程系统内存中，向他们传递一个指向服务器函数的指针，然后将对象发送回客户端，复制它的引用。如果远程过程调用必须支持引用复杂的结构，比如树和链表，他们需要将结构复制到一个无指针的表示里面（比如，一个扁平的树)，并传输到在远程端来重建数据结构。

### 如何表示数据

在本地系统上不存在数据不相容的问题，因为数据格式总是相同的。而在分布式系统中，不同远程机器上可能有不同的字节顺序，不同大小的整数，以及不同的浮点表示。对于 RPC，如果想与异构系统通信，我们就需要想出一个“标准”来对所有数据类型进行编码，并可以作为参数传递。例如，ONC RPC 使用 XDR (eXternal Data Representation) 格式 。这些数据表示格式可以使用隐式或显式类型。隐式类型，是指只传递值，而不传递变量的名称或类型。常见的例子是 ONC RPC 的 XDR 和 DCE RPC 的 NDR。显式类型，指需要传递每个字段的类型以及值。常见的例子是 ISO 标准 ASN.1 (Abstract Syntax Notation)、JSON (JavaScript Object Notation)、Google Protocol Buffers、以及各种基于 XML 的数据表示格式。

### 如何选用传输协议

有些实现只允许使用一个协议(例如 TCP )。大多数 RPC 实现支持几个（如TCP和ipc-进程间通信协议），并允许用户选择。

### 出错时,会发生什么

相比于本地过程调用，远程过程调用出错的机会将会更多。由于本地过程调用没有过程调用失败的概念，项目使用远程过程调用必须准备测试远程过程调用的失败或捕获异常。

### 远程调用的语义是什么

调用一个普通的过程语义很简单:当我们调用时，过程被执行。远程过程完全一次性调用成功是非常难以实现。执行远程过程可以有如下结果:

* 如果服务器崩溃或进程在运行服务器代码之前就死了，那么远程过程会被执行0次；
* 如果一切工作正常，远程过程会被执行1次；
* 如果服务器返回服务器存根后在发送响应前就奔溃了，远程过程会被执行1次或者多次。客户端接收不到返回的响应,可以决定再试一次,因此出现多次执行函数。如果没有再试一次,函数执行一次；
* 如果客户机超时和重新传输，那么远程过程会被执行多次。也有可能是原始请求延迟了。两者都可能会执行或不执行。

RPC 系统通常会提供至少一次或最多一次的语义，或者在两者之间选择。如果需要了解应用程序的性质和远程过程的功能是否安全，可以通过多次调用同一个函数来验证。如果一个函数可以运行任何次数而不影响结果，这是幂等（idempotent）函数的，如每天的时间、数学函数、读取静态数据等。否则,它是一个非幂等（nonidempotent）函数，如添加或修改一个文件)。

### 远程调用的性能怎么样

毫无疑问,一个远程过程调用将会比常规的本地过程调用慢得多，因为产生了额外的步骤以及网络传输本身存在延迟（me:比单纯的http块很多）。然而，这并不应该阻止我们使用远程过程调用。

### 远程调用安全吗？

使用 RPC,我们必须关注各种安全问题:

* 客户端发送消息到远程过程，那个过程是可信的吗?
* 客户端发送消息到远程计算机，那个远程机器是可信的吗?
* 服务器如何验证接收的消息是来自合法的客户端吗?服务器如何识别客户端?
* 消息在网络中传播如何防止时被其他进程嗅探?
* 可以由其他进程消息被拦截和修改时遍历网络从客户端到服务器或服务器端?
* 协议能防止重播攻击吗?
* 如何防止消息在网络传播中被意外损坏或截断?

## 远程过程调用的优点

远程过程调用有诸多的优点：

* 你不必担心传输地址问题。服务器可以绑定到任何可用的端口,然后用 RPC 名称服务来注册端口。客户端将通过该名称服务来找到对应的端口号所需要的程序。而这一切对于程序员来说是透明的。
* 系统可以独立于传输提供者。自动生成服务器存根使其可以在系统上的任何一个传输提供者上可用,包括 TCP 和 UDP，而这些，客户端可以动态选择的。当代码发送以后，接收消息是自动生成的，而不需要额外的编程代码。
* 应用程序在客户端只需要知道一个传输地址——名称服务，负责告诉应用程序去哪里连接服务器函数集。
* 使用函数调用模型来代替 socket 的发送/接收(读/写)接口。用户不需要处理参数的解析。

## RPC API

任何 RPC 实现都需要提供一组支持库。这些包括:

* 名称服务操作： 注册和查找绑定信息(端口、机器)。允许一个应用程序使用动态端口(操作系统分配的)；
* 绑定操作：使用适当的协议建立客户机/服务器通信(建立通信端点)；
* 终端操作：注册端点信息(协议、端口号、机器名)到名称服务并监听过程调用请求。这些函数通常被自动生成的主程序——服务器存根(骨架)所调用；
* 安全操作：系统应该提供机制保证客户端和服务器之间能够相互验证,两者之间提供一个安全的通信通道；
* 国际化操作(可能)：这是很少的一部分 RPC 包可能包括了转换时间格式、货币格式和特定于语言的在字符串表的字符串的功能；
* 封送处理/数据转换操作：函数将数据序列化为一个普通的的字节数组，通过网络进行传递，并能够重建；
* 存根内存管理和垃圾收集：存根可能需要分配内存来存储参数，特别是模拟引用传递语义。RPC 包需要分配和清理任何这样的分配。他们也可能需要为创建网络缓冲区而分配内存。RPC 包支持对象，RPC 系统需要一种跟踪远程客户端是否仍有引用对象或一个对象是否可以删除。
* 程序标识操作：允许应用程序访问(或处理) RPC 接口集的标识符，这样的服务器提供的接口集可以被用来交流和使用。
* 对象和函数的标识操作： 允许将远程函数或远程对象的引用传递给其他进程。并不是所有的 RPC 系统都支持。

所以，判断一种通信方式是否是 RPC，就看它是否提供上述的 API。

## 第一代 RPC

### ONC RPC(以前称为 Sun RPC)

Sun 公司是第一个提供商业化 RPC 库和 RPC 编译器。在1980年代中期 Sun 计算机提供 RPC，并在 Sun Network File System(NFS) 得到支持。该协议被主要以 Sun 和 AT&T 为首的 Open Network Computing （开放网络计算）作为一个标准来推动。这是一个非常轻量级 RPC 系统可用在大多数 POSIX 和类 POSIX 操作系统中使用，包括 Linux、SunOS、OS X 和各种发布版本的 BSD。这样的系统被称为 Sun RPC 或 ONC RPC。

ONC RPC 提供了一个编译器，需要一个远程过程接口的定义来生成客户机和服务器的存根函数。这个编译器叫做 rpcgen。在运行此编译器之前，程序员必须提供接口定义。包含函数声明的接口定义，通过版本号进行分组，并被一个独特的程序编码来标识。该程序编码能够让客户来确定所需的接口。版本号是非常有用的,即使客户没有更新到最新的代码仍然可以连接到一个新的服务器，只要该服务器还支持旧接口。

参数通过网络转化成一种隐式类型序列化格式被称为 XDR (eXternal Data Representation)。这将确保参数能够发送到异构系统可以被正常使用，及时这些系统可能使用了不同的字节顺序，不同大小的整数，或不同的浮点或字符串表示。最后，Sun RPC 提供了一个实现必要的支持 RPC 协议和 socket 例程的运行时库。

所有的程序员都需要写是一个客户端程序([client.c](https://www.cs.rutgers.edu/~pxk/417/notes/rpc/client.c)),服务器功能([server.c](https://www.cs.rutgers.edu/~pxk/417/notes/rpc/server.c))和 RPC 接口定义([date.x](https://www.cs.rutgers.edu/~pxk/417/notes/rpc/date.x))。当 RPC 接口定义(后缀为.x 的文件，例如 date.x)是用 rpcgen 编译的,会创建三个或四个文件。下面是 date.x 的例子:

* date.h：包含项目的定义、版本和声明的函数。客户端和服务器端功能应该包括这个文件。
* date\_svc.c ：C 语言代码来实现服务器存根。
* date\_clnt.c ：C 语言代码来实现客户端存根。
* date\_xdr.c ：包含 XDR 例程来将数据转化为 XDR 格式。如果这个文件生成,它应该编译并用来链接在客户端和服务器的函数。

创建客户端和服务器可执行文件的第一步是定义在文件 [date.x](https://www.cs.rutgers.edu/~pxk/417/notes/rpc/date.x) 里的编译数据。之后，客户端和服务器端函数可能被编译，并链接各自 rpcgen 生成的存根函数。

在旧版本里，传输协议只能将字符串“tcp”或字符串“udp”来指定各自的 IP 服务 RPC，且仅限于 Linux 实现的 RPC。为了使接口更加灵活，UNIX 系统从版本 4 (SunOS 从版本 5)开始网络选择程序允许一个更普通的规范。他们搜索文件(/etc/netconfig)，来查找第一个满足您需求的提供者。最后一个参数可以是:

* “netpath”：搜索 NETPATH 环境变量用于首选传输提供者的序列；
* “circuit\_n”：找到第一个在 NETPATH 列表中的虚拟电路提供者；
* “datagram\_n”：找到第一个 NETPATH 列表中的数据报提供者；
* “visible”：找到第一个在 /etc/netconfig 的可见传输提供者；
* “circuit\_v”：找到第一个在 /etc/netconfig 的可见虚拟电路传输提供者；
* “datagram\_v”：找到第一个在 /etc/netconfig 的可见数据报传输提供者；

每个远程过程调用最初仅限于接受一个输入参数。系统只是后来修改为支持多个参数。支持单一参数 RPC 在一些 rpcgen 的版本中仍然是默认的，比如苹果的 OS X。传递多个参数必须通过定义一个结构，包含所需的参数，初始化它，并传递这个结构。

远程过程调用返回一个指针指向结果而不是期望的结果。服务器函数必须修改来能接受一个 RPC 定义（.x 文件）中声明的值的指针作为输入，并返回一个结果值的指针。在服务器上，一个指针必须是指向静态数据的指针。否则，当过程返回或释放过程的框架所指出的区域将未定义。在客户端,返回指针可以让我们区分一个失败的 RPC(空指针)和一个空返回从服务器(间接空指针)。

RPC 过程的名称若在 RPC 定义文件中做了定义，则会转换为小写，并在后缀价下划线，后跟一个版本号。例如，BIN\_DATE 转成为引用函数 bin\_date\_1 。您的服务器必须实现 bin\_date\_1。

### 当我们运行这个程序时,会发生什么?

#### 服务器

当我们启动服务器，服务器存根代码(程序)在后台运行运行。它创建一个 socket 并可绑定任何本地端口到 socket。然后调用一个在 RPC 库的函数 svc\_register，来注册程序编号和版本。这个是用来联系 port mapper（端口映射器）。port mapper 是一个独立的进程，通常是在系统启动时启动。它跟踪端口号、版本号以及程序编号。在 UNIX 系统版本4中，这个进程称为 rpcbind。在 Linux 、OS X 和 BSD 系统,它被称为 portmap。

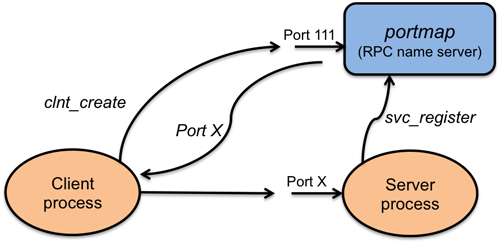


图5 ONC RPC 中的函数查找

#### 客户端

当我们开始客户端程序时，它首先用远程系统的名称、程序编号、版本号和协议来调用 clnt\_create 。它接触远程系统上的端口映射器,来为系统找到合适的端口。

然后客户端调用 RPC 存根函数(在本例中为 bin\_date\_1)。该函数发送一条消息(如，数据报)到服务器(使用早些时候发现的端口号)并等待响应。对于数据报服务来说，若没有接收到响应，它将重新发送一个固定的次数请求。

消息接着被远程系统接收到，它调用服务器函数(bin\_date\_1)并将返回值返回给客户端存根。客户端存根而后返回到客户端发出调用的代码。

## 分布式计算环境中的 RPC(DCE RPC)

DCE(Distributed Computing Environment，分布式计算环境)是一组由OFS(Open Software Foundation，开放软件基金会)设计的组件，用来提供支持分布式应用和分布式环境。与 X/Open 合并后,这组织成为了 The Open Group （开放式开发组）。DCE 提供的组件包括一个分布式文件服务、时间服务、目录服务以及其他服务。当然，我们感兴趣的是 DCE 的远程过程调用。它非常类似于 Sun RPC。接口是由 Interface Definition Notation (IDN) 定义的。类似于 Sun RPC，接口定义就像函数原型。

Sun RPC 不足之处在于，服务器的标识是一个“独特”的 32-bit 数字。虽然这是一个比在 socket 中 16-bit 可用空间更大的空间，但仍然无法满足数字唯一性的需求。DCE RPC 考虑到了这一缺陷，它无需程序员来处理编码。在编写应用程序时的第一步是从 uuidgen 程序获得一个惟一的 ID。这个程序会生成一个包含 ID 接口的原型 IDN 文件，并保证永远不会再次使用。它是一个 128-bit 的值，其中包含一个位置代码和创建时间的编码。然后用户编辑原型文件，填写远程过程声明。

在这一步后，IDN 的编译器 dceidl（类似于 rpcgen）会生成一个头、客户机存根和服务器存根。

Sun RPC 的另一个缺陷是，客户端必须知道服务器在哪台机器上。当它要访问时，必须要询问机器上的 RPC 名称服务程序编码所对应的端口号。DCE 支持将多个机器组织成为管理实体，称为 cells。cell 目录服务器使得每台机器知道如何与另外一台负责维护 cell 信息服务机器交互。

在 Sun RPC 中，服务器只能用本地名称服务(端口映射器)来注册其程序编号到端口映射。而在 DCE 中，服务器用 RPC 守护进程(名称服务器)来注册其端点(端口)到本地机器，并且用 cell 目录服务器注册其程序名字到机器的映射。当客户机想要与一个 RPC 服务器建立通信，它首先要求其 cell 目录服务器来定位服务器所在的机器。然后客户端从 RPC 守护进程处获得机器上服务器进程的端口号。DCE 的跨 cell 还支持更复杂的搜索。

DCE RPC 定义了 NDR (Network Data Representation) 用于对网络进行编码来封送信息。与用一个单一的规范来表示不同的数据类型相比，NDR 支持多规范（multi-canonical）格式。允许客户端来选择使用哪种格式，理想的情况是不需要将它从本地类型来转换。如果这不同于服务器的本地数据表示,服务器将仍然需要转换，但多规范格式可以避免当客户端和服务器都共享相同的本地格式的情况下转换为其他外部格式。例如,在一个规定了大端字节序网络数据格式的情况下，客户端和服务器只支持小端字节序，那么客户端必须将每个数据从小端字节序转为大端字节序，而当服务器接受到消息后,将每个数据转回小端字节序。多规范网络数据表示将允许客户端发送网络消息包含小端字节序格式的数据。

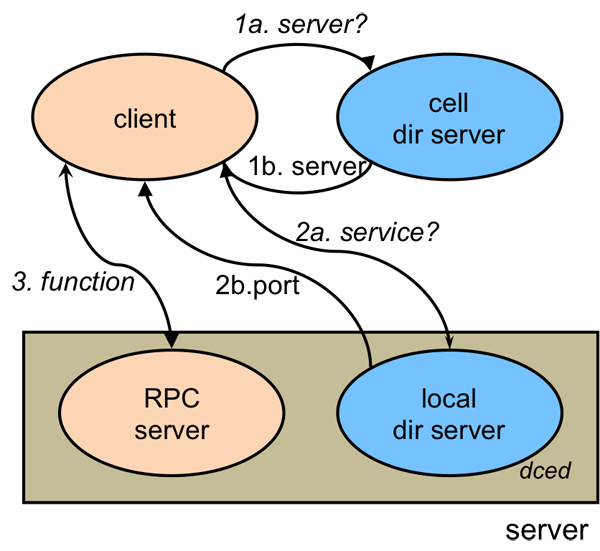


图6 DCE RPC 中的函数查找

## 第二代 RPC:支持对象

面向对象的语言开始在1980年代末兴起，很明显，当时的 Sun ONC 和 DCE RPC 系统都没有提供任何支持诸如从远程类实例化远程对象、跟踪对象的实例或提供支持多态性。现有的 RPC 机制虽然可以运作，但他们仍然不支持自动、透明的方式的面向对象编程技术。

### 微软 DCOM(COM+)

1992年4月,微软发布 Windows 3.1 包括一种机制称为 OLE (Object Linking and Embedding)。这允许一个程序动态链接其他库来支持的其他功能。如将一个电子表格嵌入到 Word 文档。OLE 演变成了 COM (Component Object Model)。一个 COM 对象是一个二进制文件。使用 COM 服务的程序来访问标准化接口的 COM 对象而不是其内部结构。COM 对象用全局唯一标识符(GUID)来命名，用类的 ID 来识别对象的类。几种方法来创建一个 COM 对象(例如 CoGetInstanceFromFile)。COM 库在系统注册表中查找相应的二进制代码(一个 DLL 或可执行文件)，来创建对象，并给调用者返回一个接口指针。COM 的着眼点是在于同一台计算机上不同应用程序之间的通讯需求.

DCOM（ Distributed Component Object Model）是 COM 的扩展，它支持不同的两台机器上的组件间的通信，而且不论它们是运行在局域网、广域网、还是 Internet 上。借助 DCOM 你的应用程序将能够进行任意空间分布。DCOM 于1996年在 Windows NT4.0 中引入的，后来更名为 COM+。由于 DCOM 是为了支持访问远程 COM 对象，需要创建一个对象的过程，此时需要提供服务器的网络名以及类 ID。微软提供了一些机制来实现这一点。最透明的方式是远程计算机的名称固定在注册表（或 DCOM 类存储）里，与特定类 ID 相关联。以此方式，应用程序不知道它正在访问一个远程对象，并且可以使用与访问本地 COM 对象相同的接口指针。另一方面，应用程序也可指定一个机器名作为参数。

由于 DCOM 是 COM 这个组件技术的无缝升级，所以你能够从你现有的有关 COM 得知识中获益，你的以前在 COM 中开发的应用程序、组件、工具都可以移入分布式的环境中。DCOM 将为你屏蔽底层网络协议的细节，你只需要集中精力于你的应用。

DCOM 最大的缺点是这是微软独家的解决办法，在跨防火墙方面的工作做得不是很好（大多数RPC系统也有类似的问题），因为防火墙必须允许某些端口来允许 ORPC 和 DCOM 通过。

### CORBA

虽然 DCE 修复的一些 Sun RPC 的缺点，但某些缺陷依然存在。例如，如果服务器没有运行，客户端是无法连接到远程过程进行调用的。管理员必须要确保在任何客户端试图连接到服务器之前将服务器启动。如果一个新服务或接口添加到了系统，客户端是不能发现的。最后,面向对象语言期望在函数调用中体现多态性，即不同类型的数据的函数的行为应该有所不同，而这点恰恰是传统的 RPC 所不支持的。

CORBA (Common Object Request Broker Architecture) 就是为了解决上面提到的各种问题。是由 OMG 组织制订的一种标准的面向对象应用程 序体系规范。或者说 CORBA体系结构是对象管理组织（OMG）为解决分布式处理环境(DCE)中，硬件和软件系统的互连而提出的一种解决方案。OMG 成立于1989年，作为一个非营利性组织，集中致力于开发在技术上具有先进性、在商业上具有可行性并且独立于厂商的软件互联规范，推广面向对象模型技术，增强软件的可移植性（Portability）、可重用性（Reusability）和互操作性（Interoperability）。该组织成立之初，成员包括 Unisys、Sun、Cannon、Hewlett-Packard 和 Philips 等在业界享有声誉的软硬件厂商，目前该组织拥有800多家成员。

CORBA 体系的主要内容包括以下几部分：

* 对象请求代理 （Object Request Broker，ORB）：负责对象在分布环境中透明地收发请求和响应，它是构建分布对象应用、在异构或同构环境下实现应用间互操作的基础。
* 对象服务（Object Services）：为使用和实现对象而提供的基本对象集合，这些服务应独立于应用领域。主要的 CORBA 服务有：名录服务（Naming Service）、事件服务（Event Service）、生命周期服务（Life Cycle Service）、关系服务（Relationship Service）以及事务服务（Transaction Service）等。这些服务几乎包括分布系统和面向对象系统的各个方面，每个组成部分都非常复杂。
* 公共设施（Common Facilitites）：向终端用户提供一组共享服务接口，例如系统管理、组合文档和电子邮件等。
* 应用接口（Application Interfaces）。由销售商提供的可控制其接口的产品，相应于传统的应用层表示，处于参考模型的最高层。
* 领域接口（Domain Interfaces）：为应用领域服务而提供的接口，如OMG 组织为 PDM 系统制定的规范。

当客户端发出请求时,ORB 做了如下事情:

* 在客户端编组参数；
* 定位服务器对象。如果有必要的话,它会在服务器创建一个过程来处理请求；
* 如果服务器是远程是,就使用 RPC 或 socket 来传送请求；
* 在服务器上将参数解析成为服务器格式；
* 在服务器上组装返回值；
* 如果服务器是远程的，就将返回值传回；
* 在客户端对返回结果进行解析；

IDL(Interface Definition Language) 是用于指定类的名字、属性和方法。它不包含对象的实现。IDL 编译器生成代码来处理编组、解封以及ORB与网络之间的交互。它会生成客户机和服务器存根。IDL 是编程语言中立，支持包括C、C++、Java、Perl、Python、Ada、COBOL、Smalltalk、Objective C 和 LISP 等语言。一个示例IDL如下所示:

Module StudentObject {

Struct StudentInfo {

String name;

int id;

float gpa;

};

exception Unknown {};

interface Student {

StudentInfo getinfo(in string name)

raises(unknown);

void putinfo(in StudentInfo data);

};

};

IDL数据类型包括:

* 基本类型:long, short, string, float, …
* 构造类型:struct、union、枚举、序列
* 对象引用
* any 类型:一个动态类型的值

编程中最常见的实现方式是通过对象引用来实现请求。下面是一个使用 IDL 的例子:

Student st = ... // get object reference

try {

StudentInfo sinfo = st.getinfo("Fred Grampp");

} catch (Throwable e) {

... // error

}

在 CORBA 规范中，没有明确说明不同厂商的中间件产品要实现所有的服务功能，并且允许厂商开发自己的服务类型。因此, 不同厂商的 ORB 产品对 CORBA 服务的支持能力不同，使我们在针对待开发系统的功能进行中间件产品选择时，有更多的选择余地。

CORBA 的不足有：

* 尽管有多家供应商提供CORBA产品，但是仍找不到能够单独为异种网络中的所有环境提供实现的供应商。不同的 CORBA 实现之间会出现缺乏互操作性的现象，从而造成一些问题；而且，由于供应商常常会自行定义扩展，而 CORBA 又缺乏针对多线程环境的规范，对于像 C 或 C++ 这样的语言，源码兼容性并未完全实现。
* CORBA 过于复杂，要熟悉 CORBA，并进行相应的设计和编程，需要许多个月来掌握，而要达到专家水平，则需要好几年。

更多有关 CORBA 的优缺点，可以参阅 Michi Henning 的《[The rise and fall of CORBA](https://queue.acm.org/detail.cfm?id=1142044)》。

### Java RMI

CORBA 旨在提供一组全面的服务来管理在异构环境中(不同语言、操作系统、网络)的对象。Java 在其最初只支持通过 socket 来实现分布式通信。1995年,作为 Java 的缔造者，Sun 公司开始创建一个 Java 的扩展，称为 Java RMI(Remote Method Invocation，远程方法调用)。Java RMI 允许程序员创建分布式应用程序时，可以从其他 Java 虚拟机(JVM)调用远程对象的方法。

一旦应用程序(客户端)引用了远程对象，就可以进行远程调用了。这是通过 RMI 提供的命名服务(RMI 注册中心)来查找远程对象，来接收作为返回值的引用。Java RMI 在概念上类似于 RPC，但能在不同地址空间支持对象调用的语义。

与大多数其他诸如 CORBA 的 RPC 系统不同，RMI 只支持基于 Java 来构建，但也正是这个原因， RMI 对于语言来说更加整洁，无需做额外的数据序列化工作。Java RMI 的设计目标应该是:

* 能够适应语言、集成到语言、易于使用；
* 支持无缝的远程调用对象；
* 支持服务器到 applet 的回调；
* 保障 Java 对象的安全环境；
* 支持分布式垃圾回收；
* 支持多种传输。

分布式对象模型与本地 Java 对象模型相似点在于:

* 引用一个对象可以作为参数传递或作为返回的结果；
* 远程对象可以投到任何使用 Java 语法实现的远程接口的集合上；
* 内置 Java instanceof 操作符可以用来测试远程对象是否支持远程接口。

不同点在于:

* 远程对象的类是与远程接口进行交互，而不是与这些接口的实现类交互；
* Non-remote 参数对于远程方法调用来说是通过复制，而不是通过引用；
* 远程对象是通过引用来传递，而不是复制实际的远程实现；
* 客户端必须处理额外的异常。

#### 接口和类

所有的远程接口都继承自 java.rmi.Remote 接口。例如:

public interface bankaccount extends Remote

{

public void deposit(float amount)

throws java.rmi.RemoteException;

public void withdraw(float amount)

throws OverdrawnException,

java.rmi.RemoteException;

}

注意,每个方法必须在 throws 里面声明 java.rmi.RemoteException 。 只要客户端调用远程方法出现失败，这个异常就会抛出。

#### 远程对象类

Java.rmi.server.RemoteObject 类提供了远程对象实现的语义包括hashCode、equals　和　toString。 java.rmi.server.RemoteServer　及其子类提供让对象实现远程可见。java.rmi.server.UnicastRemoteObject 类定义了客户机与服务器对象实例建立一对一的连接.

#### 存根

Java RMI 通过创建存根函数来工作。存根由 rmic 编译器生成。自 Java 1.5 以来,Java 支持在运行时动态生成存根类。编译器 rmic 会提供各种编译选项。

#### 定位对象

引导名称服务提供了用于存储对远程对象的命名引用。一个远程对象引用可以存储使用类 java.rmi.Naming 提供的基于 URL 的方法。例如,

BankAccount acct = new BankAcctImpl();

String url = "rmi://java.sun.com/account";

// bind url to remote object

java.rmi.Naming.bind(url, acct);

// look up account

acct = (BankAccount)java.rmi.Naming.lookup(url);



图7 Java RMI 工作流程

#### RMI 架构

RMI 是一个三层架构(图8)。最上面是 Stub/Skeleton layer（存根/骨架层）。方法调用从 Stub、Remote Reference Layer (远程引用层)和 Transport Layer（传输层）向下，传递给主机，然后再次经传 Transport Layer 层，向上穿过 Remote Reference Layer 和 Skeleton ，到达服务器对象。 Stub 扮演着远程服务器对象的代理的角色，使该对象可被客户激活。Remote Reference Layer 处理语义、管理单一或多重对象的通信，决定调用是应发往一个服务器还是多个。Transport Layer 管理实际的连接，并且追踪可以接受方法调用的远程对象。服务器端的 Skeleton 完成对服务器对象实际的方法调用，并获取返回值。返回值向下经 Remote Reference Layer 、服务器端的 Transport Layer 传递回客户端，再向上经 Transport Layer 和 Remote Reference Layer 返回。最后，Stub 程序获得返回值。

要完成以上步骤需要有以下几个步骤：

* 生成一个远程接口；
* 实现远程对象(服务器端程序)；
* 生成 Stub 和 Skeleton(服务器端程序)；
* 编写服务器程序 ；
* 编写客户程序 ；
* 注册远程对象；
* 启动远程对象

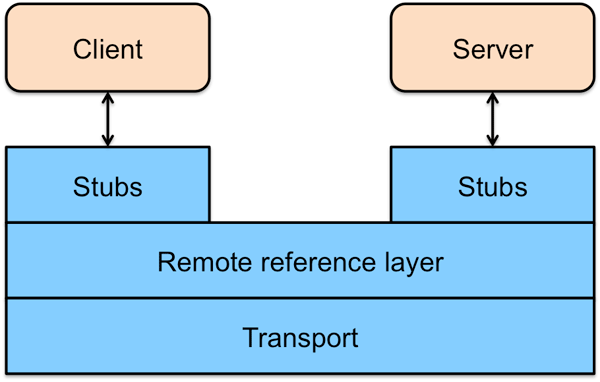


图8 Java RMI 架构

#### RMI 分布式垃圾回收

根据 Java 虚拟机的垃圾回收机制原理，在分布式环境下，服务器进程需要知道哪些对象不再由客户端引用，从而可以被删除(垃圾回收)。在 JVM中,Java 使用引用计数。当引用计数归零时，对象将会垃圾回收。在RMI,Java 支持两种操作:dirty 和 clean。本地 JVM 定期发送一个 dirty 到服务器来说明该对象仍在使用。定期重发 dirty 的周期是由服务器租赁时间来决定的。当客户端没有需要更多的本地引用远程对象时,它发送一个 clean 调用给服务器。不像 DCOM,服务器不需要计算每个客户机使用的对象,只是简单的做下通知。如果它租赁时间到期之前没有接收到任何 dirty 或者 clean 的消息，则可以安排将对象删除。

## 第三代 RPC 以及 Web Services

由于互联网的兴起，Web 浏览器成为占主导地位的用于访问信息的模型。现在的应用设计的首要任务大多数是提供用户通过浏览器来访问，而不是编程访问或操作数据。

网页设计关注的是内容。解析展现方面往往是繁琐的。传统 RPC 解决方案可以工作在互联网上，但问题是，他们通常严重依赖于动态端口分配，往往要进行额外的防火墙配置。

Web Services 成为一组协议,允许服务被发布、发现，并用于技术无关的形式。即服务不应该依赖于客户的语言、操作系统或机器架构。

Web Services 的实现一般是使用 Web 服务器作为服务请求的管道。客户端访问该服务，首先是通过一个 HTTP 协议发送请求到服务器上的 Web 服务器。Web 服务器配置识别 URL 的一部分路径名或文件名后缀并将请求传递给特定的浏览器插件模块。这个模块可以除去头、解析数据(如果需要),并根据需要调用其他函数或模块。对于这个实现流，一个常见的例子是浏览器对于 Java Servlet 的支持。HTTP 请求会被转发到 JVM 运行的服务端代码来执行处理。

### XML-RPC

XML-RPC 是1998年作为一个 RPC 消息传递协议，将请求和响应封装解析为人类可读的 XML 格式。XML 格式基于 HTTP 协议，缓解了传统企业的防火墙需要为 RPC 服务器应用程序打开额外的端口的问题。

下面是一个 XML-RPC 消息的例子:

<methodCall>

<methodName>

sample.sumAndDifference

</methodName>

<params>

<param><value><int> 5 </int></value></param>

<param><value><int> 3 </int></value></param>

</params>

</methodCall>

这个例子中，方法 sumAndDifference 有两个整数参数 5 和 3。

XML-RPC 支持的基本数据类型是:int、string、boolean、double 和 dateTime.iso8601。此外,还有 base64 类型用于编码任意二进制数据。array 和 struct 允许定义数组和结构。

XML-RPC 不限制语任何特定的语言，也不是一套完整的软件来处理远程过程，诸如存根生成、对象管理和服务查找都不在协议内。现在有很多库针可以针对不同的语言，比如 Apache XML-RPC 可以用于 Java、Python 和 Perl。

XML-RPC 是一个简单的规范(约7页)，没有雄心勃勃的目标——它只关注消息，而并不处理诸如垃圾收集、远程对象、远程过程的名称服务和其他方面的问题。然而，即使没有广泛的产业支持，简单的协议却能广泛采用。

### SOAP

SOAP（Simple Object Access Protocol，简单对象访问协议），是以 XML-RPC 规范作为创建 SOAP 的依据，成立于1998年，获得微软和 IBM 的大力支持。该协议在创建初期只作为一种对象访问协议，但由于 SOAP 的发展，其协议已经不单只是用于简单的访问对象，所以这种 SOAP 缩写已经在标准的1.2版后被废止了。1.2版在2003年6月24日成为 W3C 的推荐版本。SOAP 指定 XML 作为无状态的消息交换格式，包括了 RPC 式的过程调用。

有关 SOAP 的标准可以参阅 <https://www.w3.org/TR/soap/>。

SOAP 只是一种消息格式，并未定义垃圾回收、对象引用、存根生成和传输协议。

下面是一个简单的例子：

<?xml version='1.0' ?>

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">

<env:Header>

<m:reservation xmlns:m="http://travelcompany.example.org/reservation"

env:role="http://www.w3.org/2003/05/soap-envelope/role/next"

env:mustUnderstand="true">

<m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>

<m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>

</m:reservation>

<n:passenger xmlns:n="http://mycompany.example.com/employees"

env:role="http://www.w3.org/2003/05/soap-envelope/role/next"

env:mustUnderstand="true">

<n:name>Åke Jógvan Øyvind</n:name>

</n:passenger>

</env:Header>

<env:Body>

<p:itinerary

xmlns:p="http://travelcompany.example.org/reservation/travel">

<p:departure>

<p:departing>New York</p:departing>

<p:arriving>Los Angeles</p:arriving>

<p:departureDate>2001-12-14</p:departureDate>

<p:departureTime>late afternoon</p:departureTime>

<p:seatPreference>aisle</p:seatPreference>

</p:departure>

<p:return>

<p:departing>Los Angeles</p:departing>

<p:arriving>New York</p:arriving>

<p:departureDate>2001-12-20</p:departureDate>

<p:departureTime>mid-morning</p:departureTime>

<p:seatPreference/>

</p:return>

</p:itinerary>

<q:lodging

xmlns:q="http://travelcompany.example.org/reservation/hotels">

<q:preference>none</q:preference>

</q:lodging>

</env:Body>

</env:Envelope>

其中<soap:Envelope>是 SOAP 消息中的根节点，是 SOAP 消息中必须的部分。<soap:Header>是 SOAP 消息中可选部分，是指消息头。<soap:Body>是 SOAP 中必须部分，是指消息体。

上面例子的 SOAP 消息结构如下：

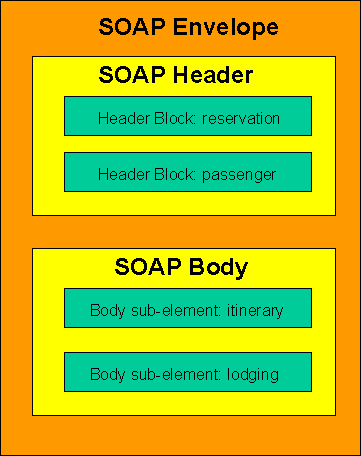


图9 SOAP 消息结构

SOAP 它只是提供了一个标准化的消息结构，为了实现它往往需要用 WSDL 来描述 Web Services 的方法。WSDL (Web Services Description Language) 是基于 XML 的一种用于描述 Web Services 以及如何访问 Web Services 的语言。

WSDL 文档包括以下几个部分:

* 类型（Types）:定义了 Web Services 使用的数据类型；
* 消息（n/a）:描述使用消息的数据元素或参数；
* 接口（Interface）:描述服务提供的操作。这包括操作以及每个操作所使用的输入和输出消息；
* 绑定（Binding）:为每个端口定义消息格式和协议细节。例如,它可以定义 RPC 式的消息；
* 服务（Service）：系统功能相关的集合，包括其关联的接口、操作、消息等；
* 终点（Endpoint）：定义了地址或者 Web Services 的连接点；
* 操作（Operation）：定义了 SOAP 的动作，以及消息编码的方式。

下面是一个 WSDL 2.0 版本的例子：

<?xml version="1.0" encoding="UTF-8"?>

<description xmlns="http://www.w3.org/ns/wsdl"

xmlns:tns="http://www.tmsws.com/wsdl20sample"

xmlns:whttp="http://schemas.xmlsoap.org/wsdl/http/"

xmlns:wsoap="http://schemas.xmlsoap.org/wsdl/soap/"

targetNamespace="http://www.tmsws.com/wsdl20sample">

<documentation>

This is a sample WSDL 2.0 document.

</documentation>

<!-- Abstract type -->

<types>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns="http://www.tmsws.com/wsdl20sample"

targetNamespace="http://www.example.com/wsdl20sample">

<xs:element name="request"> ... </xs:element>

<xs:element name="response"> ... </xs:element>

</xs:schema>

</types>

<!-- Abstract interfaces -->

<interface name="Interface1">

<fault name="Error1" element="tns:response"/>

<operation name="Get" pattern="http://www.w3.org/ns/wsdl/in-out">

<input messageLabel="In" element="tns:request"/>

<output messageLabel="Out" element="tns:response"/>

</operation>

</interface>

<!-- Concrete Binding Over HTTP -->

<binding name="HttpBinding" interface="tns:Interface1"

type="http://www.w3.org/ns/wsdl/http">

<operation ref="tns:Get" whttp:method="GET"/>

</binding>

<!-- Concrete Binding with SOAP-->

<binding name="SoapBinding" interface="tns:Interface1"

type="http://www.w3.org/ns/wsdl/soap"

wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"

wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">

<operation ref="tns:Get" />

</binding>

<!-- Web Service offering endpoints for both bindings-->

<service name="Service1" interface="tns:Interface1">

<endpoint name="HttpEndpoint"

binding="tns:HttpBinding"

address="http://www.example.com/rest/"/>

<endpoint name="SoapEndpoint"

binding="tns:SoapBinding"

address="http://www.example.com/soap/"/>

</service>

</description>

### Microsoft .NET Remoting

从微软的产品角度来看，可以说 .NET Remoting 就是 DCOM 的一种升级，它改善了很多功能，并极好的融合到 .NET 平台下。Microsoft .NET Remoting 提供了一种允许对象通过应用程序域与另一对象进行交互的框架。

.NET Remoting 提供了一种允许对象通过应用程序域与另一对象进行交互的框架。这种框架提供了多种服务，包括激活和生存期支持，以及负责与远程应用程序进行消息传输的通讯通道。格式化程序用于在消息通过通道传输之前，对其进行编码和解码。应用程序可以在注重性能的场合使用二进制编码，在需要与其他远程处理框架进行交互的场合使用 XML 编码。在从一个应用程序域向另一个应用程序域传输消息时，所有的 XML 编码都使用 SOAP 协议。出于安全性方面的考虑，远程处理提供了大量挂钩，使得在消息流通过通道进行传输之前，安全接收器能够访问消息和序列化流

#### .NET Remoting 对象

有三类对象可以配置为 .NET Remoting 对象。您可以根据应用程序的需要来选择对象类型：

* Single Call（单一调用对象）： Single Call 能且只能为一个请求提供服务。在需要对象完成的工作量有限的情况下 Single Call 非常有用。Single Call 对象通常不要求存储状态信息，并且不能在方法调用之间保留状态信息。但是，Single Call对象可以配置为负载平衡模式。
* Singleton Objects（单一元素对象）： Singleton Objects 可以为多个客户端提供服务，因此可以通过保存客户端调用的状态信息来实现数据共享。当客户端之间需要明确地共享数据，并且不能忽略创建和维护对象的开销时，这类对象非常有用。
* Client-Activated Objects (CAO，客户端激活的对象)： CAO 是服务器端的对象，将根据来自客户端的请求激活这些对象。这种激活服务器对象的方法与传统的 COM coclass 激活方法非常相似。当客户端使用“new”运算符提交对服务器对象的请求时，会向远程应用程序发送一个激活请求消息。随后，服务器创建被请求类的实例，并向调用它的客户端应用程序返回 ObjRef。然后，使用此 ObjRef 在客户端上创建代理。客户端的方法调用将在代理上执行。客户端激活的对象可以为特定的客户端（不能跨越不同的客户端对象）保存方法调用之间的状态信息。每个“new”调用都会向服务器类型的独立实例返回代理。

在 .NET Remoting 中，可以通过以下方式在应用程序之间传递对象：

* 作为方法调用中的参数，例如：public int myRemoteMethod (MyRemoteObject myObj)
* 方法调用的返回值,例如：public MyRemoteObject myRemoteMethod(String myString)
* 访问 .NET 组件的属性或字段得到的值,例如：myObj.myNestedObject

对于 Marshal By Value (MBV,按值封送)的对象，当它在应用程序之间传递时，将创建一个完整的副本。

对于 Marshal By Reference (MBR，按引用封送)的对象，当它在应用程序之间传递时，将创建该对象的引用。当对象引用 (ObjRef) 到达远程应用程序后，将转变成“代理”返回给原始对象。 　　 下面是一个简单 .NET Remoting 服务器对象的代码示例：

using System;

using System.Runtime.Remoting;

namespace myRemoteService

{

// Well Known Web Service object

public class myRemoteObject : MarshalByRefObject

{

// Method myRemoteMethod

public String myRemoteMethod(String s)

{

return "Hello World";

}

}

}

下面是客户端代码来访问这个对象：

using System;

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels;

using System.Runtime.Remoting.Channels.Http;

using myRemoteService;

public class Client

{

public static int Main(string[] args)

{

ChannelServices.RegisterChannel(new HttpChannel());

// Create an instance of a myRemoteObject class

myRemoteObject myObj = ( myRemoteObject)Activator.GetObject(typeof(myRemoteObject),

"http://myHost:7021/host/myRemoteObject.soap");

myObj. myRemoteMethod ("Hello World");

return 0;

}

}

#### 租用生存期

对于那些具有在应用程序之外传送的对象引用的对象，将创建一个租用。租用具有一个租用时间。如果租用时间为 0，则租用过期，对象就断开与 .NET Romoting 框架的连接。一旦 AppDomain 中的所有对象引用都被释放，则下次垃圾回收时，该对象将被回收。租用控制了对象的生存期。

对象有默认的租用阶段。当客户端要在同一服务器对象中维护状态时，可以通过许多方法扩展租用阶段，使对象继续生存:

* 服务器对象可以将其租用时间设置为无限，这样 Remoting 在垃圾回收时就不会回收此对象。
* 客户端可以调用 RemotingServices.GetLifetimeService 方法，从 AppDomain 的租用管理器获取服务器对象的租用。然后，客户端可以通过 Lease 对象调用 Lease.Renew 方法以延长租用。
* 客户端可用 AppDomain 的租用管理器为特定的租用注册负责人。当 Remoting 对象的租用过期时，租用管理器将通知负责人提出续租的申请。
* 如果设置了 ILease::RenewOnCallTime 属性，则每次调用 Remoting 对象时，都会用 RenewOnCallTime 属性指定的总时间更新租用。

#### 集成 .NET Remoting 对象

.NET Remoting 对象可以集成在：

* 托管可执行项: .NET Remoting 对象可以集成在任何常规的 .NET EXE 或托管服务中。
* .NET 组件服务: .NET Remoting 对象可以集成在 .NET 组件服务基础结构中，以便利用各种 COM+ 服务，例如：事务、JIT 和对象池等。
* IIS : .NET Remoting 对象可以集成在 Internet Information Server (IIS) 中。默认情况下，集成在 IIS 中的 Remoting 对象通过 HTTP 通道接收消息。要在 IIS 中集成 Remoting 对象，必须创建一个虚拟的根，并将 remoting.config 文件复制到其中。包含 Remoting 对象的可执行文件或 DLL 应放在 IIS 根指向的目录下的 bin 目录中。需要注意的是，IIS 根名称应该与配置文件中指定的应用程序名称相同。当第一个消息到达应用程序时，Remoting 配置文件会自动加载。使用这种方法，可以提供 .NET Remoting 对象作为 Web 服务。

下面是一个 Remoting.cfg 文件的例子：

<configuration>

<system.runtime.remoting>

<application name="RemotingHello">

<service>

<wellknown mode="SingleCall"

type="Hello.HelloService, Hello"

objectUri="HelloService.soap" />

</service>

<channels>

<channel port="8000"

type="System.Runtime.Remoting.Channels.Http.HttpChannel,

System.Runtime.Remoting" />

</channels>

</application>

</system.runtime.remoting>

</configuration>

#### 通道服务 (System.Runtime.Remoting.Channels)

.NET 应用程序和 AppDomain 之间使用消息进行通信。.NET 通道服务为这一通信过程提供了基础传输机制。

.NET 框架提供了 HTTP 和 TCP 通道，但是第三方也可以编写并使用自己的通道。默认情况下，HTTP 通道使用 SOAP 进行通信，TCP 通道使用二进制有效负载。

通过使用可以编写到集成混合应用程序中的自定义通道，可以插入通道服务（使用 IChannel）。

下面是一个加载通道服务的例子：

public class myRemotingObj

{

HttpChannel httpChannel;

TcpChannel tcpChannel;

public void myRemotingMethod()

{

httpChannel = new HttpChannel();

tcpChannel = new TcpChannel();

ChannelServices.RegisterChannel(httpChannel);

// Register the HTTP Channel

ChannelServices.RegisterChannel(tcpChannel);

// Register the TCP Channel

}

}

#### 序列化格式化程序 (System.Runtime.Serialization.Formatters)

.NET 序列化格式化程序对 .NET 应用程序和 AppDomain 之间的消息进行编码和解码。在 .NET 运行时中有两个本地格式化程序，分别为二进制 (System.Runtime.Serialization.Formatters.Binary) 和 SOAP (System.Runtime.Serialization.Formatters.Soap)。

通过实现 IRemotingFormatter 接口，并将其插入到上文介绍的通道中，可以插入序列化格式化程序。这样，您可以灵活地选择通道和格式化程序的组合方式，采用最适合应用程序的方案。

例如：您可以采用 HTTP 通道和二进制格式化程序（串行化二进制数据），也可以采用 TCP 通道和 SOAP 格式。

#### Remoting 上下文

上下文是一个包含共享公共运行时属性的对象的范围。某些上下文属性的例子是与同步和线程紧密相关的。当 .NET 对象被激活时，运行时将检查当前上下文是否一致，如果不一致，将创建新的上下文。多个对象可以同时在一个上下文中运行，并且一个 AppDomain 中可以有多个上下文。

一个上下文中的对象调用另一个上下文中的对象时，调用将通过上下文代理来执行，并且会受组合的上下文属性的强制策略影响。新对象的上下文通常是基于类的元数据属性选择的。

可以与上下文绑定的类称作上下文绑定类。上下文绑定类可以具有称为上下文属性的专用自定义属性。上下文属性是完全可扩展的，您可以创建这些属性并将它们附加到自己的类中。与上下文绑定的对象是从 System.ContextBoundObject 导出的。

#### .NET Remoting 元数据和配置文件

##### 元数据

.NET 框架使用元数据和程序集来存储有关组件的信息，使得跨语言编程成为可能。.NET Remoting 使用元数据来动态创建代理对象。在客户端创建的代理对象具有与原始类相同的成员。但是，代理对象的实现仅仅将所有请求通过 .NET Remoting 运行时转发给原始对象。序列化格式化程序使用元数据在方法调用和有效负载数据流之间来回转换。

客户端可以通过以下方法获取访问 Remoting 对象所需的元数据信息：

* 服务器对象的 .NET 程序集 : 服务器对象可以创建元数据程序集，并将其分发给客户端。在编译客户端对象时，客户端对象可以引用这些程序集。在客户端和服务器都是托管组件的封闭环境中，这种方法非常有用。
* Remoting 对象可以提供 WSDL文件，用于说明对象及其方法。所有可以根据 WSDL 文件读取和生成 SOAP 请求的客户端都可以调用此对象，或使用 SOAP 与之通信。使用与 .NET SDK 一同分发的 SOAPSUDS.EXE 工具，.NET Remoting 服务器对象可以生成具有元数据功能的 WSDL 文件。当某个组织希望提供所有客户都能访问和使用的公共服务时，这种方法非常有用。
* .NET 客户可以使用 SOAPSUDS 工具从服务器上下载 XML 架构（在服务器上生成），生成仅包含元数据（没有代码）的源文件或程序集。您可以根据需要将源文件编译到客户端应用程序中。如果多层应用程序中某一层的对象需要访问其他层的 Remoting 对象，则经常使用此方法。

##### 配置文件

　　 配置文件（.config 文件）用于指定给定对象的各种 Remoting 特有信息。通常情况下，每个 AppDomain 都有自己的配置文件。使用配置文件有助于实现位置的透明性。配置文件中指定的详细信息也可以通过编程来完成。使用配置文件的主要好处在于，它将与客户端代码无关的配置信息分离出来，这样在日后更改时仅需要修改配置文件，而不用编辑和重新编译源文件。.NET Remoting 的客户端和服务器对象都使用配置文件。

典型的配置文件包含以下信息及其他信息：

* 集成应用程序信息
* 对象名称
* 对象的 URI
* 注册的通道（可以同时注册多个通道）
* 服务器对象的租用时间信息

下面是一个配置文件示例：

<configuration>

<system.runtime.remoting>

<application name="HelloNew">

<lifetime leaseTime="20ms" sponsorshipTimeout="20ms"

renewOnCallTime="20ms" />

<client url="http://localhost:8000/RemotingHello">

<wellknown type="Hello.HelloService, MyHello"

url="http://localhost:8000/RemotingHello/HelloService.soap" />

<activated type="Hello.AddService, MyHello"/>

</client>

<channels>

<channel port="8001"

type="System.Runtime.Remoting.Channels.Http.HttpChannel,

System.Runtime.Remoting" />

</channels>

</application>

</system.runtime.remoting>

</configuration>

#### .NET Remoting 方案

了解 .NET Remoting 如何工作之后，让我们来看一下各种方案，分析如何在不同的方案中充分发挥 .NET Remoting 的优势。下表列出了可能的客户端/服务器组合，以及默认情况下采用的底层协议和有效负载。请注意，.NET Remoting 框架是可扩展的，您可以编写自己的通信通道和序列化格式化程序。

| **客户端** | **服务器** | **有效负载** | **协议** |
| --- | --- | --- | --- |
| .NET 组件 | .NET 组件 | SOAP/XML | http |
| .NET 组件 | .NET 组件 | 二进制 | TCP |
| 托管/非托管 | .NET Web 服务 | SOAP/XML | http |
| .NET 组件 | 非托管的传统 COM 组件 | NDR（网络数据表示形式） | DCOM |
| 非托管的传统 COM 组件 | .NET 组件 | NDR | DCOM |

##### 使用 HTTP-SOAP

Web 服务是可以通过 URL 寻址的资源，并通过编程向需要使用这些资源的客户端返回信息。客户端使用 Web Services 时不必考虑其实现细节。Web Services 使用称为“合约”的严格定义的接口，此接口采用 Web 服务说明语言 (WSDL) 文件描述。

.NET Remoting 对象可以集成在 IIS 中，作为 Web 服务提供。任何可以使用 WSDL 文件的客户端都可以按照 WSDL 文件中指定的合约，对 Remoting 对象执行 SOAP 调用。IIS 使用 ISAPI 扩展将这些请求路由到相应的对象。这样，Remoting 对象就可以作为 Web 服务对象来使用，从而充分发挥 .NET 框架基础结构的作用。如果您希望不同平台/环境的程序均能够访问对象，可以采用这种配置。这种配置便于客户端通过防火墙访问您的 .NET 对象。

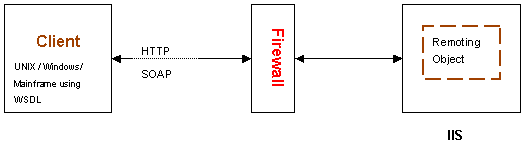


图10 .NET 使用 HTTP-SOAP

##### 使用 SOAP-HTTP 通道

默认情况下，HTTP 通道使用 SOAP 格式化程序。因此，如果客户端需要通过 Internet 访问对象，可以使用 HTTP 通道。因为这种方法使用 HTTP，所以此配置允许通过防火墙远程访问 .NET 对象。只需要按前一节中介绍的方法将这些对象集成在 IIS 中，即可将它配置为 Web 服务对象。随后，客户端就可以读取这些对象的 WSDL 文件，使用 SOAP 与 Remoting 对象通信。

##### 使用 TCP 通道

默认情况下，TCP 通道使用二进制格式化程序。此格式化程序以二进制格式对数据进行序列化，并使用原始 socket 在网络中传送数据。如果对象部署在受防火墙保护的封闭环境中，此方法是理想的选择。这种方法使用 socket 在对象之间传递二进制数据，因此性能极佳。由于它使用 TCP 通道来提供对象，因此在封闭环境中具有低开销的优点。由于防火墙和配置的问题，此方法不宜在 Internet 上使用。

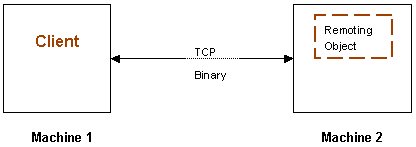


图11 .NET 使用 TCP 通道

##### 使用非托管的 COM 组件

可以通过 COM Interop Service 调用非托管的传统 COM 组件。当 .NET Remoting 客户端对象创建 COM 对象的实例时，该对象通过运行时可调用包装程序 (RCW) 来提供。其中，RCW 担当真正的非托管对象的代理。对于 .NET 客户，这些包装程序看起来和 .NET 客户端的任何其他托管类一样。但实际上，它们仅仅是托管 (.NET) 和非托管 (COM) 代码之间的封送调用。

同样地，您可以将 .NET Remoting 服务器对象提供给传统 COM 客户端。当 COM 客户端创建 .NET 对象的实例时，该对象通过 COM 可调用包装程序 (CCW) 来提供。其中，CCW 担当真正的托管对象的代理。 　　 这两种方案都使用 DCOM 通信。如果环境中既有传统的 COM 组件，又有 .NET 组件，那么这种互操作性将为您提供便利。

#### 总结

Microsoft .NET 框架提供了强大、可扩展、独立于语言的框架，适合开发可靠、可伸缩的分布式系统。.NET Romoting 框架提供了根据系统需求进行远程交互的强大手段。.NET Remoting 实现了与 Web 服务的无缝集成，并提供了一种方法，可以提供 .NET 对象以供跨平台访问。

### Java 中的 XML Web Services

Java RMI 与远程对象进行交互，其实现是需要基于 Java 的模型。此外,它没有使用 Web Services 和基于 HTTP 的消息传递。现在，已经出现了大量的软件来支持基于 Java 的 Web Services。JAX-WS (Java API for XML Web Services) 就是作为 Web Services 消息息和远程过程调用的规范。它允许一个调用基于Java的web服务使用Java RMI(即。,相对透明的程序员)。JAX-WS 的一个目标是平台互操作性。其 API 使用 SOAP 和WSDL。双方不需要 Java 环境。

#### 创建一个 RPC 端点

在服务器端,进行下面的步骤来创建一个 RPC 端点:

* 定义一个接口(Java接口)；
* 实现服务；
* 创建一个发布者,用于创建服务的实例，并发布一个服务名字。

在客户端:

* 创建一个代理(客户端存根)。wsimport 命令根据 WSDL 文档,创建一个客户机存根；
* 编写一个客户端，通过代理创建远程服务的一个实例(存根),调用它的方法。

JAX-RPC 执行流程如下:

* Java 客户机调用存根上的方法(代理)；
* 存根调用适当的 Web 服务；
* Web 服务器被调用并指导 JAX-WS 框架；
* 框架调用实现；
* 实现返回结果给该框架；
* 该框架将结果返回给 Web 服务器；
* 服务器将结果发送给客户端存根；
* 客户端存根返回信息给调用者；

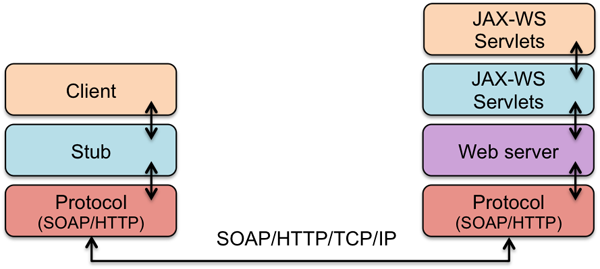


图12 JAX-WS 调用流程

### 超越 SOAP

SOAP 虽然仍然是广泛部署应用，但在许多环境中很多厂商已经抛弃了 SOAP，转而使用其他更轻量、更容易理解、或者与 Web 交互模型更干净的机制。例如,Google 的 API 在2006年后就不再支持 SOAP 接口，而是使用AJAX、XML-RPC 和 REST 作为替代。一个匿名的微软员工批评 SOAP 过于复杂，因为“我们希望我们的工具来阅读它,而不是人”。不管上述言论是否准确，有一点是可以肯定的，SOAP 显然是一个复杂和高度冗长的格式。

#### AJAX

Web 浏览器最初的设计，是为 Web 页面提供非动态的交互模型。Web 浏览器是建立在同步的请求-响应(request-response)的交互模型。发送一个请求到服务器,服务器返回整个页面。在当时没有更新部分页面的好方法，而唯一可行的方法是利用帧，即将不同的页面加载到每一帧，其实现是笨重的，也有很大的限制性。而改变了这一切的关键因素是：

* 文档对象模型（Document Object Model）和 JavaScript 的出现,使得可以以编程方式来更改 Web 页面的各个部分；
* AJAX 提供了与服务器以非阻塞方式进行交互，即允许底层 JavaScript 在等待服务器结果时，用户仍然可以与页面进行交互。

AJAX 全称是 Asynchronous JavaScript And XML（异步的 JavaScript 和 XML）。让我们看看这些三项:

* 它是异步的,因为客户端等待服务器结果不会被阻塞；
* AJAX 集成到了 JavaScript，作为浏览器解释 Web 页面的一部分。JavaScript 使用 HTTPRequest 来调用 AJAX 请求。JavaScript 也可能修改文档对象模型，定义了页面的样子；
* 数据以 XML 文档形式发送和接收。（在后期发展中，AJAX 也支持其他的数据格式，比如 JSON）

AJAX 在推动 Web 2.0 的过程中发挥了重要的，比如产生了很多高度交互的服务,如Google Maps、Writely等。基本上,它允许 JavaScript 发出HTTP 请求，获取和处理结果，刷新局部页面元素而不是整个页面。在大多数浏览器请求的格式如下:

new XMLHttpRequest()

xmlhttp.open(“HEAD”, “index.html”, true)Tell object:

### REST

SOAP 在创建自己的消息传递协议时是基于HTTP，但实际上 REST (REpresentational State Transfer) 的方式才是保持 Web 的原理和使用 HTTP 协议的核心部分。

原始的 HTTP 协议已经定义了四个命令，清晰地映射到各种数据(定义为“资源”)操作:

* PUT (插入)
* GET (选择)
* POST (更新)
* DELETE (删除)

REST 其背后的理念是使用这些 HTTP 命令来请求和操作数据。作为 HTTP协议的一部分，REST 使用 URL 来引用对象和操作。考虑这个 HTTP 操作列表的例子:

HTTP GET //www.waylau.com/parts

这个命令将返回一个 XML 文档,其中包含部分的列表。注意,返回的不是一个网页,只是一个包含所请求的数据 XML 数据结构。

<?xml version="1.0"?>

<p:Parts xmlns:p="http://www.waylau.com"

xmlns:xlink="http://www.w3.org/1999/xlink">

<Part id="00345" xlink:href="http://www.waylau.com/parts/00345"/>

<Part id="00346" xlink:href="http://www.waylau.com/parts/00346"/>

<Part id="00347" xlink:href="http://www.waylau.com/parts/00347"/>

<Part id="00348" xlink:href="http://www.waylau.com/parts/00348"/>

</p:Parts>

要特定部分的详细信息,发送一个HTTP get 命令:

HTTP GET //www.waylau.com/parts/00345

这将返回一个特定的信息部分:

<?xml version="1.0"?>

<p:Part xmlns:p="http://www.waylau.com"

xmlns:xlink="http://www.w3.org/1999/xlink">

<Part-ID>00345</Part-ID>

<Name>Widget-A</Name>

<Description>This part is used within the frap assembly</Description>

<Specification xlink:href="http://www.waylau.com/parts/00345/specification"/>

<UnitCost currency="USD">0.10</UnitCost>

<Quantity>10</Quantity>

</p:Part>

注意，上面例子简化了 partid 作为 URL 的参数。例如：

HTTP GET //www.waylau.com/parts?partid=00345

REST 不是 RPC，但也有类似的请求-响应模式。制定透明度请求、封送数据、解析响应这些不属于 REST。REST 应用非常广泛，如 Yahoo! Search API、Ruby on Rails、Twiter 和 Open Zing Services 等。

### Google Protocol Buffers:封送处理

有些时候，不仅仅是为了 RPC 和 Web Services 的需要，程序员只是想简化对网络上的数据的封送编组和解封的操作。Google Protocol Buffers 就是为序列化结构化数据提供了一种有效的机制，使它容易对网络上的数据进行编码和解码。Protocol Buffers 是一个紧凑的二进制格式比 XML 更简单、体积更小、速度更快。他们是独立于语言的，只定义数据类型。每个消息是对数据名称、类型和值的结构化集合。消息结构定义在一个高级别的格式，类似于许多接口定义语言。然后文件可以根据你选择的语言来编译转换成与该语言相应的格式。Protocol Buffers 在 Google 中广泛使用。目前已经有超过48000种不同的消息类型定义。Protocol Buffers 可以被运用在类 RPC 消息传递以及持久性存储(将数据转换成标准的串行形式写入到一个文件中）。下面是一个定义 Protocol Buffers 的例子:

message Person {

required string name = 1;

required int32 id = 2;

optional string email = 3;

enum PhoneType {

MOBILE = 0;

HOME = 1;

WORK = 2;

}

message PhoneNumber {

required string number = 1;

optional PhoneType type = 2 [default = HOME];

}

repeated PhoneNumber phone = 4;

}

请注意，这只定义了数据结构，而不是功能。使用这个结构的例子是:

Person person;

person.set\_name("John Doe");

person.set\_id(1234);

person.set\_email("jdoe@example.com");

fstream output("myfile", ios::out | ios::binary);

person.SerializeToOstream(&output);

即使与紧凑的 XML 版本相比，Protocol Buffers 在时间和空间方面，解析将更加有效。下面是两者的对比.

这个是 XML 格式：

<person>

<name>John Doe</name>

<email>jdoe@example.com</email>

</person>

这个没有编译的 Protocol Buffers 格式:

person {

name: "John Doe"

email: "jdoe@example.com"

}

Protocol Buffers 产生的二进制消息大约是28字节长，解析耗时大概需要100-200ns。相比之下,XML 版本需要69个字节长(是 Protocol Buffers 的 2.5倍),耗时是5000-10000ns(是 Protocol Buffers 的 50倍)。

### JSON

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于 ECMAScript 的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括C、C++、C#、Java、JavaScript、Perl、Python 等）。这些特性使 JSON 成为理想的数据交换语言。易于人阅读和编写，同时也易于机器解析和生成。JSON 不是一个诸如 Google Protocol Buffers 的二进制格式，因此适合使用基于 HTTP的消息传递。JSON 是可以作为 XML 替代品，在远程过程调用中，很多语言都支持 JSON-RPC。记住，这只是一个消息传递格式，JSON 并没有试图提供 RPC 库来支持服务发现、绑定、托管和垃圾收集。

## 参考引用

* [http://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf](https://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf)
* <https://www.cs.rutgers.edu/~pxk/417/notes/03-rpc.html>
* [http://queue.acm.org/detail.cfm?id=1142044](https://queue.acm.org/detail.cfm?id=1142044)
* <https://en.wikipedia.org/wiki/Web_Services_Description_Language>
* <https://msdn.microsoft.com/en-us/library/ms973864.aspx>
* <https://msdn.microsoft.com/en-us/library/bb985129.aspx>

[Maven 项目构建失败“文件名、目录名或卷标语法不正确”](https://waylau.com/maven-file-name-syntax-incorrect/)  [尝试从 Windows 完全切换到 Linux](https://waylau.com/switch-from-windows-to-linux/)