目　录

致谢

当前文档 《分布式 Java(Distributed Java)》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-04-26。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/distributed-java

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Summary

Summary

Introduction

Distributed Java.《分布式 Java》

Distributed Java.Let’s READ!

《分布式 Java》是一本关于 Java 分布式应用的学习教程，是对市面上基于 Java 的分布式系统最佳实践的技术总结。图文并茂，并通过大量实例让你走近 Java 的世界！

本书业余时间所著，水平有限、时间紧张，难免疏漏，欢迎指正，点此提问。感谢您的参与！

书中所有实例，在https://github.com/waylau/distributed-java 的 `samples` 目录下，代码遵循《Java 编码规范》。另外有 GitBook 的版本方便阅读 http://waylau.gitbooks.io/distributed-java。

从目录开始阅读吧！

Contact 联系作者:

- Blog:waylau.com
- Gmail: waylau521@gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

基础概念

基础概念

概述

本书主要讲解的是关于 Java 在分布式系统中的应用。本书的读者假设都具备了 Java 的基础知识。
如果你需要了解有关 Java 基础知识，可以参阅笔者的另外一本开源书《Java 编程要点》。

分布式系统的概念

《分布式系统原理与范型》一书中是这样定义分布式系统的：

> 分布式系统是若干独立计算机的集合，这些计算机对于用户来说就像是单个相关系统。

这里面包含了2个含义：

- 硬件独立：计算机机器本身是独立的
- 软件统一：对于用户来说，他们就像是跟单个系统打交道

分布式系统的扩展和升级都比较容易。分布式系统某些节点故障，不影响整理可用。用户和应用程序交互时，不会察觉哪些部分正在替换或者维修，也不会感知到新部分的加入。

万维网就是一个分布式文档模型的简单例子。要查看某个文档，用户只需要电机相关的链接即可，文档就会呈现在屏幕上。用户无需关心文档是位于哪个位置，是由什么服务器处理。概念上讲 Web 看起来就是一个单独的服务器。而实际上，Web 在物理上是分布到非常多的服务器上的。

如何设计分布式系统

设计分布式系统的本质就是

“如何合理将一个系统拆分成多个子系统部署到不同机器上”

分布式系统的设计应考虑以下几个问题：

- 系统如何拆分为子系统？
- 如何规划子系统间的通信？
- 如何让子系统可以扩展？
- 通信过程中的安全如何考虑？
- 子系统的可靠性如何保证？
- 数据的一致性是如何实现的？

本书就是针对分布式系统中常见的问题进行探讨。

Java 分布式系统

上面讲到的都是分布式系统的通用的概念，那么采用 Java 来实现分布式系统有什么好处呢？

- Java 编程语言是一种通用的、并行的、基于类的、面向对象的语言。它被设计得非常简单，这样程序员可以在该语言上流畅的交流。Java 编程语言与 C 和 C++ 有关联，但组织却截然不同，其中也省略了其他语言的一些用法，比如指针。它的目的是作为一个生产性语言，而不是一个研究性语言，因此，在设计上避免了包括新的和未经考验的功能。更多可以参考《Java 编程要点》。
- Java 天然支持分布式应用。
- Java 分布式系统应用广泛，且久经考验，很多大公司比如 Amazon，Linkedin，阿里巴巴等都提供了很多 Java 分布式系统技术方案。
- Java 拥有丰富的中间件框架，避免了很多底层编码的复杂性，帮助你站在巨人的肩膀上。

本书所讲的示例、代码也是基于 Java 语言或者 Java 框架。

通信原理

IPC

IPC

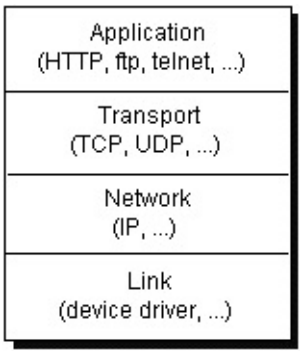
网络通信基础

概述

概述

网络基础

在互联网上之间的通信交流，一般是基于 TCP (Transmission Control Protocol，传输控制协议) 或者 UDP (User Datagram Protocol，用户数据报协议) ，如下图:



编写 Java 应用，我们只需关注于应用层 （application layer），而不用关心 TCP 和 UDP 所在的传输层是如何实现的。java.net 包含了你编程所需的类，这些类是与操作系统无关的。比如 URL, URLConnection, Socket, 和 ServerSocket 类是使用 TCP 连接网络的，DatagramPacket, DatagramSocket, 和 MulticastSocket 类是用于 UDP 的。

Java 支持的协议只有 TCP 和 UDP ，以及在建立在 TCP 和 UDP 之上其他应用层协议。所有其他传输层、网际层和更底层的协议，如 ICMP、IGMP、ARP、RARP、RSVP 和其他协议 在 Java 中只能链接到原生代码来实现。

TCP

TCP (Transmission Control Protocol) 是面向连接的、提供端到端可靠的数据流(flow of

data)。TCP 提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。

“面向连接”就是在正式通信前必须要与对方建立起连接。这一过程与打电话很相似，先拨号振铃，等待对方摘机说“喂”，然后才说明是谁。

三次握手

TCP 是基于连接的协议，也就是说，在正式收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次“握手”才能建立起来，简单的讲就是：

1. 主机 A 向主机 B 发出连接请求数据包：“我想给你发数据，可以吗？”；
2. 主机 B 向主机 A 发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包：“可以，你来吧”；
3. 主机 A 再发出一个数据包确认主机 B 的要求同步：“好的，我来也，你接着吧！”

三次“握手”的目的是使数据包的发送和接收同步，经过三次“对话”之后，主机 A 才向主机 B 正式发送数据。

可以详见我的另外一篇博客《TCP 协议的三次握手、四次分手》。

如何保证数据的可靠

TCP 通过下列方式来提供可靠性：

- 应用数据被分割成 TCP 认为最适合发送的数据块。这和 UDP 完全不同，应用程序产生的数据报长度将保持不变。由 TCP 传递给 IP 的信息单位称为报文段或段（segment）。
- 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。（可自行了解 TCP 协议中自适应的超时及重传策略）。
- 当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒。
- TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错， TCP 将丢弃这个报文段和不确认收到此报文段（希望发送端超时并重发）。
- 既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。如果必要， TCP 将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。
- 既然 IP 数据报会发生重复， TCP 的接收端必须丢弃重复的数据。
- TCP 还能提供流量控制。TCP 连接的每一方都有固定大小的缓冲空间。 TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

UDP

UDP (User Datagram Protocol) 不是面向连接的，主机发送独立的数据报（datagram）给其他主机，不保证数据到达。由于 UDP 在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

而无连接是一开始就发送信息（严格说来，这是没有开始、结束的），只是一次性的传递，是先不需要接受方的响应，因而在一定程度上也无法保证信息传递的可靠性了，就像写信一样，我们只是将信寄出去，却不能保证收信人一定可以收到。

TCP 和 UDP 如何抉择

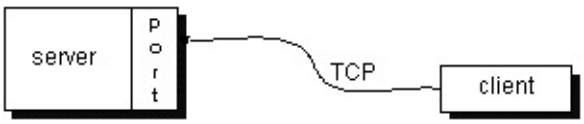
TCP 是面向连接的，有比较高的可靠性， 一些要求比较高的服务一般使用这个协议，如FTP、Telnet、SMTP、HTTP、POP3 等，而 UDP 是面向无连接的，使用这个协议的常见服务有 DNS、SNMP、QQ 等。对于 QQ 必须另外说明一下，QQ2003 以前是只使用UDP协议的，其服务器使用8000端口，侦听是否有信息传来，客户端使用 4000 端口，向外发送信息（这也就不难理解在一般的显IP的QQ版本中显示好友的IP地址信息中端口常为4000或其后续端口的原因了），即QQ程序既接受服务又提供服务，在以后的 QQ 版本中也支持使用 TCP 协议了。

端口

一般来说，一台计算机具有单个物理连接到网络。数据通过这个连接去往特定的计算机。然而，该数据可以被用于在计算机上运行的不同应用。那么，计算机知道哪个应用程序转发数据？通过使用端口。

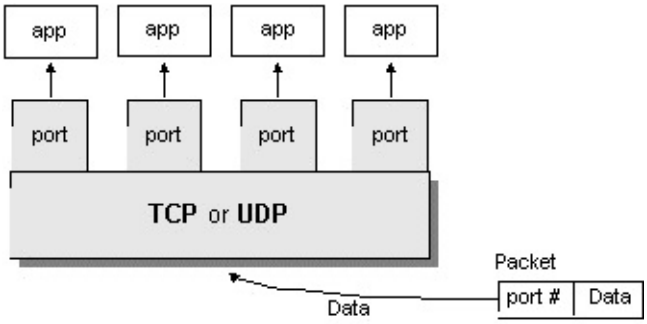
在互联网上传输的数据是通过计算机的标识和端口来定位的。计算机的标识是 32-bit 的 IP 地址。端口由一个 16-bit 的数字。

在诸如面向连接的通信如 TCP，服务器应用将套接字绑定到一个特定端口号。这是向系统注册服务用来接受该端口的数据。然后，客户端可以在与服务器在服务器端口会合，如下图所示：



TCP 和 UDP 协议使用的端口来将接收到的数据映射到一个计算机上运行的进程。

在基于数据报的通信，如 UDP，数据报包中包含它的目的地的端口号，然后 UDP 将数据包路由到相应的应用程序，如本图所示的端口号：



端口号取值范围是从 0 到 65535 （16-bit 长度），其中范围从 0 到 1023 是受限的，它们是被知名的服务所保留使用，例如 HTTP （端口是 80）和 FTP （端口是20、21）等系统服务。这些端口被称为众所周知的端口（well-known ports）。您的应用程序不应该试图绑定到他们。你可以访问 http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml 来查询各种常用的已经分配的端口号列表。

Socket

Socket

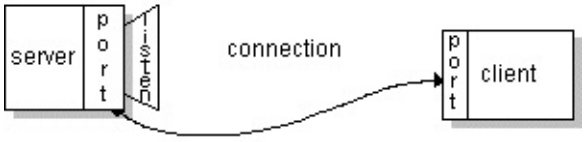
什么是 Socket

Socket（套接字）：是在网络上运行两个程序之间的双向通信链路的一个端点。socket绑定到一个端口号，使得 TCP 层可以标识数据最终要被发送到哪个应用程序。

正常情况下，一台服务器在特定计算机上运行，并具有被绑定到特定端口号的 socket。服务器只是等待，并监听用于客户发起的连接请求的 socket 。

在客户端：客户端知道服务器所运行的主机名称以及服务器正在侦听的端口号。建立连接请求时，客户端尝试与主机服务器和端口会合。客户端也需要在连接中将自己绑定到本地端口以便于给服务器做识别。本地端口号通常是由系统分配的。

如果一切顺利的话，服务器接受连接。一旦接受，服务器获取绑定到相同的本地端口的新 socket ，并且还具有其远程端点设定为客户端的地址和端口。它需要一个新的socket，以便它可以继续监听原来用于客户端连接请求的 socket 。



在客户端，如果连接被接受，则成功地创建一个套接字和客户端可以使用该 socket 与服务器进行通信。

客户机和服务器现在可以通过 socket 写入或读取来交互了。

端点是IP地址和端口号的组合。每个 TCP 连接可以通过它的两个端点被唯一标识。这样，你的主机和服务器之间可以有多个连接。

java.net 包中提供了一个类 Socket，实现您的 Java 程序和网络上的其他程序之间的双向连接。 Socket 类隐藏任何特定系统的细节。通过使用 java.net.Socket 类，而不是依赖于原生代码，Java 程序可以用独立于平台的方式与网络进行通信。

此外，java.net 包含了 ServerSocket 类，它实现了服务器的 socket 可以侦监听和接受客户端的连接。下文将展示如何使用 Socket 和 ServerSocket 类。

实现一个 echo 服务器

让我们来看看这个例子，程序可以建立使用 Socket 类连接到服务器程序，客户端可以通过 socket 向服务器发送数据和接收数据。

EchoClient 示例程序实现了一个客户端，连接到回声服务器。回声服务器从它的客户端接收数据并原样返回回来。EchoServer 实现了 echo 服务器。 （客户端可以连接到支持 Echo 协议的任何主机）

EchoClient 创建一个 socket，从而得到回声服务器的连接。它从标准输入流中读取用户输入，然后通过 socket 转发该文本给回声服务器。服务器通过该 socket 将文本原样输入回给客户端。客户机程序读取并显示从服务器传递回给它的数据。

注意，EchoClient 例子既从 socket 写入又从 socket 中读取数据。

EchoClient 代码：

```
1.  public class EchoClient {
2.      public static void main(String[] args) throws IOException {
3.
4.          if (args.length != 2) {
5.              System.err.println(
6.                  "Usage: java EchoClient <host name> <port number>");
7.              System.exit(1);
8.          }
9.
10.         String hostName = args[0];
11.         int portNumber = Integer.parseInt(args[1]);
12.
13.         try (
14.             Socket echoSocket = new Socket(hostName, portNumber);
15.             PrintWriter out =
16.                 new PrintWriter(echoSocket.getOutputStream(), true);
17.             BufferedReader in =
18.                 new BufferedReader(
19.                     new InputStreamReader(echoSocket.getInputStream()));
20.             BufferedReader stdIn =
21.                 new BufferedReader(
```

```
                    new InputStreamReader(System.in))
        ) {
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
            }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        }
    }
}
```

EchoServer 代码:

```
public class EchoServer {
    public static void main(String[] args) throws IOException {

        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try (
            ServerSocket serverSocket =
                new ServerSocket(Integer.parseInt(args[0]));
            Socket clientSocket = serverSocket.accept();
            PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        ) {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
```

```
28.         }
29.     }
30. }
```

首先启动服务器，在命令行输入如下，设定一个端口号，比如 7（Echo 协议指定端口是 7）：

```
1. java EchoServer 7
```

而后启动客户端，echoserver.example.com 是你主机的名称，如果是本机的话，主机名称可以是 localhost

```
1. java EchoClient echoserver.example.com 7
```

输出效果如下：

```
1. 你好吗？
2. echo: 你好吗？
3. 我很好哦
4. echo: 我很好哦
5. 要过年了，waylau.com 祝你 猴年大吉，身体健康哦！
6. echo: 要过年了，waylau.com 祝你 猴年大吉，身体健康哦！
```

BIO

BIO

NIO

NIO

AIO

AIO

分布式对象 Java RMI

概述

概述

在基于对象的分布式系统中，对象的概念在分布式实现中起着极其关键的作用。从原理上来讲，所有的一切都被作为对象抽象出来，而客户端将以调用对象的方式来获得服务和资源。

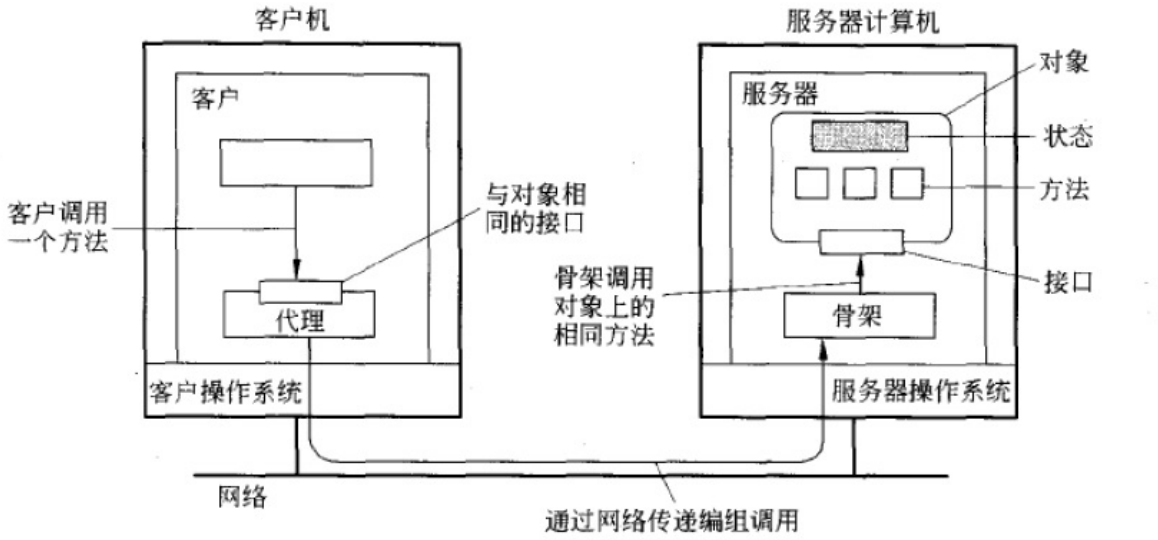
分布式对象之所以成为重要的范型，是因为它相对比较容易的把分布的特性隐藏在对象接口后面。此外，因为对象实际上可以是任何事务，所以它也是构建系统的强大范型。

面向对象技术于20世纪80年代开始用于开发分布式系统。同样，在达到高度分布式透明性的同时，通过远程服务器宿主独立对象的理念构成了开发新一代分布式系统的稳固的基础。在本节中，我们将看到基于对象的分布式系统的常用体系结构。

分布式对象

软件世界中的对象和现实世界中的对象类似，对象存储状态在字段（field）里，而通过方法（methods）暴露其行为。方法对对象的内部状态进行操作，并作为对象与对象之间通信主要机制。隐藏对象内部状态，通过方法进行所有的交互，这个面向对象编程的一个基本原则—数据封装（data encapsulation）。可以通过接口（interface）来使用方法。一个对象可能实现多个接口，而给定的一个接口定义，可能有多个对象为其提供实现。

把接口与实现这些接口的对象进行分隔开，对于分布式系统是至关重要的。严格的隔离允许我们把接口放在一台机器上，而使对象本身驻留在另外一台机器上。这种组织通常称为分布式对象（distributed object）。



（注：该图片引用自《分布式系统原理与范式》一书）

当客户绑定（bind）到一个分布式对象时，就会把这个对象的接口的实现—称为代理（proxy）—加载近客户的地址空间中。代理类似于 RPC 系统中的客户存根（client stub）。它所做的事，是把方法调用编组进消息中，以及对应答消息进行解组，把方法调用的结果返回给客户。实际的对象驻留在服务器计算机上，它在这里提供了与它在客户机上提供的相同的接口。进入的调用请求首先被传递给服务器存根，服务器存根对它们进行解码，在服务器中的对象接口上进行方法的调用。服务器存根还负责对应答进行编码，并把应答消息转发给客户端代理。

服务器端存根通常被称为骨架（skeleton），因为它提供了明确的方式，允许服务器中间件来访问用户定义的对象。实际上，它通常以特定于语言的类的形式包含不完整的代码，需要开发人员来对一步对其进行特殊化处理。

大多数分布式对象的一个特性是它们的状态不是分布式的。状态驻留在单台机器上。在其他机器上，智能地使用被对象实现的接口。这样的对象也被称为远程对象（remote object）。分布式对象的状态本身可能物理地分布在多台机器上，但是这种分布对于对象接口背后的客户来说是透明的。

Java RMI

目前，常用的分布式对象常用技术有微软 DCOM（COM+）、CORBA 以及 Java RMI 等。

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。

架构

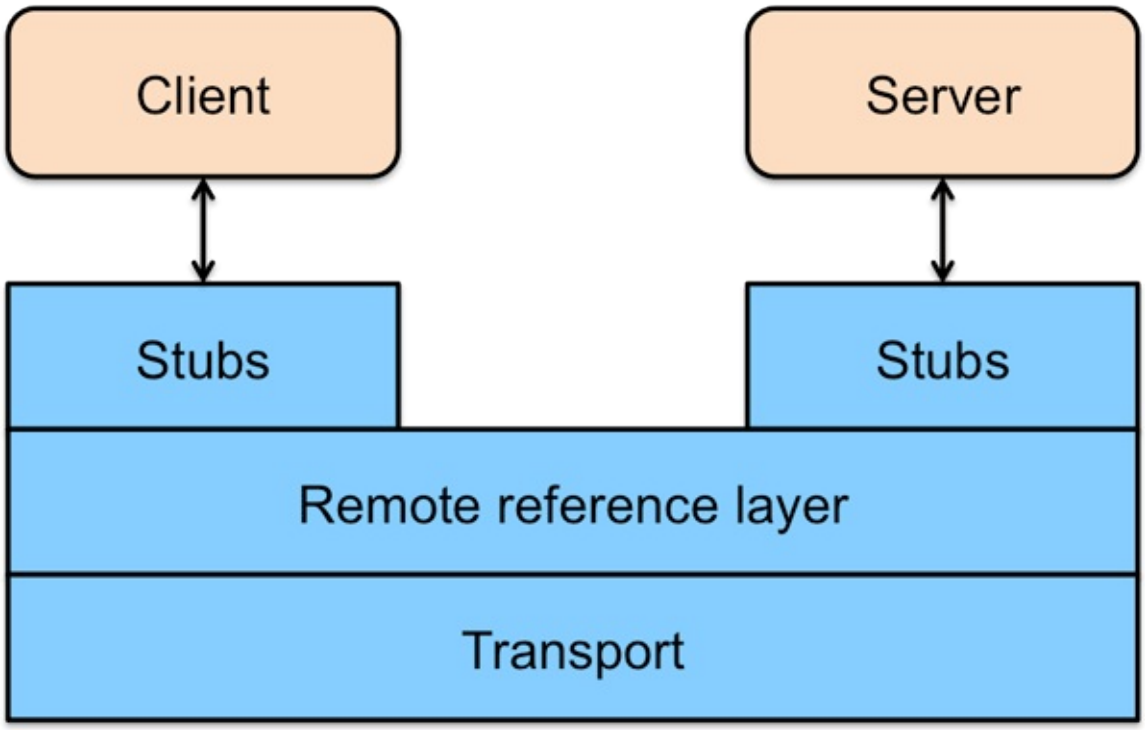
架构

RMI 架构

RMI 是一个三层架构（见图）。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



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。

常用接口

常用接口

接口和类

所有的远程接口都继承自 `java.rmi.Remote` 接口。例如：

```
1. public interface bankaccount extends Remote
2. {
3.     public void deposit(float amount)
4.         throws java.rmi.RemoteException;
5.
6.     public void withdraw(float amount)
7.         throws OverdrawnException,
8.         java.rmi.RemoteException;
9. }
```

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.

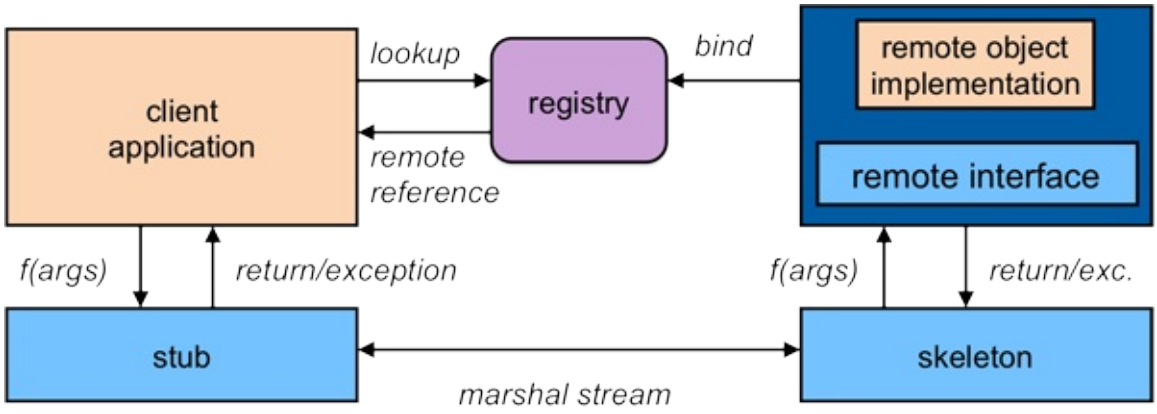
存根

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，

```
1. BankAccount acct = new BankAcctImpl();
2. String url = "rmi://java.sun.com/account";
3. // bind url to remote object
4. java.rmi.Naming.bind(url, acct);
5.
6. // look up account
7. acct = (BankAccount)java.rmi.Naming.lookup(url);
```



示例

- 示例

示例

基于消息的通信

概述

概述

面向消息的中间件（MOM）或者消息队列（MQ），提供了以松散耦合的灵活方式集成应用程序的一种机制。它们提供了基于存储和转发的应用程序之间的异步数据发送，即应用程序彼此不直接通信，而是与作为中介的 MOM 通信。MOM 提供了有保证的消息发送，应用程序开发人员无需了解远程过程调用（PRC）和网络/通信协议的细节。

消息中间件简介

远程过程调用有助于隐藏分布式系统中的通信细节，也就是说增强了访问透明性。但这种机制并不一定适合所有场景。特别是当无法保证发出请求时接收端一定正在执行的情况下，就必须有其他的通信服务。同时 RPC 的同步特性也会造成客户在发出请求得到处理之前被阻塞了，因而有时也需要采取其他的办法。而面向消息的通信就是解决了上面提到的种种问题。

面向消息的通信一般是由消息队列系统（Message-Queuing System，MQ）或者面向消息中间件(Message-Oriented Middleware，MOM)来提供高效可靠的消息传递机制进行平台无关的数据交流，并可基于数据通信进行分布系统的集成。通过提供消息传递和消息排队模型，可在分布环境下扩展进程间的通信，并支持多种通讽协议、语言、应用程序、硬件和软件平台。

通过使用 MQ 或者 MOM，通信双方的程序（称其为消息客户程序）可以在不同的时间运行，程序不在网络上直接通话，而是间接地将消息放入 MQ 或者 MOM 服务器的消息机制中。因为程序间没有直接的联系，所以它们不必同时运行：消息放入适当的队列时，目标程序不需要正在运行；即使目标程序在运行，也不意味着要立即处理该消息。

消息客户程序之间通过将消息放入消息队列或从消息队列中取出消息来进行通讯。客户程序不直接与其他程序通信，避免了网络通讯的复杂性。消息队列和网络通信的维护工作由 MQ 或者 MOM 完成。

常见的 MQ 或者 MOM 产品有 Java Message Service、Apache ActiveMQ、RocketMQ、RabbitMQ、Apache Kafka 等。

消息中间件优点

消息中间件作为一个中间层软件，它为分布式系统中创建、发送、接收消息提供了一套可靠通用的方法，实现了分布式系统中可靠的、高效的、实时的跨平台数据传输。消息中间件减少了开发跨平台和

网络协议软件的复杂性，它屏蔽了不同操作系统和网络协议的具体细节，面对规模和复杂度都越来越高的分布式系统，消息中间件技术显示出了它的优越性：

1. 采用异步通信模式：发送消息者可以在发送消息后进行其它的工作，不用等待接收者的回应，而接收者也不必在接到消息后立即对发送者的请求进行处理；
2. 客户和服务对象生命周期的松耦合关系：客户进程和服务对象进程不要求都正常运行，如果由于服务对象崩溃或者网络故障导致客户的请求不可达，客户不会接收到异常，消息中间件能保证消息不会丢失。

JMS

JMS

Java Message Service（JMS）API是一个 Java 面向消息中间件的 API，用于两个或者多个客户端之间发送消息。

JMS 的目标包括：

- 包含实现复杂企业应用所需要的功能特性；
- 定义了企业消息概念和功能的一组通用集合；
- 最小化这些 Java 程序员必须学习以使用企业消息产品的概念集合；
- 最大化消息应用的可移植性。

JMS 支持企业消息产品提供两种主要的消息风格：

- 点对点（Point-to-Point，PTP）消息风格：允许一个客户端通过一个叫“队列（queue）”的中间抽象发送一个消息给另一个客户端。发送消息的客户端将一个消息发送到指定的队列中，接收消息的客户端从这个队列中抽取消息。
- 发布订阅（Publish/Subscribe，Pub/Sub）消息风：则允许一个客户端通过一个叫“主题（topic）”的中间抽象发送一个消息给多个客户端。发送消息的客户端将一个消息发布到指定的主题中，然后这个消息将被投递到所有订阅了这个主题的客户端。

JMS API

由于历史的原因，JMS 提供四组用于发送和接收消息的接口。

- JMS1.0 定义了两个特定领域相关的API，一个用于点对点的消息处理（queue），一个用于发布订阅的消息处理（topic）。尽管由于向后兼容的理由这些接口一直被保留在 JMS 中，但是在以后的 API 中应该考虑被废弃掉。
- JMS1.1 引入了一个新的统一的一组 API，可以同时用于点对点和发布订阅消息模式。这也被称作标准（standard） API。
- JMS2.0引入了一组简化 API，它拥有标准 API 的全部特性，同时接口更少、使用更方便。

以上每组 API 提供一组不同的接口集合，用于连接到 JMS 提供者、发送和接收消息。因此，它们共享一组代表消息、消息目的地和其他各方面功能特性的通用接口。

下面是使用标准 API 来发送信息的例子：

```
@Resource(lookup = "jms/connectionFactory ")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public void sendMessageOld (String payload) throws JMSException{
    try (Connection connection = connectionFactory.createConnection()) {
        Session session = connection.createSession();
        MessageProducer messageProducer =
        session.createProducer(inboundQueue);
        TextMessage textMessage =
        session.createTextMessage(payload);
        messageProducer.send(textMessage);
    }
}
```

下面是使用简化 API 来发送信息的例子：

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public void sendMessageNew (String payload) {
    try (MessagingContext context = connectionFactory.createMessagingContext();){
        context.send(inboundQueue,payload);
    }
}
```

所有的接口都在 `javax.jms` 包下。

如果读者想了解更多有关 JMS 的规范，可以在线查阅 https://java.net/projects/jms-spec/pages/Home 。

ActiveMQ 示例

ActiveMQ 示例

其他

其他

Web Services

概述

概述

JAX-WS

JAX-WS

RESTful 概述

RESTful 概述

基于 **Jersey** 的 **JAX-RS** 示例

基于 Jersey 的 JAX-RS 示例

微服务

概述

概述

Spring Boot

Spring Boot

监控

概述

概述

JMX

JMX

ZooKeeper

ZooKeeper

参考文献

参考文献