Broadview
www.broadview.com.cn

分布式系统

常用技术及案例分析

柳伟卫 编著

中国工信出版集团
电子工业出版社

内 容 简 介

本书全面介绍在设计分布式系统时所要考虑的技术方案，内容丰富、案例新颖，相关理论与技术实践较为前瞻。本书不仅仅介绍了分布式系统的原理、基础理论，同时还引入了大量市面上常用的最新分布式系统技术，不仅告诉读者怎么用，同时也分析了为什么这么用，并阐述了这些技术的优缺点。希望本书可以成为读者案头的工具书，供读者随手翻阅。

本书分为三大部分，即分布式系统基础理论、分布式系统常用技术以及经典的分布式系统案例分析。第一部分主要介绍分布式系统基础理论知识，总结一些在设计分布式系统时需要考虑的范式、知识点以及可能会面临的问题，其中包括线程、通信、一致性、容错性、CAP 理论、安全性和并发等相关内容；同时讲述分布式系统的常见架构体系，其中也包括最近比较火的 RESTful 风格架构、微服务、容器技术等。第二部分主要列举了在分布式系统应用中经常用到的一些主流技术，并介绍这些技术的作用和用法；这些技术涵盖了分布式消息服务、分布式计算、分布式存储、分布式监控系统、分布式版本控制、RESTful、微服务、容器等领域的内容。第三部分选取了以淘宝网和 Twitter 为代表的国内外知名互联网企业的大型分布式系统案例，分析其架构设计以及演变过程；这部分相当于是对第二部分零散的技术点做一个“串烧”，让读者可以结合技术的理论，看到实战的效果。

本书主要面向的读者是对分布式系统感兴趣的计算机专业的学生、软件工程师、系统架构师等。

图书在版编目（CIP）数据

分布式系统常用技术及案例分析 / 柳伟卫编著. —北京：电子工业出版社，2017.2
ISBN 978-7-121-30771-3

Ⅰ. ①分… Ⅱ. ①柳… Ⅲ. ①分布式操作系统－研究 Ⅳ. ①TP316.4

中国版本图书馆 CIP 数据核字（2016）第 323368 号

责任编辑：陈晓猛
印　　刷：北京京科印刷有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：43.75　　字数：980 千字
版　　次：2017 年 2 月第 1 版
印　　次：2017 年 2 月第 1 次印刷
印　　数：3000 册　　定价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

前　言

写作背景

我一直想写一本关于分布式系统方面的书。一方面是想把个人多年工作中涉及的分布式技术做一下总结，另一方面也想把个人的经验分享给广大的读者朋友。由于我的开发工作大都以Java 为主，所以一开始的主题设想是“分布式 Java”，书也以开源方式发布在互联网上（网址为https://github.com/waylau/distributed-java）。

后来，陈晓猛编辑看到了这本开源书，以及我关于分布式系统方面的博文，问我是否有兴趣出版分布式相关题材的图书。当然书的内容不仅仅是“分布式 Java”。

对于出书一事，我犹豫良久。首先，本身工作挺忙，实在无暇顾及其他；其次，虽然我之前写过超过一打的书籍（可见 https://waylau.com/books/），但多是开源电子书，时间、内容方面自然也就不会有太多约束，几乎是“想写就写，没有时间就不写”，这个跟正式出版还是存在比较大的差异的；最后，这本书涉及面相对较广，需要查阅大量资料，实在是太耗费精力。

但陈晓猛编辑还是鼓励我能够去尝试做这个事情。思索再三，于是我便答应。当然，最后这本书还是在规定时间内完成了。它几乎耗尽了我写作期间所有的业余和休息时间。

“不积跬步，无以至千里；不积小流，无以成江海。”虽然整本书从构思到编写完成的时间不足一年，但书中的大部分知识点，却是我在多年的学习、工作中积累下来的。之所以能够实现快速写作，一方面是做了比较严格的时间管理，另一方面也得益于我多年坚持写博客和开源书的习惯。

内容介绍

本书分为三大部分，即分布式系统基础理论、分布式系统常用技术以及经典的分布式系统案例分析。第一部分为第 1 章和第 2 章，主要介绍分布式系统基础理论知识，总结一些在设计分布式系统时需要考虑的范式、知识点以及可能会面临的问题。 第二部分为第 3 章到第 8 章，主要列举了在分布式系统应用中经常用到的一些主流技术，并介绍这些技术的作用和用法。第

三部分为第 9 章和第 10 章，选取了以淘宝网和 Twitter 为代表的国内外知名互联网企业的大型分布式系统案例，分析其架构设计以及演变过程。

第 1 章介绍分布式系统基础理论知识，总结一些在设计分布式系统时需要考虑的范式、知识点以及可能会面临的问题，其中包括线程、通信、一致性、容错性、CAP 理论、安全性和并发等相关内容。

第 2 章详细介绍分布式系统的架构体系，包括传统的基于对象的体系结构、SOA，也包括最近比较火的 RESTful 风格架构、微服务、容器技术、Serverless 架构等。

第 3 章介绍常用的分布式消息服务框架，包括 Apache ActiveMQ、RabbitMQ、RocketMQ、Apache Kafka 等。

第 4 章介绍分布式计算理论和应用框架方面的内容，包括 MapReduce、Apache Hadoop、Apache Spark、Apache Mesos 等。

第 5 章介绍分布式存储理论和应用框架方面的内容，包括 Bigtable、Apache HBase、Apache Cassandra、Memcached、Redis、MongoDB 等。

第 6 章介绍分布式监控方面常用的技术，包括 Nagios、Zabbix、Consul、ZooKeeper 等。

第 7 章介绍常用的分布式版本控制工具，包括 Bazaar、Mercurial、Git 等。

第 8 章介绍 RESTful API、微服务及容器相关的技术，着重介绍 Jersey、Spring Boot、Docker 等技术的应用。

第 9 章和第 10 章分别介绍以淘宝网和 Twitter 为代表的国内外知名互联网企业的大型分布式系统案例，分析其架构设计以及演变过程。

源代码

本书提供源代码下载，下载地址为 https://github.com/waylau/distributed-systems-technologies-and-cases-analysis。

勘误和交流

本书如有勘误，会在 https://github.com/waylau/distributed-systems-technologies-and-cases-analysis 上进行发布。由于笔者能力有限，时间仓促，难免错漏，欢迎读者批评指正。读者也可以到博文视点官网的本书页面进行交流（www.broadview.com.cn/30771）。

您也可以直接联系我：

博客：https://waylau.com

邮箱：waylau521@gmail.com

微博：http://weibo.com/waylau521

开源：https://github.com/waylau

致谢

首先，感谢电子工业出版社博文视点公司的陈晓猛编辑，是您鼓励我将本书付诸成册，并在我写作过程中审阅了大量稿件，给予了我很多指导和帮助。感谢工作在幕后的电子工业出版社评审团队对于本书在校对、排版、审核、封面设计、错误改进方面所给予的帮助，使本书得以顺利出版发行。

其次，感谢在我十几年求学生涯中教育过我的所有老师。是你们将知识和学习方法传递给了我。感谢我曾经工作过的公司和单位，感谢和我一起共事过的同事和战友，你们的优秀一直是我追逐的目标，你们所给予的压力正是我不断改进自己的动力。

感谢我的父母、妻子 Funny 和两个女儿。由于撰写本书，牺牲了很多陪伴家人的时间。感谢你们对于我工作的理解和支持。

最后，特别要感谢这个时代，互联网让所有人可以公平地享受这个时代的成果。感谢那些为计算机、互联网所做出贡献的先驱，是你们让我可以站在更高的“肩膀”上！感谢那些为本书提供灵感的佳作，包括《分布式系统原理与范式》《Unix Network Programming》《Enterprise SOA》《MapReduce Design Patterns》《Hadoop: The Definitive Guide》《Learning Hbase》《Advanced Analytics with Spark》《Pro Git》《Docker in Action》《淘宝技术这十年》《Hatching Twitter》，等等，详细的书单可以参阅本书最后的“参考文献”部分。

柳伟卫

2016 年 11 月 13 日于杭州

第 1 章

分布式系统基础知识

1.1 概述

自从有了计算机，人类的生活就发生了巨大的变化，原来人力需要数年才能解决的计算问题，计算机“分分钟”就搞定了。回顾历史，在 19 世纪 50 年代，计算机在诞生初期，主要用于军事方面，而天才的数学家图灵建造了计算机来破译德军的“英格玛（Enigma）”密码系统。那时候的计算机如一个庞大笨重的巨人，足足占据了好几间房子的空间。在 19 世纪 80 年代，计算机的体积越来越小，开始被广泛应用于科研计算，但那时候计算机的价格仍然是极其昂贵的，只有少数几个科研单位或者大学才能买得起。今天，个人计算机或者以手机为代表的智能设备已经走进寻常百姓家了。每个人几乎都拥有手机，手机不仅仅是通信工具，还能发语音、看视频、玩游戏，让人与人之间的联系变得更加紧密。智能手环随时监控你的身体状况，并根据你每天的运动量、身体指标来给你提供合理的饮食运动建议。出门逛街甚至不需要带钱包了，吃饭购物搭车时使用手机就可以支付费用，多么方便快捷。智能家居系统更是你生活上的“管家”，什么时候该睡觉了，智能家居系统就自动拉上窗帘，关灯；早上起床了，智能家居系统会自动拉开窗帘，并播放动人的音乐，让你可以愉快地享受新的一天的来临；你再也不用担心家里的安全情况，智能家居系统会帮你监控一切，有异常情况时会及时发送通知到你的手机，让你第一时间掌握家里的状况。

毫无疑问，计算机改变了人类的工作和生活方式，而计算机系统也正在进行一场变革。没错，任何一个手机应用，或者智能 App，都离不开背后那个神秘的巨人——分布式系统。正是那些看不见的分布式系统，每天处理着数以亿计的计算，提供可靠而稳定的服务。

本章就揭开分布式系统的神秘面纱。

1.1.1 什么是分布式系统

《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单个相关系统”。

这里面包含了 2 个含义：

- 硬件独立；
- 软件统一。

什么是硬件独立？所谓硬件独立，是指计算机机器本身是独立的。一个大型的分布式系统，会由若干台计算机来组成系统的基础设施。而软件统一，是指对于用户来说，用户就像是跟单个系统打交道。就好比我们每天上网看视频，视频网站对我们来说就是一个系统软件，它们背后是如何运作的，部署了几台服务器，每台服务器是干什么的，这些对用户来说是透明的，不

可见的。用户不关心背后的这些服务器，用户所关心的是，今天访问的这个网站能提供什么样的节目，视频运行是否流畅，卡不卡帧、清晰度如何等。

而软件统一的另外一个方面是指，分布式系统的扩展和升级都比较容易。分布式系统中的某些节点发生故障，不会影响整体系统的可用性。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，也不会感知到新加入的部分。

1.1.2　集中式系统 VS.分布式系统

集中式系统主要部署在 HP、IBM、SUN 等小型机以上档次的服务器中，把所有的功能都集成到主服务器上，这样对服务器的要求很高，性能也极其苛刻。它们的主要特色在于年宕机时间只有几小时，所以又统称为 z 系列（zero，零）。AS/400 主要应用在银行和制造业，还用于 Domino，主要的技术在于 TIMI（技术独立机器界面）、单级存储，有了 TIMI 技术可以做到硬件与软件相互独立。RS/6000 比较常见，用于科学计算和事务处理等。这类主机的单机性能一般都很强，带多个终端。终端没有数据处理能力，运算全部在主机上进行。现在的银行系统大部分都是这种集中式的系统，此外，在大型企业、科研单位、军队、政府等也有应用。

集中式系统主要流行于 20 世纪。现在还在使用集中式系统的，很大一部分原因是为了沿用原来的软件，而这些软件往往很昂贵。优点是便于维护，操作简单。但这样的系统也有缺陷，就是不出问题还好，一出问题，就会造成单点故障，所有功能就都不能正常工作了。由于集中式的系统相关的技术只被少数厂商所掌握，个人要对这些系统进行扩展和升级往往也比较麻烦，一般的企业级应用很少会用到集中式系统。图 1-1 是一个典型的集中式系统的示意图。

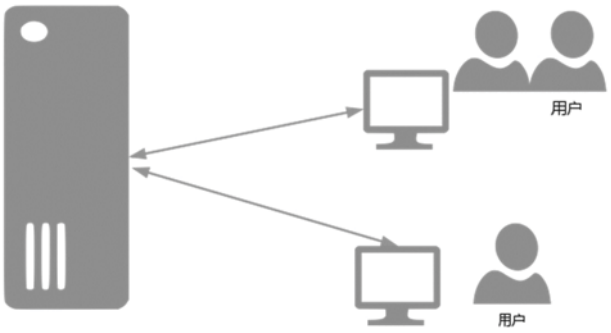

图 1-1　集中式系统示意图

而分布式系统恰恰相反。分布式系统是通过中间软件来对现有计算机的硬件能力和相应的软件功能进行重新配置和整合。它是一种多处理器的计算机系统，各处理器通过互连网络构成统一的系统。系统采用分布式计算结构，即把原来系统内中央处理器处理的任务分散给相应的处理器，实现不同功能的各个处理器相互协调，共享系统的外设与软件。这样就加快了系统的

处理速度，简化了主机的逻辑结构。它甚至不需要很高的配置，一些低配置“退休”下来的机器也能被重新纳入分布式系统中使用，这样无疑就降低了成本，而且还易于维护。同时，分布式系统往往由多个主机组成，任何一台主机宕机都不影响整体系统的使用，所以分布式系统的可用性往往比较高。图 1-2 是一个典型的分布式系统的示意图。

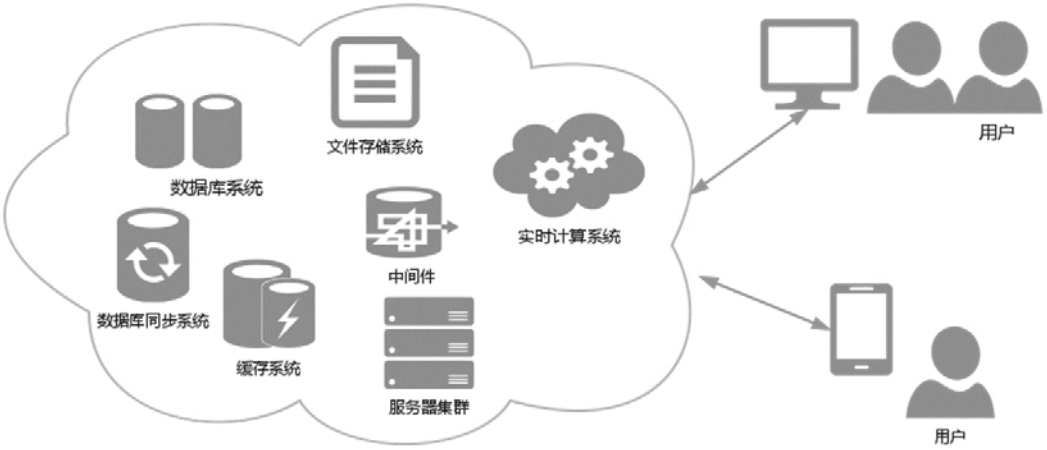

图 1-2　分布式系统示意图

正是由于分布式系统的这些优点，使得分布式系统的应用越来越广泛，也代表了未来应用的发展趋势。

1.1.3　如何设计分布式系统

设计分布式系统的本质就是“如何合理地将一个系统拆分成多个子系统并部署到不同机器上”。所以首要考虑的问题是如何合理地将系统进行拆分。由于拆分后的各个子系统不可能孤立地存在，必然要通过网络进行连接交互，所以它们之间如何通信变得尤为重要。当然在通信过程中要识别“敌我”，防止信息在传递过程中被拦截和篡改，这就涉及安全问题了。分布式系统要适应不断增长的业务需求，就需要考虑其扩展性。分布式系统还必须要保证可靠性和数据的一致性。

概括起来，在设计分布式系统时，应考虑以下几个问题：

- 如何将系统拆分为子系统？
- 如何规划子系统间的通信？
- 如何考虑通信过程中的安全？
- 如何让子系统可以扩展？

- 如何保证子系统的可靠性？
- 如何实现数据的一致性？

本书的大部分内容就是针对分布式系统中常见的问题进行探讨。

1.1.4　分布式系统所面临的挑战

分布式系统是难于理解、设计、构建和管理的，它们将比单个机器数倍还要多的变量引入到设计中，使应用程序的根源问题更难发现。SLA（Service－Level Agreement，服务水平协议）是衡量停机和/或性能下降的标准，大多数现代应用程序有一个期望的弹性 SLA 水平，通常按“9”的数量增加（如每月 99.9%或 99.99%可用性）。每个额外的 9 变得越来越难实现。

设计分布式系统时，经常需要考虑如下的挑战。

- **异构性**：分布式系统由于基于不同的网络、操作系统、计算机硬件和编程语言来构造，必须要考虑一种通用的网络通信协议来屏蔽异构系统之间的差异。一般交由中间件来处理这些差异。
- **缺乏全球时钟**：在程序需要协作时，通过交换消息来协调它们的动作。紧密的协调经常依赖于对程序动作发生时间的共识。但是，实际网络上计算机同步时钟的准确性受到极大的限制，即没有一个正确时间的全局概念。这是通过网络发送消息作为唯一的通信方式这一事实带来的直接结果。
- **一致性**：数据被分散或者复制到不同的机器上，如何保证各台主机之间的数据的一致性将成为一个难点。
- **故障的独立性**：任何计算机都有可能故障，且各种故障不尽相同。它们之间出现故障的时机也是相互独立的。一般分布式系统要设计成允许出现部分故障而不影响整个系统的正常使用。
- **并发**：分布式系统的目的是为了更好地共享资源。那么系统中的每个资源都必须被设计成在并发环境中是安全的。
- **透明性**：分布式系统中任何组件的故障，或者主机的升级、迁移，对于用户来说都是透明的，不可见的。
- **开放性**：分布式系统由不同的程序员来编写不同的组件，组件最终要集成为一个系统，那么组件所发布的接口必须遵守一定的规范且能够被互相理解。
- **安全性**：加密用于给共享资源提供适当的保护，在网络上所有传递的敏感信息都需要进行加密。拒绝服务攻击仍然是一个有待解决的问题。
- **可扩展性**：系统要设计成随着业务量的增加，相应的系统也必须能扩展来提供对应的服务。

1.2 线程

在早期的计算机操作系统中，能拥有资源和独立运行的基本单位是进程。然而随着计算机技术的发展，进程出现了很多弊端，一是由于进程是资源拥有者，创建、撤消与切换存在较大的时空开销，因此需要引入轻量型进程；二是由于对称多处理机（Symmetric Multi-Processor，SMP）出现，可以满足多个运行单位，而多个进程并行开销过大。

因此在上世纪 80 年代，出现了能独立运行的基本单位——线程（Thread）。

1.2.1 什么是线程

线程是程序执行流的最小单元。一个标准的线程由线程 ID、当前指令指针（PC）、寄存器集合和堆栈组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位。线程自己不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其他线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。

线程拥有三种基本状态：

- 就绪；
- 阻塞；
- 运行。

线程的状态图如图 1-3 所示。

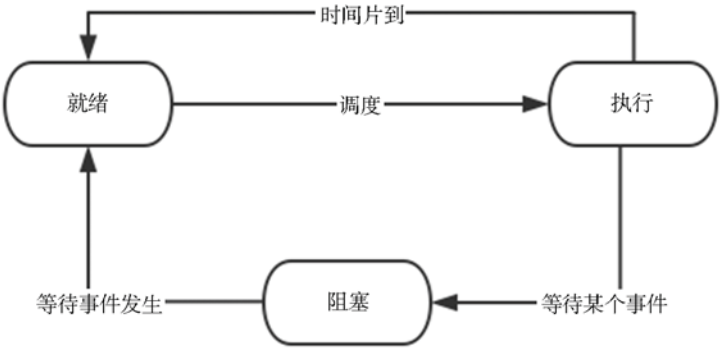

图 1-3 线程的状态图

就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；运行状态是指线程占有处理机正在运行；阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执

行。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。

线程是程序中一个单一的顺序控制流程，是进程内一个相对独立的、可调度的执行单元。在单个程序中同时运行多个线程完成不同的工作，称为多线程。多数情况下，多线程能提升程序的性能。

1.2.2　进程和线程

进程和线程是并发编程的两个基本的执行单元。在大多数编程语言中，并发编程主要涉及线程。

一个计算机系统通常有许多活动的进程和线程。在给定的时间内，每个处理器中只能有一个线程得到真正的运行。对于单核处理器来说，处理时间是通过时间切片来在进程和线程之间进行共享的。

进程有一个独立的执行环境。进程通常有一个完整的、私人的基本运行时资源。特别是每个进程都有自己的内存空间。操作系统的进程表（Process table）存储了 CPU 寄存器值、内存映像、打开的文件、统计信息、特权信息等。进程一般定义为执行中的程序，也就是当前操作系统的某个虚拟处理器上运行的一个程序。多个进程并发共享同一个 CPU 以及其他硬件资源是透明的，操作系统支持进程之间的隔离。这种并发透明性需要付出相对较高的代价。

进程往往被视为等同于程序或应用程序。然而，用户看到的一个单独的应用程序可能实际上是一组合作的进程。大多数操作系统都支持进程间通信（Inter Process Communication，IPC），如管道和 socket。IPC 不仅用于同个系统的进程之间的通信，也可以用在不同系统的进程之间进行通信。

线程，有时被称为轻量级进程（Lightweight Process，LWP）。进程和线程都提供了一个执行环境，但创建一个新的线程比创建一个新的进程需要更少的资源。线程系统一般只维护用来让多个线程共享 CPU 所必须的最少量信息。特别是线程上下文（Thread Context）中一般只包含 CPU 上下文以及某些其他线程管理信息。通常忽略那些对于多线程管理不是完全必要的信息。这样单个进程中防止数据遭到某些线程不合法的访问的任务就完全落在了应用程序开发人员的肩上。线程不像进程那样彼此隔离以及受到操作系统的自动保护，所以在多线程程序开发过程中需要开发人员做更多的努力。

线程中存在于进程中，每个进程都至少一个线程。线程共享进程的资源，包括内存和打开的文件。这使得工作变得高效，但也存在了一个潜在的问题——通信。关于通信的内容，会在后面章节中讲述。

现在多核处理器或多进程的计算机系统越来越流行。这大大增强了系统的进程和线程的并发执行能力。但即便是没有多处理器或多进程的系统中，并发仍然是可能的。关于并发的内容，

会在后面章节中讲述。

1.2.3 编程语言中的线程对象

在面向对象语言开发过程中，每个线程都与 Thread 类的一个实例相关联。由于 Java 语言的流行，下文中的例子将用 Java 来实现和使用线程对象，以作为并发应用程序的基本原型。

1. 定义和启动一个线程

Java 中有两种创建 Thread 实例的方式：

- 提供 Runnable 对象。Runnable 接口定义了一个方法 run，用来包含线程要执行的代码。HelloRunnable 示例如下所示。

```
public class HelloRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Hello from a runnable!");
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

- 继承 Thread。Thread 类本身是实现 Runnable，虽然它的 run 方法什么都没干。HelloThread 示例如下所示。

```
public class HelloThread extends Thread {
    @Override
    public void run() {
        System.out.println("Hello from a thread!");
    }
    /**
     * @param args
     */
    public static void main(String[] args) {
        (new HelloThread()).start();
```

```
        }
    }
```

请注意，这两个例子调用 start 来启动线程。

第一种方式，它使用 Runnable 对象，在实际应用中更普遍，因为 Runnable 对象可以继承 Thread 以外的类。第二种方式，在简单的应用程序中更容易使用，但受限于你的任务类必须是一个 Thread 的后代。本书推荐使用第一种方法，将 Runnable 任务从 Thread 对象分离来执行任务。这样不仅更灵活，而且它适用于高级线程管理 API。

Thread 类还定义了大量的方法用于线程管理。

2. 使用 sleep 来暂停执行

Thread.sleep 可以让当前线程执行暂停一个时间段，这样处理器的时间就可以给其他线程使用。

sleep 有两种重载形式：一个是指定睡眠时间为毫秒级，另外一个是指定睡眠时间为纳秒级。然而，这些睡眠时间不能保证是精确的，因为它们是由操作系统提供的，并受其限制，因而不能假设 sleep 的睡眠时间是精确的。此外，睡眠周期也可以通过中断来终止，我们将在后面的章节中看到。

SleepMessages 示例——使用 sleep 每隔 4 秒打印一次消息：

```
public class SleepMessages {

    /**
     * @param args
     */
    public static void main(String[] args) throws InterruptedException
    {
        String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too" };

        for (int i = 0; i < importantInfo.length; i++) {

            // 暂停 4 秒
            Thread.sleep(4000);
```

```
            // 打印消息
            System.out.println(importantInfo[i]);
        }
    }
}
```

请注意 main 声明抛出 InterruptedException。当 sleep 是激活的时候，若有另一个线程中断当前线程时，则 sleep 抛出异常。由于该应用程序还没有定义另一个线程来引起中断，所以考虑捕捉 InterruptedException。

3. 中断（interrupt）

中断是表明一个线程应该停止它正在做和将要做的事。线程通过在 Thread 对象调用 interrupt 来实现线程的中断。为了中断机制能正常工作，被中断的线程必须支持自己的中断。

支持中断

如何实现线程支持自己的中断？这要看是它目前正在做什么。如果线程调用方法频繁抛出 InterruptedException 异常，那么它只要在 run 方法捕获了异常之后返回即可。例如：

```
for (int i = 0; i < importantInfo.length; i++) {

    // 暂停 4 秒
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {

        // 已经中断，无须更多信息
        return;
    }

    // 打印消息
    System.out.println(importantInfo[i]);
}
```

很多方法都会抛出 InterruptedException，如 sleep，被设计成在收到中断时立即取消它们当前的操作并返回。

若线程长时间没有调用方法抛出 InterruptedException 的话，那么它必须定期调用 Thread.interrupted，该方法在接收到中断后将返回 true。

```
for (int i = 0; i < inputs.length; i++) {
```

```
        heavyCrunch(inputs[i]);

        if (Thread.interrupted()) {

            // 已经中断，无须更多信息
            return;
        }
    }
```

在这个简单的例子中，代码简单地测试该中断，如果已接收到中断线程就退出。在更复杂的应用程序中，它可能会更有意义地抛出一个 InterruptedException：

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

中断状态标志

中断机制是使用被称为中断状态的内部标志实现的。调用 Thread.interrupt 可以设置该标志。当一个线程通过调用静态方法 Thread.interrupted 来检查中断时，中断状态被清除。非静态 isInterrupted 方法用于线程来查询另一个线程的中断状态，而不会改变中断状态标志。

按照惯例，任何方法因抛出一个 InterruptedException 而退出都会清除中断状态。当然，它可能因为另一个线程调用 interrupt 而让那个中断状态立即被重新设置回来。

4. join 方法

join 方法允许一个线程等待另一个线程完成。假设 t 是一个正在执行的 Thread 对象，那么

```
t.join();
```

会导致当前线程暂停执行直到 t 线程终止。join 允许程序员指定一个等待周期。与 sleep 一样，等待时间是依赖于操作系统的时间，同时不能假设 join 等待时间是精确的。

像 sleep 一样，join 并通过 InterruptedException 退出来响应中断。

1.2.4　SimpleThreads 示例

SimpleThreads 示例有两个线程。第一个线程是每个 Java 应用程序都有的主线程。主线程创建 Runnable 对象的 MessageLoop，并等待它完成。如果 MessageLoop 需要很长时间才能完成，主线程就中断它。

该 MessageLoop 线程打印出一系列消息。如果中断之前就已经打印了所有消息，则 MessageLoop

线程打印一条消息并退出。

```
public class SimpleThreads {

    // 显示当前执行线程的名称、信息
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
                          threadName,
                          message);
    }

    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < importantInfo.length; i++) {

                    // 暂停 4 秒
                    Thread.sleep(4000);

                    // 打印消息
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }

    public static void main(String args[])
        throws InterruptedException {
```

```
// 在中断 MessageLoop 线程（默认为 1 小时）前先延迟一段时间（单位是毫秒）
long patience = 1000 * 60 * 60;

// 如果命令行参数出现
// 设置 present 的时间值
// 单位是秒
if (args.length > 0) {
    try {
        patience = Long.parseLong(args[0]) * 1000;
    } catch (NumberFormatException e) {
        System.err.println("Argument must be an integer.");
        System.exit(1);
    }
}

threadMessage("Starting MessageLoop thread");
long startTime = System.currentTimeMillis();
Thread t = new Thread(new MessageLoop());
t.start();

threadMessage("Waiting for MessageLoop thread to finish");

// 循环直到 MessageLoop 线程退出
while (t.isAlive()) {
    threadMessage("Still waiting...");

    // 最长等待 1 秒
    // 给 MessageLoop 线程来完成

    t.join(1000);
    if (((System.currentTimeMillis() - startTime) > patience)
          && t.isAlive()) {
        threadMessage("Tired of waiting!");
        t.interrupt();

        // 等待
        t.join();
    }
```

```
            }
            threadMessage("Finally!");
        }
    }
```

若对 Java 的 Thread 类感兴趣，可以查看其在线 API：https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html。

上面例子的代码可以在 https://github.com/waylau/distributed-systems-technologies-and-cases-analysis 的 distributed-systems-java-demos 程序的 com.waylau.essentialjava.thread 包下找到。

1.3 通信

进程间的通信是一切分布式系统的核心。如果没有通信机制，分布式系统的各个子系统将是“一盘散沙”，毫无作用。

本节将介绍网络的基础知识，以及常用的通信方式。

1.3.1 网络基础知识

1. OSI 参考模型

OSI 参考模型，即开放式通信系统互联参考模型（Open Systems Interconnection Reference Model），是国际标准化组织（ISO）提出的一个试图使各种计算机在世界范围内互连为网络的标准框架。OSI 模型把网络通信的工作分为 7 层，分别是物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。表 1-1 描述了各个层次的关系。

表 1-1 OSI 各层次关系表

层　　次	数据格式	功能与连接方式	典型设备
应用层（Application）	数据（Data）	网络服务与使用者应用程序间的一个接口	终端设备（PC、手机、平板等）
表示层（Presentation）	数据（Data）	数据表示、数据安全、数据压缩	终端设备（PC、手机、平板等）
会话层（Session）	数据（Data）	会话层连接到传输层的映射；会话连接的流量控制；数据传输；会话连接恢复与释放；会话连接管理、差错控制	终端设备（PC、手机、平板等）

续表

层　　次	数据格式	功能与连接方式	典型设备
传输层（Transport）	数据组织成数据段（Segment）	用一个寻址机制来标识一个特定的应用程序（端口号）	终端设备（PC、手机、平板等）
网络层（Network）	分割和重新组合数据包（Packet）	基于网络层地址（IP 地址）进行不同网络系统间的路径选择	路由器
数据链路层（Data Link）	将比特信息封装成数据帧（Frame）	物理层上建立、撤销、标识逻辑链接和链路复用以及差错校验等功能。通过使用接收系统的硬件地址或物理地址来寻址	网桥、交换机
物理层（Physical）	传输比特（bit）流	建立、维护和取消物理连接	光纤、同轴电缆、双绞线、网卡、中继器

OSI 分层的优点：

- 分层清晰、协议规范，易于理解和学习；
- 层间的标准接口方便了工程模块化；
- 创建了一个更好的互连环境；
- 降低了复杂度，使程序更容易修改，产品开发的速度更快；
- 每层利用紧邻的下层服务，更容易记住每个层的功能。

OSI 是一个定义良好的协议规范集，并有许多可选部分来完成类似的任务。它定义了开放系统的层次结构、层次之间的相互关系以及各层所包括的可能的任务。

OSI 参考模型并没有提供一个可以实现的方法，而是描述了一些概念，用来协调进程间通信标准的制定。即 OSI 参考模型并不是一个标准，而是一个在制定标准时所使用的概念性框架。

2. TCP/IP 网络模型

TCP/IP 模型实际上是 OSI 模型的一个浓缩版本，它只有四个层次：

（1）应用层（Application），对应 OSI 的应用层、表示层、会话层；

（2）传输层（Transport），对应 OSI 的传输层；

（3）网络层（Network），对应 OSI 的网络层；

（4）链路层（Link），对应 OSI 的数据链路层和物理层。

TCP/IP 模型图如图 1-4 所示。

图 1-4　TCP/IP 模型图

每一层负责不同的功能。

- **链路层**：有时也称作数据链路层或网络接口层，通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆（或其他任何传输媒介）的物理接口细节。
- **网络层**：有时也称作互联网层，处理分组在网络中的活动，例如分组的选路。在 TCP/IP 协议簇中，网络层协议包括 IP 协议（互联网协议）、ICMP 协议（互联网控制报文协议），以及 IGMP 协议（互联网组管理协议）。
- **传输层**：主要为两台主机上的应用程序提供端到端的通信。在 TCP/IP 协议簇中，有两个互不相同的传输协议：TCP（传输控制协议）和 UDP（用户数据报协议）。TCP 为两台主机提供高可靠性的数据通信。它所做的工作包括把应用程序交给它的数据分成合适的小块交给下面的网络层，确认接收到的分组，设置发送最后确认分组的超时时钟等。由于传输层提供了高可靠性的端到端的通信，因此应用层可以忽略所有这些细节。而另一方面，UDP 则为应用层提供一种非常简单的服务。它只是把称作数据报的分组从一台主机发送到另一台主机，但并不保证该数据报能到达另一端。任何必需的可靠性必须由应用层来提供。这两种运输层协议在不同的应用程序中有不同的用途，这一点将在后面讲解。
- **应用层**：负责处理特定的应用程序细节，常用的通用应用有 Telnet（远程登录）、FTP（文件传输协议）、SMTP（简单邮件传送协议）、SNMP（简单网络管理协议）。

OSI 参考模型与 TCP/IP 协议模型的对比。

- **相同点**：都有应用层、传输层、网络层；都是下层服务上层。
- **不同点**：层数不同；模型与协议出现的次序不同，TCP/IP 先有协议，后有模型（出现早），OSI 先有模型，后有协议（出现晚）。

3. 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”就是指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，等待对方摘机应答，然后才说明是谁。

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，简单地讲就是：

（1）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。

那么，TCP 如何保证数据的可靠性？总结来说，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。如果有必要，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存空间。TCP 的接收端只允许另一端发送接收端缓存区所能接纳的数据。这将防止较快主机致使较慢主机的缓存区溢出。

4. 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，这是没有开始和结束的），只是一次性的传递，事先不需要接收方的响应，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、即时聊天工具等。如果你的应用对于可靠性的要求不高，而又希望有较高的传输效率，那么可以选择 UDP。

5. 端口

一般来说，一台计算机具有单个物理连接到网络的能力。数据通过这个连接去往特定的计算机。然而，该数据可以被用在计算机上运行的不同应用中。那么，计算机如何知道使用哪个应用程序转发数据呢？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 的数字组成。

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。这时向系统注册服务用来接收该端口的数据。然后，客户端可以在与服务器在服务器端口会合，如图 1-5 所示。

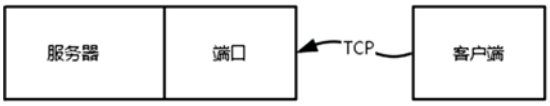

图 1-5　TCP 端口

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

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，如图 1-6 所示。

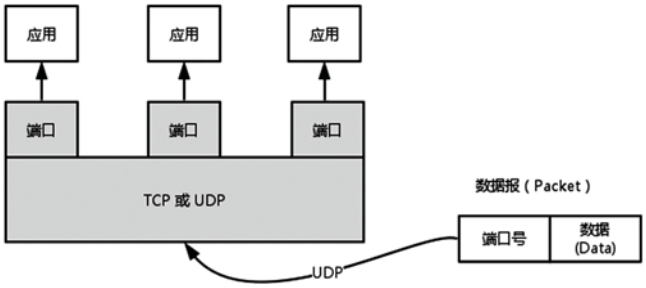

图 1-6　UDP 端口

端口号的取值范围是从 0 到 65535（16-bit 长度），其中从 0 到 1023 的范围是受限的，它们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）。应用程序不应该试图绑定到它们。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被分配的端口号列表。

1.3.2　网络 I/O 模型的演进

1. 同步和异步

同步和异步描述的是用户线程与内核的交互方式：

- **同步**是指用户线程发起 I/O 请求后需要等待或者轮询内核 I/O 操作完成后才能继续执行；
- **异步**是指用户线程发起 I/O 请求后仍继续执行，当内核 I/O 操作完成后会通知用户线程，或者调用用户线程注册的回调函数。

2. 阻塞和非阻塞

阻塞和非阻塞描述的是用户线程调用内核 I/O 操作的方式：

- **阻塞**是指 I/O 操作需要彻底完成后才返回到用户空间；
- **非阻塞**是指 I/O 操作被调用后立即返回给用户一个状态值，无须等到 I/O 操作彻底完成。

一个 I/O 操作其实分成了两个步骤：发起 I/O 请求和实际的 I/O 操作。

阻塞 I/O 和非阻塞 I/O 的区别在于第一步，发起 I/O 请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞 I/O，如果不阻塞，那么就是非阻塞 I/O。

同步 I/O 和异步 I/O 的区别就在于第二个步骤是否阻塞，如果实际的 I/O 读写阻塞请求进程，那么就是同步 I/O。

3. UNIX I/O 模型

UNIX 下共有五种 I/O 模型：

- 阻塞 I/O；
- 非阻塞 I/O；
- I/O 复用（select 和 poll）；
- 信号驱动 I/O（SIGIO）；
- 异步 I/O（Posix.1 的 aio_系列函数）。

注：若读者想深入了解 UNIX 的网络知识，推荐阅读 W.Richard Stevens 的《UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI》，本节只简单介绍了这五种模型，文中的图例也引用自该书的图例。

阻塞 I/O 模型

请求无法立即完成则保持阻塞。

- **阶段 1**：等待数据就绪。网络 I/O 的情况就是等待远端数据陆续抵达；磁盘 I/O 的情况就是等待磁盘数据从磁盘上读取到内核态内存中。
- **阶段 2**：数据复制。出于系统安全，用户态的程序没有权限直接读取内核态内存，因此内核负责把内核态内存中的数据复制一份到用户态内存中。

阻塞 I/O 模型如图 1-7 所示。

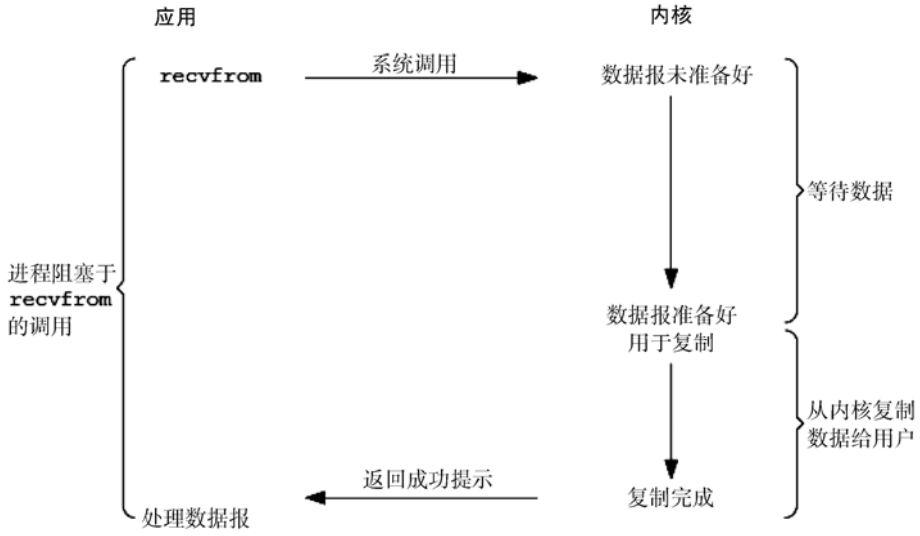

图 1-7　阻塞 I/O 模型

本节中将 recvfrom 函数视为系统调用。一般 recvfrom 实现都有一个从应用程序进程中运行到内核中运行的切换，一段事件后再跟一个返回到应用进程的切换。

图 1-7 中，进程阻塞的整段时间是指从调用 recvfrom 开始到它返回的这段时间，当进程返回成功指示时，应用进程开始处理数据报。

非阻塞 I/O 模型

- socket 设置为 NONBLOCK（非阻塞）就是告诉内核，当所请求的 I/O 操作无法完成时，不要将进程睡眠，而是立刻返回一个错误码（EWOULDBLOCK），这样请求就不会阻塞；

- I/O 操作函数将不断地测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。整个 I/O 请求的过程中，虽然用户线程每次发起 I/O 请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，这是对 CPU 时间的极大浪费。
- 数据准备好了，从内核复制到用户空间。

非阻塞 I/O 模型如图 1-8 所示。

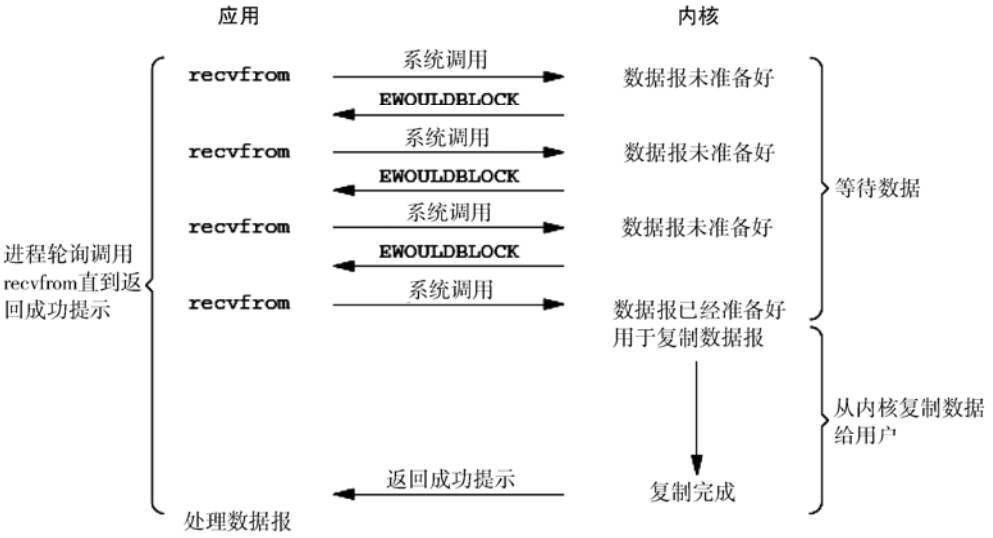

图 1-8　非阻塞 I/O 模型

一般很少直接使用这种模型，而是在其他 I/O 模型中使用非阻塞 I/O 这一特性。这种方式对单个 I/O 请求的意义不大，但给 I/O 复用铺平了道路。

I/O 复用模型

I/O 复用会用到 select 或者 poll 函数，在这两个系统调用中的某一个上阻塞，而不是阻塞于真正的 I/O 系统调用。函数也会使进程阻塞，但是和阻塞 I/O 所不同的是，这两个函数可以同时阻塞多个 I/O 操作。而且可以同时对多个读操作、多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。

I/O 复用模型如图 1-9 所示。

从流程上来看，使用 select 函数进行 I/O 请求和同步阻塞模型没有太大的区别，甚至还多了添加监视 socket，以及调用 select 函数的额外操作，效率更差。但是，使用 select 最大的优势是用户可以在一个线程内同时处理多个 socket 的 I/O 请求。用户可以注册多个 socket，然后不断地调用 select 来读取被激活的 socket，即可达到在同一个线程内同时处理多个 I/O 请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

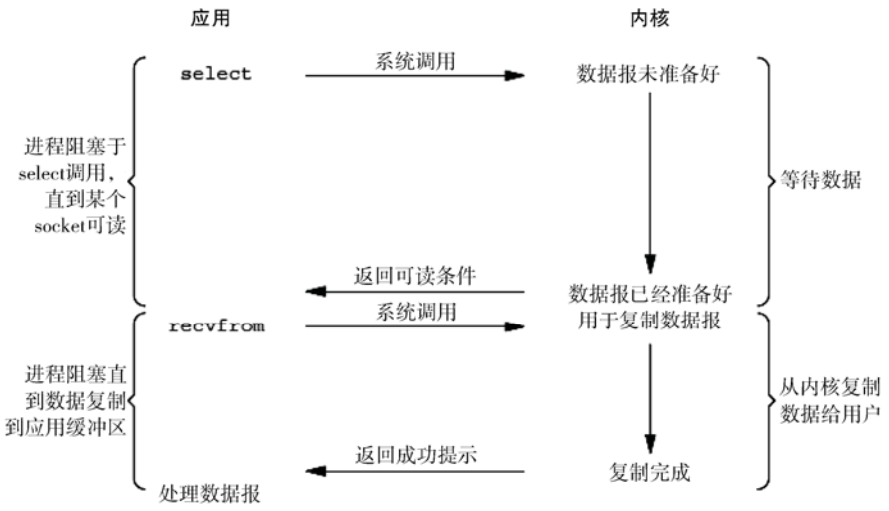

图 1-9　I/O 复用模型

I/O 复用模型使用了 Reactor 设计模式实现了这一机制。

调用 select/poll 该方法由一个用户态线程负责轮询多个 socket，直到某个阶段 1 的数据就绪，再通知实际的用户线程执行阶段 2 的复制操作。通过一个专职的用户态线程执行非阻塞 I/O 轮询，模拟实现了阶段 1 的异步化。

信号驱动 I/O（SIGIO）模型

首先，我们允许 socket 进行信号驱动 I/O，并通过调用 sigaction 来安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 recvfrom 来读取数据报，并通知主循环数据已准备好被处理，也可以通知主循环，让它来读取数据报。

信号驱动 I/O（SIGIO）模型如图 1-10 所示。

该模型的好处是，当等待数据报到达时，可以不阻塞。主循环可以继续执行，只是等待信号处理程序的通知；或者数据已准备好被处理，或者数据报已准备好被读。

异步 I/O 模型

异步 I/O 是 POSIX 规范定义的。通常，这些函数会通知内核来启动操作并在整个操作（包括从内核复制数据到我们的缓存中）完成时通知我们。

该模式与信号驱动 I/O（SIGIO）模型的不同点在于，驱动 I/O（SIGIO）模型告诉我们 I/O 操作何时可以启动，而异步 I/O 模型告诉我们 I/O 操作何时完成。

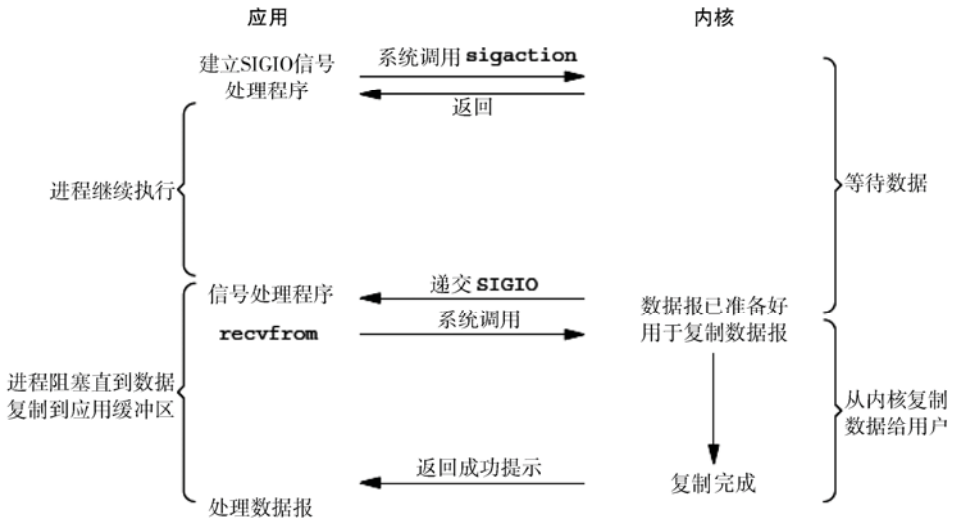

图 1-10　信号驱动 I/O（SIGIO）模型

调用 aio_read 函数，告诉内核传递描述字、缓存区指针、缓存区大小、文件偏移，然后立即返回，我们的进程不阻塞于等待 I/O 操作的完成。当内核将数据复制到缓存区后，才会生成一个信号，来通知应用程序。

异步 I/O 模型如图 1-11 所示。

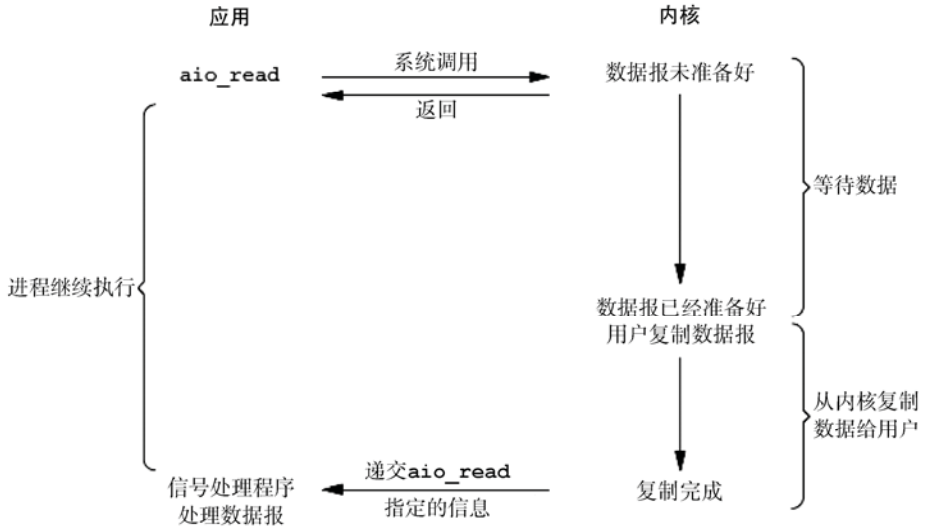

图 1-11　异步 I/O 模型

异步 I/O 模型使用了 Proactor 设计模式实现了这一机制。[1]

异步 I/O 模型告知内核：当整个过程（包括阶段 1 和阶段 2）全部完成时，通知应用程序来读数据。

几种 I/O 模型的比较

前四种模型的区别是阶段 1 不相同，阶段 2 基本相同，都是将数据从内核复制到调用者的缓存区。而异步 I/O 的两个阶段都不同于前四个模型。几种 I/O 模型的比较如图 1-12 所示。

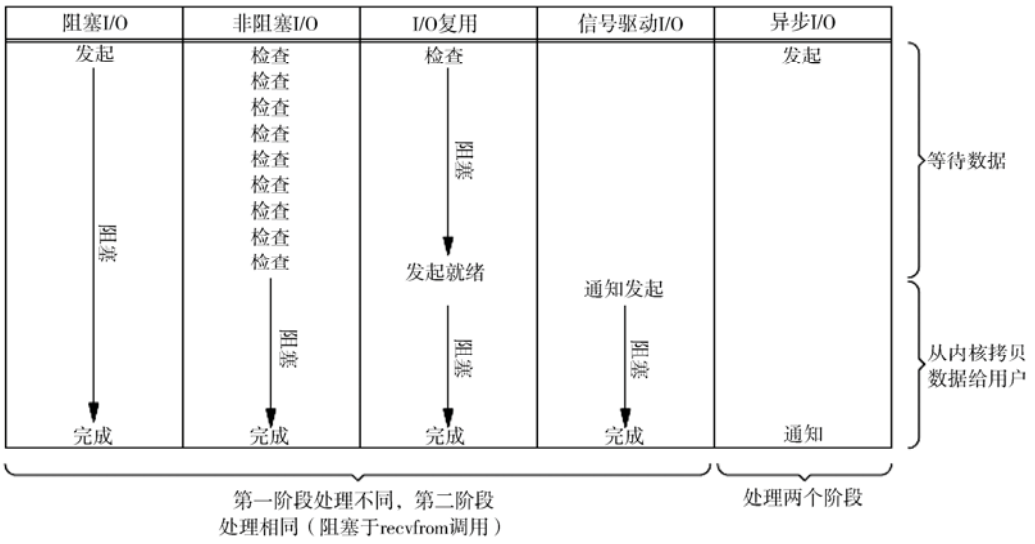

图 1-12　几种 I/O 模型的比较

同步 I/O 操作引起请求进程阻塞，直到 I/O 操作完成。异步 I/O 操作不引起请求进程阻塞。上面前四个模型——阻塞 I/O 模型、非阻塞 I/O 模型、I/O 复用模型和信号驱动 I/O 模型都是同步 I/O 模型，而异步 I/O 模型才是真正的异步 I/O。

4. 常见 Java I/O 模型

在了解了 UNIX 的 I/O 模型之后，就能明白其实 Java 的 I/O 模型也是类似的。

“阻塞 I/O”模式

下面的 EchoServer 是一个简单的阻塞 I/O 例子，服务器启动后，等待客户端连接。在客户端连接服务器后，服务器就阻塞读写数据流。

---

[1] 有关“Proactor 设计模式”可以参阅 https://en.wikipedia.org/wiki/Proactor_pattern。

EchoServer 代码：

```
public class EchoServer {
    public static int DEFAULT_PORT = 7;

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

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }

        try (
            ServerSocket serverSocket =
                new ServerSocket(port);
            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 "+ port + " or listening for a connection");
              System.out.println(e.getMessage());
        }
    }
}
```

改进为“阻塞 I/O+多线程”模式

使用多线程来支持多个客户端访问服务器。

主线程 MultiThreadEchoServer.java：

```
public class MultiThreadEchoServer {
    public static int DEFAULT_PORT = 7;

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

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        Socket clientSocket = null;
        try (ServerSocket serverSocket = new ServerSocket(port);) {
            while (true) {
                clientSocket = serverSocket.accept();

                // 多线程
                new Thread(new EchoServerHandler(clientSocket)).start();
            }
        } catch (IOException e) {
            System.out.println(
                   "Exception caught when trying to listen
                      on port " + port + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

处理器类 EchoServerHandler.java：

```
public class EchoServerHandler implements Runnable {
    private Socket clientSocket;

    public EchoServerHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run() {
        try (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(e.getMessage());
        }
    }
}
```

存在问题：每次接收到新的连接都要新建一个线程，处理完后销毁线程，代价大。当有大量地短连接出现时，性能比较低。

改进为“阻塞 I/O+线程池”模式

针对上面多线程的模型中出现的线程重复创建、销毁带来的开销，可以采用线程池来优化。每次接收到新连接后从池中取一个空闲线程进行处理，处理完后再放回池中，重用线程避免了频繁地创建和销毁线程带来的开销。

主线程 ThreadPoolEchoServer.java：

```
public class ThreadPoolEchoServer {
    public static int DEFAULT_PORT = 7;

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

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        ExecutorService threadPool = Executors.newFixedThreadPool(5);
        Socket clientSocket = null;
        try (ServerSocket serverSocket = new ServerSocket(port);) {
            while (true) {
```

```
                clientSocket = serverSocket.accept();

                // 线程池
                threadPool.submit(new Thread(new EchoServerHandler(clientSocket)));
            }
        } catch (IOException e) {
            System.out.println(
                   "Exception caught when trying to listen
                      on port " + port + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

存在问题：在大量短连接的场景中性能会有提升，因为不用每次都创建和销毁线程，而是重用连接池中的线程。但在大量长连接的场景中，因为线程被连接长期占用，不需要频繁地创建和销毁线程，因而没有什么优势。

虽然这种方法可以适用于小到中度规模的客户端的并发数，如果连接数超过 100000 或更多，那么性能将很不理想。

改进为“非阻塞 I/O”模式

“阻塞 I/O+线程池”网络模型虽然比“阻塞 I/O+多线程”网络模型在性能方面有所提升，但这两种模型都存在一个共同的问题：读和写操作都是同步阻塞的，面对大并发（持续大量连接同时请求）的场景，需要消耗大量的线程来维持连接。CPU 在大量的线程之间频繁切换，性能损耗很大。一旦单机的连接超过 1 万，甚至达到几万的时候，服务器的性能会急剧下降。

而 NIO 的 Selector 却很好地解决了这个问题，用主线程（一个线程或者是 CPU 个数的线程）保持住所有的连接，管理和读取客户端连接的数据，将读取的数据交给后面的线程池处理，线程池处理完业务逻辑后，将结果交给主线程发送响应给客户端，少量的线程就可以处理大量连接的请求。

Java NIO 由以下几个核心部分组成：

- Channel；
- Buffer；
- Selector。

要使用 Selector，得向 Selector 注册 Channel，然后调用它的 select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例

子有新连接进来、数据接收等。

主线程 NonBlokingEchoServer.java：

```
public class NonBlokingEchoServer {
    public static int DEFAULT_PORT = 7;

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

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        System.out.println("Listening for connections on port " + port);

        ServerSocketChannel serverChannel;
        Selector selector;
        try {
            serverChannel = ServerSocketChannel.open();
            InetSocketAddress address = new InetSocketAddress(port);
            serverChannel.bind(address);
            serverChannel.configureBlocking(false);
            selector = Selector.open();
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        } catch (IOException ex) {
            ex.printStackTrace();
            return;
        }

        while (true) {
            try {
                selector.select();
            } catch (IOException ex) {
                ex.printStackTrace();
                break;
            }
```

```
                Set<SelectionKey> readyKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = readyKeys.iterator();
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    iterator.remove();
                    try {
                        if (key.isAcceptable()) {
                            ServerSocketChannel  server  =  (ServerSocketChannel)
key.channel();
                            SocketChannel client = server.accept();
                            System.out.println("Accepted connection from " + client);
                            client.configureBlocking(false);
                            SelectionKey clientKey = client.register(selector,
                                   SelectionKey.OP_WRITE | SelectionKey.OP_READ);
                            ByteBuffer buffer = ByteBuffer.allocate(100);
                            clientKey.attach(buffer);
                        }
                        if (key.isReadable()) {
                            SocketChannel client = (SocketChannel) key.channel();
                            ByteBuffer output = (ByteBuffer) key.attachment();
                            client.read(output);
                        }
                        if (key.isWritable()) {
                            SocketChannel client = (SocketChannel) key.channel();
                            ByteBuffer output = (ByteBuffer) key.attachment();
                            output.flip();
                            client.write(output);

                            output.compact();
                        }
                    } catch (IOException ex) {
                        key.cancel();
                        try {
                            key.channel().close();
                        } catch (IOException cex) {
                        }
                    }
                }
```

```
            }

        }
    }
```

改进为“异步 I/O”模式

Java SE 7 版本之后，引入了对异步 I/O（NIO.2）的支持，为构建高性能的网络应用提供了一个利器。

主线程 AsyncEchoServer.java：

```
public class AsyncEchoServer {

    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {
        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }

        ExecutorService taskExecutor = Executors.newCachedThreadPool
Executors.defaultThreadFactory());

        // 创建异步服务器 socket channel 并绑定到默认组
        try (AsynchronousServerSocketChannel asynchronousServerSocketChannel
= AsynchronousServerSocketChannel.open()) {
            if (asynchronousServerSocketChannel.isOpen()) {

                // 设置一些参数
                asynchronousServerSocketChannel.setOption
(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
                asynchronousServerSocketChannel.setOption
(StandardSocketOptions.SO_REUSEADDR, true);

                // 绑定服务器 socket channel 到本地地址
```

```
                asynchronousServerSocketChannel.bind(new  InetSocketAddress
(port));

                // 显示等待客户端的信息
                System.out.println("Waiting for connections ...");
                while (true) {
                    Future<AsynchronousSocketChannel>
asynchronousSocketChannelFuture = asynchronousServerSocketChannel
                            .accept();
                    try {
                        final AsynchronousSocketChannel asynchronousSocketChannel
= asynchronousSocketChannelFuture
                                .get();
                        Callable<String> worker = new Callable<String>() {
                            @Override
                            public String call() throws Exception {
                                String host = asynchronousSocketChannel
.getRemoteAddress().toString();
                                System.out.println("Incoming connection from: "
+ host);
                                final ByteBuffer buffer = ByteBuffer
.allocateDirect(1024);

                                // 发送数据
                                while (asynchronousSocketChannel.read
(buffer).get() != -1) {
                                    buffer.flip();
                                    asynchronousSocketChannel.write(buffer).get();
                                    if (buffer.hasRemaining()) {
                                        buffer.compact();
                                    } else {
                                        buffer.clear();
                                    }
                                }
                                asynchronousSocketChannel.close();
                                System.out.println(host + " was successfully served!");
                                return host;
                            }
```

```
                        };
                        taskExecutor.submit(worker);
                    } catch (InterruptedException | ExecutionException ex) {
                        System.err.println(ex);
                        System.err.println("\n Server is shutting down ...");

                        // 执行器不再接收新线程
                        // 并完成队列中所有的线程
                        taskExecutor.shutdown();

                        // 等待所有线程完成
                        while (!taskExecutor.isTerminated()) {
                        }
                        break;
                    }
                }
            } else {
                System.out.println("The asynchronous server-socket channel
cannot be opened!");
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

1.3.3　远程过程调用（RPC）

1. 进程间通信（IPC）

进程间通信（Inter-Process Communication，IPC）指至少两个进程或线程间传送数据或信号的一些技术或方法。进程是计算机系统分配资源的最小单位。每个进程都有自己的一部分独立的系统资源，彼此是隔离的。为了能使不同的进程互相访问资源并进行协调工作，才有了进程间通信。这些进程可以运行在同一计算机上或网络连接的不同计算机上。进程间的通信技术包括消息传递、同步、共享内存和远程过程调用。IPC 是一种标准的 UNIX 通信机制。

2. 过程调用的类型

在讨论客户/服务器（C/S）模型和过程调用时，主要有三种不同类型的过程调用。

- 本地过程调用（Local Procedure Call，LPC）：指被调用的过程（函数）与调用过程处于同一个进程中。典型的情况是，调用者通过执行某条机器指令把控制传给新过程，被调用过程保存机器寄存器的值，并在栈顶分配存放其本地变量的空间。
- 同主机间的远程过程调用（Remote Procedure Call，RPC）：指被调用的过程与调用过程处于不同的进程中，但同属于一台主机上。我们通常称调用者为客户，被调用者的过程为服务器。
- 不同主机间的远程过程调用：指一台主机上的某个客户调用另外一台主机上的某个服务器的过程。

3. 什么是远程过程调用

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》[2]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。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。

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，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允许同时运行多个远程过程调用。

图 1-13 描述了数据报在一个简单的 RPC 传递的过程。

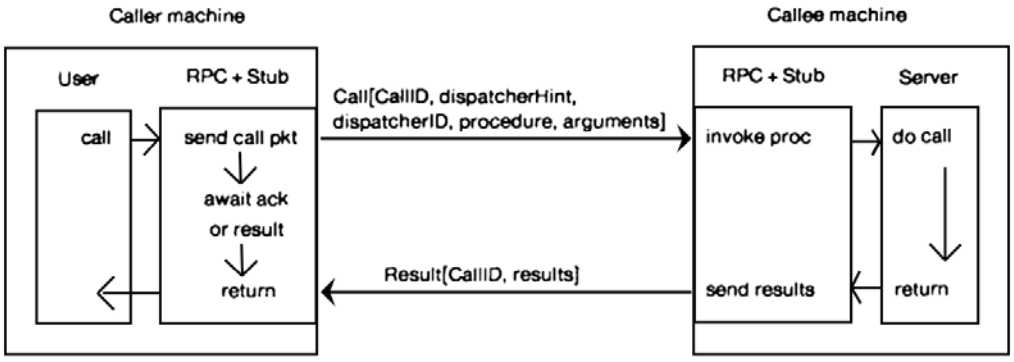

图 1-13　数据报在一个简单的 RPC 传递的过程

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

4. 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 1-14（a）所示。为了进行调用，调用方首先把参数反序压入堆栈，即最后一个参数先压入，如图 1-14（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。

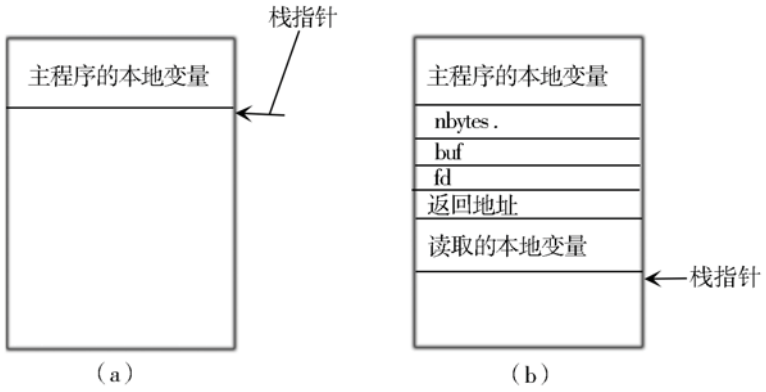

图 1-14　过程调用中的参数传递

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中读取数据，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实现调用的。如图 1-14（b）所示，程序员并不知道 read 干了什么。

RPC 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 1-14（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，而后请求将此消息发送到服务器，如图 1-15 所示。在对 send 的调用后，客户存根调用 receive 过程，随即阻塞自己，直到收到响应消息。

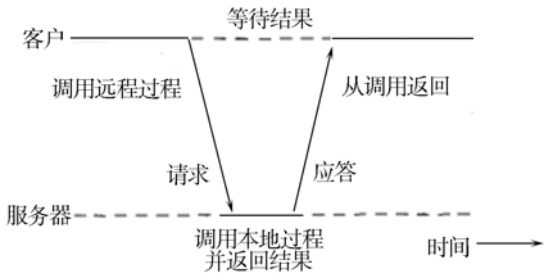

图 1-15　客户与服务器之间的 RPC 原理

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，然后被阻塞，等待消息输入。收到消息后，服务器将参数从消息中提取出来，然后以常规方式调用服务器上的相应过程（见图 1-15）。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存区，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（缓存区）打包成消息，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，它所知道的唯一事情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的，还是远程完成的。

整个方法中，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。

以上步骤就是客户过程将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 参考模型中，RPC 跨越了会话层和表示层。

5. 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。

1）如何传递参数

传递值参数

传递值参数比较简单，图 1-16 是一个简单 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。

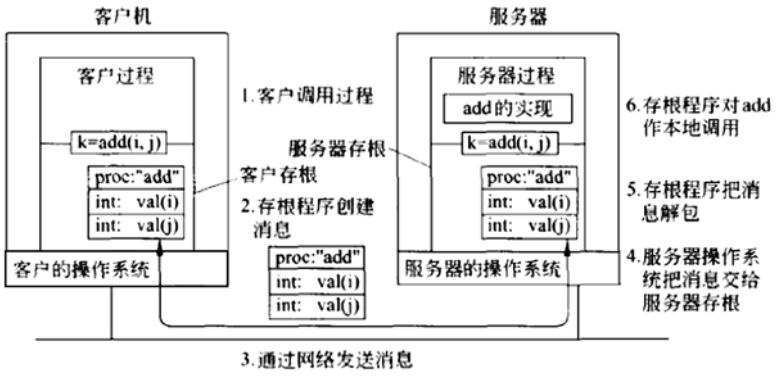

图 1-16　通过 RPC 进行远程计算的步骤

通过 RPC 进行远程计算的步骤如下：

（1）将参数放入消息中，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，就必须发送参数的副本，将它们放置在远程系统内存中，向它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，它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），并传输到远程端来重建数据结构。

2）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和值。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。

3）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，并允许用户选择。

4）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。

5）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的。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崩溃了，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，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。

6）远程调用的性能怎么样

毫无疑问，一个远程过程调用将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。然而，这并不应该阻止我们使用远程过程调用。

7）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，这个过程是可信的吗？
- 客户端发送消息到远程计算机，这个远程机器是可信的吗？
- 服务器如何验证接收的消息来自合法的客户端？服务器如何识别客户端？
- 消息在网络中传播时如何防止被其他进程嗅探？
- 如何防止消息在客户端和服务器的网络传播中被其他进程拦截和修改？
- 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？

6. 远程过程调用的优点

远程过程调用有诸多优点：

- 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，客户端是可以动态选择的。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。

7. 远程过程调用 API

任何 RPC 实现都需要提供一组支持库。

- **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)**: 目前，有一小部分 RPC 包支持转换包括时间格式、货币格式和特定于语言的字符串的功能。
- **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内存。它们也可能需要为创建网络缓存区而分配内存。RPC 包支持对象，RPC 系统需要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。

8. 远程过程调用发展历程

- **第一代 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）上得到支持。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。该阶段的其他代表产品还有 DCE RPC。

- **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，但它们仍然不支持自动、透明的方式的面向对象编程技术。该阶段的主要产品有微软 DCOM（COM+）、CORBA、Java RMI。
- **第三代 RPC 以及 Web Services**：传统 RPC 解决方案可以工作在互联网上，但问题是，它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。该阶段的代表产品有 XML-RPC、SOAP、Microsoft .NET Remoting、JAX-WS 等。

1.3.4　面向消息的通信

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的请求得到处理之前就被阻塞了，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）提供高效可靠的消息传递机制来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，并支持多种通信协议、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信。客户程序不直接与其他程序通信，避免了网络通信的复杂性。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，这些产品在第 3 章再详细讲解。

Java Message Service (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）消息风格——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 提供了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），另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。

1.4 一致性

分布式系统的一个重要问题是数据的复制。对数据进行复制一般是为了增强系统的可靠性和提高性能。举例来说，当一个数据库的副本被破坏以后，那么系统只需要转换到其他数据副本就能继续运行下去。另外一个例子，当访问单一服务器管理的数据的进程数不断增加时，系统就需要对服务器的数量进行扩充，此时，对服务器进行复制，随后让它们分担工作负荷，就可以提高性能。

但复制数据的同时也带来了一个难点，那就是如何保持各个副本数据的一致性。换句话说，

更新其中任意一个副本时，必须确保同时更新其他副本；否则，数据的各个副本将不再相同（数据不一致）。本节就来探讨如何实现数据的一致性。

1.4.1 以数据为中心的一致性模型

一致性模型实质上是进程和数据存储之间的一个约定。正常情况下，在一个数据项上执行读操作时，它期待该操作返回的是该数据在其最后一次写操作之后的结果。在没有全局时钟的情况下，精确地定义哪次写操作是最后一次写操作是十分困难的。于是就产生了一系列用其他方式定义的一致性模型。

1. 严格一致性（strict consistency）

任意读操作都要读到最新的写的结果。严格一致性是限制性最强的模型，依赖于绝对的全局时钟，但是在分布式系统中实现这种模型的代价太大，所以在实际系统中的运用有限，基本上不可能做到。

2. 持续一致性（continuous consistency）

有多种不同的方法来为应用程序指定它们能容忍哪些不一致性，其中有一种通用的方法，它定义了区分不一致性的三个互相独立的坐标轴：副本之间的数值偏差，副本之间新旧程度偏差以及更新操作顺序的偏差。这些偏差形成了持续一致性的范围。

数值偏差可以这样理解：已应用于其他的副本，但还没有应用于给定副本的更新数目。比如，Web 缓存可能还没有得到 Web 服务器执行的一批操作。

新旧程度偏差与副本最近一次的更新有关。对于某些应用，只要副本提供的数据不是很旧，是可以容忍的，比如，天气预报通常会滞后一段时间。

更新操作顺序的偏差是指只要可以界定副本之间的差异，就允许不同的副本采用不同的更新顺序。

3. 顺序一致性（sequential consistency）

任何执行结果都是相同的，就好像所有进程对数据存储的读/写操作是按某种序列顺序执行的一样，并且每个进程的操作按照程序所制定的顺序出现在这个序列中。

也就是说，任何读/写操作的交叉都是可接受的，但是所有进程都看到相同的操作交叉。

4. 因果一致性（casual consistency）

所有进程必须以相同的顺序看到具有潜在因果关系的写操作。不同机器上的进程可以以不同的顺序看到并发的写操作。

假设 P1 和 P2 是有因果关系的两个进程，如果 P2 的写操作信赖于 P1 的写操作，那么 P1 和 P2 对 x 的修改顺序，在 P3 和 P4 看来一定是一样的。但如果 P1 和 P2 没有关系，那么 P1 和 P2 对 x 的修改顺序，在 P3 和 P4 看来可以是不一样的。

相比顺序一致性，因果一致性去掉了那些没有联系的操作需达成一致顺序观点的要求，只是保留了那些必要的顺序（有因果关系的）。

5. 入口一致性（entry consistency）

入口一致性其实也就是对每个共享的数据定义一个同步变量（即：锁）。当然，没有进行同步就进行读操作，是不保证正确的结果的。

1.4.2　以客户为中心的一致性

以客户为中心的一致性也就是从用户视角来看，数据是一致的。客户只关心数据最终是否会一致。

只要保证对于同一个用户，他访问到的数据是一致的就可以了。如果用户只是访问一个副本，这个就很好实现，否则就需要一定的策略了。当没有更多的更新时，要保证当前的更新最终会传播到所有副本上。著名的例子有 DNS 系统、万维网。

最终一致性需要注意一个典型的问题，即当客户访问不同的副本时，问题就出现了。更具体的例子比如，作者在博客上更改了一篇博文内容，在 A 地的用户先访问到最新的内容，而 B 地由于离博客服务器远，看到的还是原先的内容。

对于最终一致性的数据存储而言，这个示例很有代表性。问题是由用户有时可能对不同的副本进行操作的事实引起的。以客户为中心的一致性分为如下几大类。

1. 单调读一致性（monotonic-read consistency）

当进程从一个地方读出数据 x，那么以后再读到的 x 应该是和当前 x 相同或比当前更新的版本。也就是说，如果进程迁移到了别的位置，那么对 x 的更新应该比进程先到达。

以分布式邮件数据库系统为例。每个用户的邮箱可能分布式地复制在多台机器上。邮件可能被插入到任何一个位置的邮箱中。但是，数据更新是以一种懒惰的方式传播的。假设用户在杭州读取到了他的邮件（假定只读取邮件不会影响其他邮箱，也就是说，消息不会被删除，甚至不会被标记为已读），当用户飞到惠州后，再次打开他的邮箱时，单调读一致性可以保证当他在惠州打开他的邮箱时，邮箱中仍然有杭州邮箱里的那些消息。

2. 单调写一致性（monotonic-write consistency）

跟单调读相应，如果一个进程写一个数据 x，那么它在本地或者迁移到别的地方再进行写

操作的时候，原来的写操作必须要先传播到这个位置。也就是说，进程要在任何地方至少和上一次写一样新的数据。

3. 读写一致性（read-your-writes consistency)

读写一致性指一个进程对于数据 x 的写操作，进程无论到任何副本上都应该能被后续读操作看到这个写操作的影响，也就是看到和自己写操作的影响或者更新的值。

也就是说，写操作总是在同一个进程执行的后续读操作之前完成，而不管这个后续读操作发生在什么位置。

4. 写读一致性（writes-follow-reads consistency)

顾名思义，写读一致性就是在读操作后面的写操作基于至少跟上一次读出来一样新的值。也就是说，如果进程在地点 1 读了 x，那么在地点 2 要写 x 的副本的话，至少写的时候应该是基于至少和地点 1 读出的一样新的值。

举个例子，用户先读了文章 A，然后他回复了一篇文章 B。为了满足读写一致性，B 被写入任何副本之前，需要保证 A 也必须已经被写入那个副本。即，当原文章存储在某个本地副本上时，该文章的回应文章才能被存储到这个本地副本上。

1.5 容错性

集中式系统中任何一个组件的故障，都会导致整个系统无法正常使用。这就是为什么集中式系统的主机往往要求有比较高的性能。而分布式系统区别于集中式系统的一个特性是它容许部分失效。

分布式系统设计中的一个重要目标是以这样的方式构建系统：它可以从部分失效中自动恢复，而且不会严重地影响整体性能。特别是当故障发生时，分布式系统应该在进行恢复的同时继续以可接受的方式进行操作，也就是说，它应该能容忍错误，在发生错误时某种程度上可以继续操作。

1.5.1 基本概念

容错往往与可靠的系统紧密相关，而可靠的系统需要满足以下要求。

- **可用性**（Availability）：用来描述系统在给定时刻可以正确地工作。
- **可靠性**（Reliability）：指系统在可以无故障地连续运行。与可用性相反，可靠性是根据时间间隔而不是任何时刻来进行定义的。如果系统在每小时中崩溃的时间为 1ms，那么它的可用性就超过 99.9999%，但是它还是高度不可靠的。与之相反，如果一个系统从

来不崩溃，但是要在每年 8 月中停机两个星期，那么它是高度可靠的，但是它的可用性只有 98%。因此，这两种属性并不相同。

- **安全性**（Safety）：指系统在偶然出现故障的情况下能正确操作而不会造成任何灾难。
- **可维护性**（Maintainability）：发生故障的系统被恢复的难易程度。

容错，意味着即使系统发生了故障，还能正常提供服务。

1.5.2　故障分类

故障通常被分为三类。

- **暂时故障**（Transient fault）：只发生一次，然后就消失了，不再重现该故障。
- **间歇故障**（Intermittent fault）：发生，消失不见，而后再次发生，如此反复进行。
- **持久故障**（Permanent fault）：那些直到故障组件被修复之前持续存在的故障。

分布式系统中的典型故障模式可以分为以下几种。

- **崩溃性故障**（Crash failure）：服务器停机，但是在停机之前工作正常。
- **遗漏性故障**（Omission failure）：服务器不能响应到来的请求。可以细分为服务期不能接收到来的消息，或者是服务期不能发送消息。
- **定时性故障**（Timing failure）：服务器对请求响应得过快或者过慢。
- **响应性故障**（Response failure）：服务器对请求以错误的方式进行了响应。
- **任意性故障**（Arbitrary failure）：服务器可能在任意的时间产生任意类型的故障。其中，任意性故障是最严重的故障，也被称为拜占庭故障（Byzantine failure）。当发生故障时，服务器可能产生它从来没有产生过的输出，但是又不能检测出错误。更坏的情况是，发生故障的服务器恶意地与其他服务器共同工作来产生恶意的错误结果。臭名昭著的 Windows 系统“蓝屏”，正是为了尽可能避免这种情况而设计的。这种情况也说明了为什么谈到可靠系统时安全被认为是一个重要的需求。

术语“拜占庭”是指拜占庭帝国，它存在的时间是公元 330 年到 1453 年，地点在巴尔干半岛和现在的土耳其，在当时的统治阶级中存在着无休止的阴谋诡计和谎言。拜占庭故障的问题由 Pease、Lamport 等首先进行了分析。

随意性故障与崩溃性故障紧密相关。崩溃性故障的一个典型的例子就是操作系统崩溃，此时的解决办法只有一个：重新启动。与人们的期望相反，PC 经常遭遇崩溃性故障，最终导致设计者们把复位按钮从机箱背后移到前面。

1.5.3 使用冗余来掩盖故障

如果系统是容错的，那么它能做的最好的事情就是对其他进程隐藏故障的发生。关键技术是使用冗余来掩盖故障。有三种可能：信息冗余、时间冗余和物理冗余。

在信息冗余中，添加额外的位可以使错乱的位恢复正常。例如可以在传输的数据中添加一段 Hamming 码来从传输线路上的噪声中恢复数据。可以利用信息冗余进行错误检测和纠正。

在时间冗余中，执行一个动作，如果需要就再次执行。使用事务就是这种方法的一个例子。如果一个事务中止，那么它就可以无害地重新执行。当发生临时性或间歇性的错误时，时间冗余特别有用。TCP/IP 协议中的重传机制是另外一个例子。

在物理冗余中，通过添加额外的装备或进程使系统作为一个整体来容忍部分组件的失效或故障成为可能。物理冗余可以在硬件上也可以在软件上进行。其中，一种著名的设计是 TMR（Triple Modular Redundancy，三倍模块冗余）。在包括 TMR 的系统中，每个关键模块中的部件都被复制了三份，采用多数表决的方法，确保当某些某块中的单个部件发生故障时，系统还可以正确地运行。

1.5.4 分布式提交

在分布式系统中，事务往往包含多个参与者的活动，单个参与者的活动是能够保证原子性的，而保证多个参与者之间原子性则需要通过两阶段提交或者三阶段提交算法实现。

1. 两阶段提交

两阶段提交协议（Two-phase commit protocol，2PC）的过程涉及协调者和参与者。协调者可以看作事务的发起者，同时也是事务的一个参与者。对于一个分布式事务来说，一个事务是涉及多个参与者的。具体的两阶段提交的过程如下所示。

第一阶段（准备阶段）

- 协调者节点向所有参与者节点询问是否可以执行提交操作（vote），并开始等待各参与者节点的响应。
- 参与者节点执行所有事务操作，并将 Undo 信息和 Redo 信息写入日志（注意：若成功这里其实每个参与者已经执行了事务操作）。
- 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个“同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个“中止”消息。

第二阶段（提交阶段）

如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚（Rollback）消息；否则，发送提交（Commit）消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源（注意：必须在最后阶段释放锁资源）。

- 当协调者节点从所有参与者节点处获得的相应消息都为“同意”时：
  - 协调者节点向所有参与者节点发出“正式提交（commit）”的请求。
  - 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
  - 参与者节点向协调者节点发送“完成”消息。
- 如果任一参与者节点在第一阶段返回的响应消息为“中止”，或者协调者节点在第一阶段的询问在超时之前无法获取所有参与者节点的响应消息时：
  - 协调者节点向所有参与者节点发出“回滚操作（rollback）”的请求。
  - 参与者节点利用之前写入的 Undo 信息执行回滚，并释放在整个事务期间内占用的资源。
  - 参与者节点向协调者节点发送“回滚完成”消息。
  - 协调者节点收到所有参与者节点反馈的“回滚完成”消息后，取消事务。
  - 协调者节点收到所有参与者节点反馈的“完成”消息后，完成事务。

不管最后结果如何，第二阶段都会结束当前事务。

两段式提交协议的优缺点如下所示。

优点：原理简单，实现方便。

缺点：

- 同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。
- 单点故障。由于协调者的重要性，一旦协调者发生故障，参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者都还处于锁定事务资源的状态中，而无法继续完成事务操作。
- 数据不一致。在阶段二中，当协调者向参与者发送 commit 请求之后，发生了局部网络异常，或者在发送 commit 请求过程中协调者发生了故障，这会导致只有一部分参与者接收到了 commit 请求。而在这部分参与者接收到 commit 请求之后就会执行 commit 操作。但是其他部分未接收到 commit 请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。
- 两阶段无法解决的问题：协调者再发出 commit 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

为了解决两阶段提交协议的种种问题，研究者们在两阶段提交的基础上做了改进，提出了三阶段提交。

2. 三阶段提交

三阶段提交协议（Three-phase commit protocol，3PC）是两阶段提交（2PC）的改进版本。与两阶段提交不同的是，三阶段提交有两个改动点：

- 引入超时机制，同时在协调者和参与者中都引入超时机制。
- 在第一阶段和第二阶段中插入一个准备阶段，保证了在最后提交阶段之前各参与节点的状态是一致的。

即 3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 CanCommit、PreCommit、DoCommit 三个阶段。

CanCommit 阶段

CanCommit 阶段其实和 2PC 的准备阶段很像。协调者向参与者发送 commit 请求，参与者如果可以提交就返回 Yes 响应，否则返回 No 响应。

- **事务询问**：协调者向参与者发送 CanCommit 请求，询问是否可以执行事务提交操作，然后开始等待参与者的响应。
- **响应反馈**：参与者接收到 CanCommit 请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回 Yes 响应，并进入预备状态，否则返回 No 响应。

PreCommit 阶段

协调者根据参与者的反应情况来决定是否可以执行事务的 PreCommit 操作。根据响应情况，有以下两种可能。

- 假如协调者从所有的参与者处获得的反馈都是 Yes 响应，那么就会执行事务的预执行。
    - **发送预提交请求**：协调者向参与者发送 PreCommit 请求，并进入 Prepared 阶段。
    - **事务预提交**:参与者接收到 PreCommit 请求后，会执行事务操作，并将 undo 和 redo 信息记录到事务日志中。
    - **响应反馈**：如果参与者成功地执行了事务操作，则返回 ACK 响应，同时开始等待最终指令。
- 假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后，协调者都没有接收到参与者的响应，那么就执行事务的中断操作。

- **发送中断请求**：协调者向所有参与者发送 abort 请求。
- **中断事务**：参与者接收到来自协调者的 abort 请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断操作。

doCommit 阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

- 执行提交
    - **发送提交请求**：协调接收到参与者发送的 ACK 响应，那么它将从预提交状态进入到提交状态，并向所有参与者发送 doCommit 请求。
    - **事务提交**：参与者接收到 doCommit 请求之后，执行正式的事务提交，并在完成事务提交之后释放所有事务资源。
    - **响应反馈**：事务提交完之后，向协调者发送 ACK 响应。
    - **完成事务**：协调者接收到所有参与者的 ACK 响应之后，完成事务。
- **中断事务**：协调者没有接收到参与者发送的 ACK 响应（可能是接收者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务操作。
    - **发送中断请求**：协调者向所有参与者发送 abort 请求。
    - **事务回滚**：参与者接收到 abort 请求之后，利用其在阶段二记录的 undo 信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
    - **反馈结果**：参与者完成事务回滚之后，向协调者发送 ACK 消息。
    - **中断事务**：协调者接收到参与者反馈的 ACK 消息之后，执行事务的中断。

在 doCommit 阶段，如果参与者无法及时接收到来自协调者的 doCommit 或者 rebort 请求时，会在等待超时之后，继续进行事务的提交。即当进入第三阶段时，由于网络超时等原因，虽然参与者没有接收到 commit 或者 abort 响应，事务仍然会提交。

三阶段提交不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的 abort 响应没有及时被参与者接收到，那么参与者在等

待超时之后执行了 commit 操作，这样就和其他接到 abort 命令并执行回滚的参与者之间存在数据不一致的情况。

3. Paxos 算法

Paxos 算法是 Leslie Lamport 于 1990 年提出的一种基于消息传递且具有高度容错特性的一致性算法。Paxos 算法目前在 Google 的 Chubby、MegaStore、Spanner 等系统中得到了应用，Hadoop 中的 ZooKeeper 也使用了 Paxos 算法。

在 Paxos 算法中，分为 4 种角色。

- Proposer：提议者；
- Acceptor：决策者；
- Client：产生议题者；
- Learner：最终决策学习者。

算法可以分为两个阶段来执行。

阶段 1

- Proposer 选择一个议案编号 n，向 acceptor 的多数派发送编号也为 n 的 prepare 请求。
- Acceptor：如果接收到的 prepare 请求的编号 n 大于它已经回应的任何 prepare 请求，它就回应已经批准的编号最高的议案（如果有的话），并承诺不再回应任何编号小于 n 的议案。

阶段 2

- Proposer：如果收到了多数 acceptor 对 prepare 请求（编号为 n）的回应，它就向这些 acceptor 发送议案{n, v}的 accept 请求，其中 v 是所有回应中编号最高的议案的决议，或者是 proposer 选择的值，如果响应中不包含议案，那么它就是任意值。
- Acceptor：如果收到了议案{n, v}的 accept 请求，它就批准该议案，除非它已经回应了一个编号大于 n 的议案。
- Proposer 可以提出多个议案，只要它遵循上面的算法。它可以在任何时刻放弃一个议案（这不会破坏正确性，即使在议案被放弃后，议案的请求或者回应消息才到达目标）。如果其他 proposer 已经开始提出更高编号的议案，那么最好能放弃当前的议案。因此，如果 acceptor 忽略一个 prepare 或者 accept 请求（因为已经收到了更高编号的 prepare 请求），它应该告知 proposer 放弃议案。这是一个性能优化，而不影响正确性。

有关该算法的详细描述，可以在线参阅 http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf。

1.6　CAP 理论

在单机的数据库系统中，我们很容易就可以实现一套满足 ACID 特性的事务处理系统，事务的一致性不存在问题。但是在分布式系统中，由于数据分布在不同的主机节点上，如何对这些数据进行分布式的事务处理具有非常大的挑战。CAP 理论的出现，让我们对于分布式事务的一致性有了另外一种看法。

1.6.1　什么是 CAP 理论

在计算机科学理论，CAP 理论（也称为 Brewer 定理）是由计算机科学家 Eric Brewer 在 2000 年提出的，其理论观点是，在分布式计算机系统中不可能同时提供以下全部三个保证。

- **一致性**（Consistency）：所有节点同一时间看到的是相同的数据。
- **可用性**（Availability）：不管是否成功，确保每一个请求都能接收到响应。
- **分区容错性**（Partition tolerance）：系统任意分区后，在网络故障时，仍能操作。

CAP 定理如图 1-17 所示。

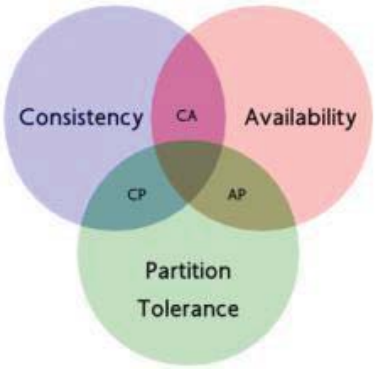

图 1-17　CAP 定理

在 2003 年的时候，Gilbert 和 Lynch 就正式证明了这三个特征确实是不可以兼得的。Gilbert 认为这里所说的一致性（Consistency）其实就是数据库系统中提到的 ACID 的另一种表述：一个用户请求要么成功、要么失败，不能处于中间状态（Atomic）；一旦一个事务完成，将来的所有事务都必须基于这个完成后的状态（Consistent）；未完成的事务不会互相影响（Isolated）；一旦一个事务完成，就是持久的（Durable）。 对于可用性（Availability），其概念没有变化，指的是对于一个系统而言，所有的请求都应该“成功”并且收到“返回”。分区容错性（Partition tolerance）指就是分布式系统的容错性。节点 crash 或者网络分片都不应该导致一个分布式系统

停止服务。

1.6.2 为什么说 CAP 只能三选二

下面分别举例说明为什么说 CAP 只能三选二。

图 1-18 显示了在一个网络中，N1 和 N2 两个节点，它们都共享数据块 V，其中有一个值 V0。运行在 N1 的 A 程序可以认为是安全的、无 bug、可预测的和可靠的。运行在 N2 的是 B 程序。在这个例子中，A 将写入 V 的新值，而 B 从 V 中读取值。

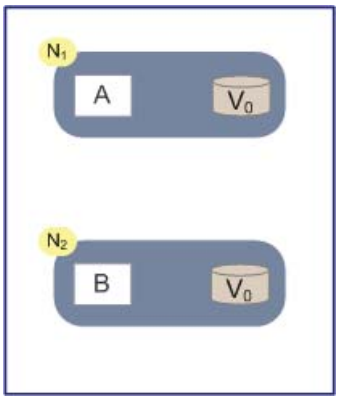

图 1-18 示例一

（图片选自 http://www.julianbrowne.com/article/viewer/brewers-cap-theorem）

系统预期执行下面的操作，如图 1-19 所示。

（1）首先写一个 V 的新值 V1。

（2）然后消息（M）从 N1 更新 V 的副本到 N2。

（3）现在，从 B 读取返回的 V1。

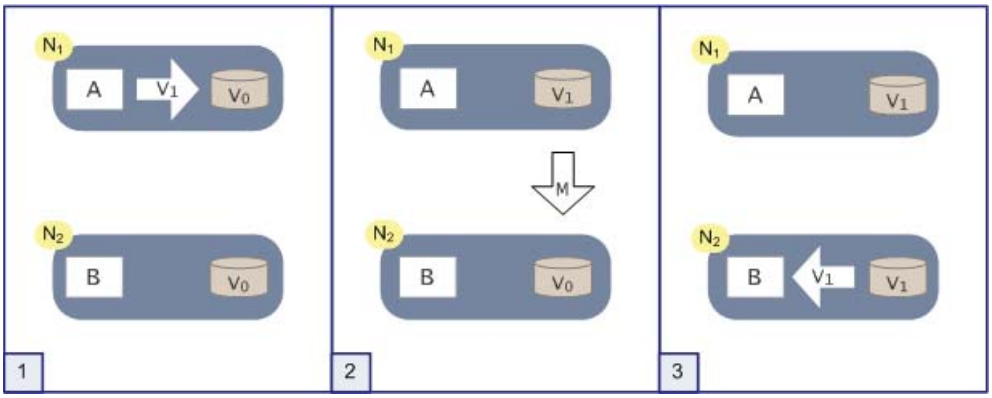

图 1-19　示例二

如果网络是分区的，当 N1 到 N2 的消息不能传递时，执行图 1-20 中的第三步，会出现虽然 N2 能访问到 V 的值（可用性），但其实与 N1 的 V 的值已经不一致了（一致性）。

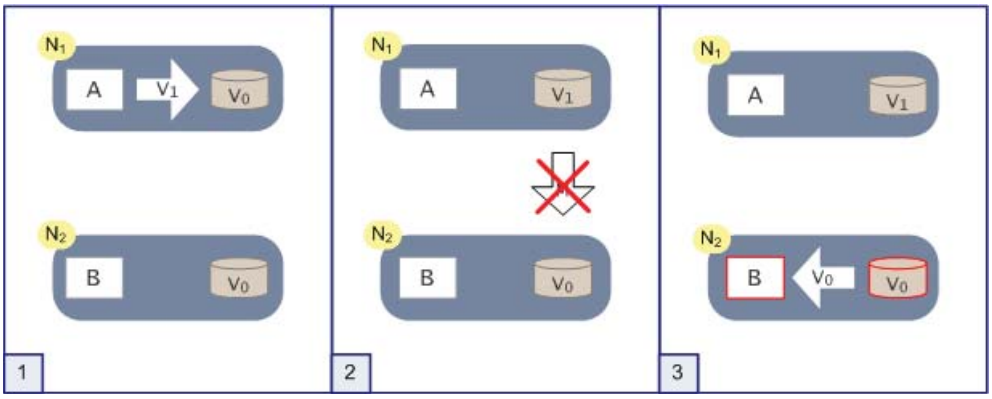

图 1-20　示例三

1.6.3　CAP 常见模型

既然 CAP 理论已经证明了一致性、可用性、分区容错性三者不可能同时达成。那么在实际应用中，可以在其中的某一些方面来放松条件，从而达到妥协。下面是常见的三种模型。

1. 牺牲分区（CA 模型）

牺牲分区容错性意味着把所有的机器搬到一台机器内部，或者放到一个“要死大家一起死”的机架上（当然机架也可能出现部分失效），这明显违背了我们希望的可伸缩性。

CA 模型常见的例子：

- 单站点数据库；
- 集群数据库；
- LDAP；
- xFS 文件系统。

实现方式：

- 两阶段提交；
- 缓存验证协议。

2. 牺牲可用性（CP 模型）

牺牲可用性意味着一旦系统中出现分区这样的错误，系统直接就停止服务。

CP 模型常见的例子：

- 分布式数据库；
- 分布式锁定；
- 绝大部分协议。

实现方式：

- 悲观锁；
- 少数分区不可用。

3. 牺牲一致性（AP 模型）

AP 模型常见的例子：

- Coda；
- Web 缓存；
- DNS。

实现方式：

- 到期/租赁；
- 解决冲突；
- 乐观。

1.6.4 CAP 的意义

在系统架构时，应该根据具体的业务场景来权衡 CAP。比如，对于大多数互联网应用来说

（如门户网站），因为机器数量庞大，部署节点分散，网络故障是常态的，可用性是必须要保证的，所以只有舍弃一致性来保证服务的 AP。而对于银行等需要确保一致性的场景，通常会权衡 CA 和 CP 模型，CA 模型网络故障时完全不可用，CP 模型具备部分可用性。

1.6.5　CAP 最新发展

Eric Brewer 在 2012 年发表文章[3]，指出了 CAP 里面“三选二”的做法存在一定的误导性。主要体现在：

- 由于分区很少发生，那么在系统不存在分区的情况下没什么理由牺牲 C 或 A；
- C 与 A 之间的取舍可以在同一系统内以非常细小的粒度反复发生，而每一次的决策可能因为具体的操作，乃至因为牵涉特定的数据或用户而有所不同；
- 这三种性质都可以在一定程度上衡量，并不是非黑即白的有或无。可用性显然是在 0% 到 100%之间连续变化的，一致性分很多级别，连分区也可以细分为不同含义，如系统内的不同部分对于是否存在分区可以有不一样的认知。

理解 CAP 理论最简单的方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了 C 性质。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了 A 性质。除非两个节点可以互相通信，才能既保证 C 又保证 A，但这又会导致丧失 P 性质。一般来说跨区域的系统，设计师无法舍弃 P 性质，那么就只能在数据一致性和可用性上做一个艰难选择。不确切地说，NoSQL 运动的主题其实是创造各种可用性优先、数据一致性其次的方案；而传统数据库坚守 ACID 特性，做的是相反的事情。

BASE

BASE（Basically Available、Soft state、Eventual consistency）来自于互联网的电子商务领域的实践，它是基于 CAP 理论逐步演化而来的，核心思想是即便不能达到强一致性（Strong consistency），但可以根据应用特点采用适当的方式来达到最终一致性（Eventual consistency）的效果。BASE 是对 CAP 中 C 和 A 的延伸。BASE 的含义如下所示。

- Basically Available：基本可用。
- Soft state：软状态/柔性事务，即状态可以有一段时间的不同步。
- Eventual consistency：最终一致性。

BASE 是反 ACID 的，它完全不同于 ACID 模型，牺牲强一致性，获得基本可用性和柔性可靠性并要求达到最终一致性。

---

[3] 该文章可以在线查阅 https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules- have-changed。

1.7 安全性

计算机的安全性通常包括两个部分：认证和访问控制。认证包括对有效用户身份的确认和识别。而访问控制则致力于避免对数据文件和系统资源的有害篡改。举例来说，在一个孤立、集中、单用户系统中（例如一台计算机），通过锁上存放该计算机的房间并将磁盘锁起来就能够实现其安全性。因此只有拥有房间和磁盘钥匙的用户才能访问系统资源和文件。这就同时实现了认证和访问控制。因此，安全性实际上就相当于锁住计算机和房间的钥匙。

分布式系统的安全相比于集中式的系统将会复杂得多，因为安全涉及整个系统，所以有关安全的任何一个单纯的设计缺陷都可能导致所有的安全措施无效。分布式系统是由各个子系统组成的，子系统之间也需要做身份识别；各子系统都是通过网络进行连接的，那么它们之间的安全通信也是需要保障的。

由于安全是一个复杂的话题，并不是本书所要阐述的重点，所以下面章节只会涉及分布式系统安全方面的基本知识。

1.7.1 基本概念

1.安全威胁、策略和机制

计算机系统中的安全性与可靠性密切相关。非正式地说，一个可靠的计算机系统是一个我们可以有理由信任其所提供的服务的系统。可靠性包括可用性、可信赖性、安全性和可维护性。但是，如果我们要信任一个计算机系统，还应该考虑机密性和完整性。考虑计算机系统中安全的另一种角度是我们试图保护该系统所提供的服务和数据不受到安全威胁，包括 4 种安全威胁。

- **窃听**：指一个未经授权的用户获得了对一项服务或数据的访问权限。窃听的一个典型事例是两人之间的通信被其他人偷听到。窃听还发生在非法复制数据时。
- **中断**：指服务或数据变得难以获得、不能使用、被破坏等情况。在这个意义上说，服务拒绝攻击是一种安全威胁，它归为中断类，一些人正是通过它恶意地试图使其他人不能访问服务。中断的一个实例是文件被损坏或丢失时所出现的情况。
- **修改**：包括对数据未经授权的改变或篡改一项服务以使其不再遵循其原始规范。修改的实例包括窃听，然后改变传输的数据，篡改数据库条目以及改变一个程序使其秘密记录其用户的活动。
- **伪造**：指产生通常不存在的附加数据或活动的情况。例如，一个入侵者可能尝试向密码文件或数据库中添加数据，同样，有时可能通过重放先前发送过的消息来侵入一个系统。

仅仅声明系统应该能够保护其自身免受任何可能的安全威胁并不是实际建立一个安全系统

的方式。首先需要的是安全需求的一个描述，也就是一个策略。安全策略准确地描述了系统中的实体能够采取的行为以及禁止采取的行动。实体包括用户、服务、数据、机器等。制定了安全策略之后，就可能集中考虑安全机制，策略通过该机制来实施。重要的机制包括：

- 加密；
- 身份验证；
- 授权；
- 审计。

加密是计算机的基础。加密将数据转换为一些攻击者不能理解的形式。身份验证用于检验用户、客户、服务器等所声明的身份。对一个客户进行身份验证之后，有必要检查是否授予客户执行该请求操作的权限。审计工具用于追踪各个客户的访问内容以及访问方式。

2. 密码与数字签名

加密包括使用密钥对数据进行编码，从而使偷听者无法方便地阅读这些数据。经过加密的数据称为**密文**，原始的数据称为**明文**。从密文到明文的转换过程称为解密。

图 1-21 描述了 Alice 和 Bob 通过凯撒（Caesar）密码方式进行的通信过程，这里的 Alice 和 Bob 常用来表示密码通信的双方。

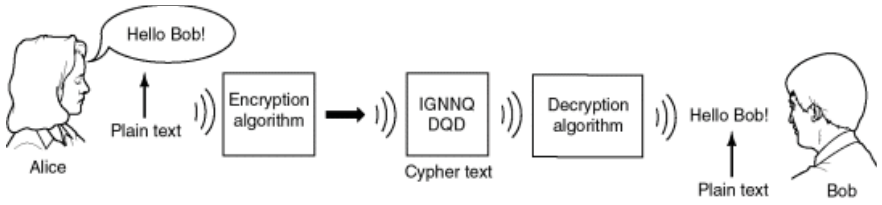

图 1-21　Caesar 密码加解密过程

（注：图片选自 http://www.microsoft.com/china/security/bestprac/ch11ce.mspx）

凯撒密码又叫循环移位密码，它的加密方法就是将明文中的每个字母用字母表中该字母后的第 R 个字母来替换，从而达到加密的目的。

它的加密过程可以表示为下面的函数：

```
E(m)=(m+k)modn
```

其中，m 为明文字母在字母表中的位置数；n 为字母表中的字母个数；k 为密钥；E(m)为密文字母在字母表中对应的位置数。例如，对于明文字母 H，其在字母表中的位置数为 8，则按照上式计算出来的密文为 L，计算过程如下：

```
E(8)=（8+4）mod26=12
```

上面例子中，每个字母用其后面的第三个字母代替。

下面是一个 C 语言实现的凯撒密码程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char *caesar(const char *str,int offset)
{
    char *start,*ret_str;
    start = ret_str = (char *) malloc(strlen(str) + 1);
    for(;*str!='\0';str++,ret_str++)
    {
        if(*str>='A' && *str<='Z')
            *ret_str = 'A' + (*str - 'A' + offset) % 26;
        else if(*str>='a' && *str<='z')
            *ret_str = 'a' + (*str - 'a' + offset) % 26;
        else
            *ret_str = *str;
    }
    *ret_str = '\0';
    return (char *) start;
}

int main(void)
{
    printf("%s\n","ABCDEFGHIJKLMNOPQRSTUVWXYZ");
    printf("%s\n",caesar("ABCDEFGHIJKLMNOPQRSTUVWXYZ",3));
    return 0;
}
```

1.7.2 加密算法

评估一种加密算法安全性最常用的方法是判断该算法是否是计算安全的。如果利用可用资源进行系统分析后无法攻破系统，那么这种加密算法就是计算安全的。目前有两种常用的加密类型：私钥加密和公钥加密。除了加密整条消息，两种加密类型都可以用来对一个文档进行数字签名。当使用一个足够长的密钥时，密码被破解的难度就会越大，系统也就越安全。当然，密钥越长，本身加解密的成本也就越高。

1. 对称加密

对称加密：指的是加密和解密算法都使用相同密钥的加密算法。具体如下：

```
E(p,k)=C ； D(C,k)=p
```

其中

- E = 加密算法；
- D = 解密算法；
- p = 明文（原始数据）；
- k = 加密密钥；
- C = 密文。

由于在加密和解密数据时使用了同一个密钥，因此这个密钥必须保密。这样的加密也称为秘密密钥算法，或单密钥算法/常规加密。很显然，对称算法的安全性依赖于密钥，泄漏密钥就意味着任何人都可以对他们发送或接收的消息解密，所以密钥的保密性对通信的安全性至关重要。

对称加密算法的特点是算法公开、计算量小、加密速度快、加密效率高。

不足之处是，交易双方都使用同样钥匙，安全性得不到保证。此外，每对用户每次使用对称加密算法时，都需要使用其他人不知道的唯一钥匙，这会使得发收信双方所拥有的钥匙数量呈几何级数增长，密钥管理成为用户的负担。对称加密算法在分布式网络系统上使用较为困难，主要是因为密钥管理困难，使用成本较高。而与公开密钥加密算法比起来，对称加密算法能够提供加密和认证却缺乏了签名功能，使得使用范围有所缩小。

常见的对称加密算法有 DES、3DES、TDEA、Blowfish、RC2、RC4、RC5、IDEA、SKIPJACK、AES 等。

2. 使用对称密钥加密的数字签名

在通过网络发送数据的过程中，有两种对文档进行数字签名的基本方法。在这里我们讨论第一种方法，利用私钥加密法。数字签名也称为**消息摘要**（Message Digest）或**数字摘要**（Digital Digest），它是一个唯一对应一个消息或文本的固定长度的值，它由一个单向 Hash 加密函数对消息进行作用而产生。如果消息在途中改变了，则接收者通过对收到的消息新产生的摘要与原摘要比较，就可以知道消息是否被改变了。因此消息摘要保证了消息的完整性。消息摘要采用单向 Hash 函数将需加密的明文“摘要”成一串 128bit 的密文，这一串密文也称为**数字指纹**（Finger Print），它有固定的长度，且不同的明文摘要成密文，其结果总是不同的，而同样的明文其摘要必定一致。这样这串摘要便可成为验证明文是否是“真身”的“指纹”了。有两种方法可以利用共享的私钥来计算摘要。最简单、快捷的方法是计算消息的哈希值，然后通过私钥对这个数

值进行加密。然后消息和已加密的摘要一起发送。接收者可以再次计算消息摘要，对摘要进行加密，并与接收到的加密摘要进行比较。如果这两个加密摘要相同，那就说明该文档没有被改动。第二种方法将私钥应用到消息上，然后计算哈希值。这种方法的过程如下：

计算公式为：

```
D(M,K)
```

其中

- D 是摘要函数；
- M 是消息；
- K 是共享的私钥。

然后可以发布或分发这个文档。由于第三方并不知道私钥，而计算正确的摘要值恰恰需要它，因此消息摘要能够避免对摘要值自身的伪造。在这两种情况下，只有那些了解秘密密钥的用户才能验证其完整性，所有欺骗性的文档都可以很容易地检验出来。

消息摘要算法有：MD2、MD4、MD5、SHA-1、SHA-256、RIPEMD128、RIPEMD160 等。

3. 非对称加密

非对称加密（也称为公钥加密）由两个密钥组成，包括公开密钥（public key，简称公钥）和私有密钥（private key，简称私钥）。如果信息使用公钥进行加密，那么通过使用相对应的私钥可以解密这些信息，过程如下：

```
E(p,ku)=C ;  D(C,kr)=p
```

其中

- E = 加密算法；
- D = 解密算法；
- p = 明文（原始数据）；
- ku = 公钥；
- kr = 私钥；
- C = 密文。

如果信息使用私钥进行加密，那么通过使用其相对应的公钥可以解密这些信息，过程如下：

```
E(p,kr)=C ;  D(C,ku)=p
```

其中

- E = 加密算法；

- D = 解密算法；
- p = 明文（原始数据）；
- ku = 公钥；
- kr = 私钥；
- C = 密文。

不能使用加密所用的密钥来解密一个消息。而且，由一个密钥计算出另一个密钥从数学上来说是很困难的。私钥只有用户本人知道，因此得名。公钥并不保密，可以通过公共列表服务获得，通常公钥是使用 X.509 实现的。公钥加密的想法最早是由 Diffie 和 Hellman 于 1976 年提出的。

非对称加密与对称加密相比，其安全性更好。对称加密的通信双方使用相同的秘钥，如果一方的秘钥遭泄露，那么整个通信就会被破解。而非对称加密使用一对秘钥，一个用来加密，一个用来解密，而且公钥是公开的，秘钥是自己保存的，不需要像对称加密那样在通信之前要先同步秘钥。

非对称加密的缺点是加密和解密花费时间长、速度慢，只适合对少量数据进行加密。

在非对称加密中使用的主要算法有：RSA、Elgamal、背包算法、Rabin、D-H、ECC（椭圆曲线加密算法）等。

4. 使用公钥加密的数字签名

用于数字签名的公钥加密使用 RSA 算法。在这种方法中，发送者利用私钥通过摘要函数对整个数据文件（代价昂贵）或文件的签名进行加密。私钥匹配最主要的优点就是不存在密钥分发问题。这种方法假定你信任发布公钥的来源。然后接收者可以利用公钥来解密签名或文件，并验证它的来源和/或内容。由于公钥密码学的复杂性，因此只有正确的公钥才能够解密信息或摘要。最后，如果你要将消息发送给拥有已知公钥的用户，那么你就可以使用接收者的公钥来加密消息或摘要，这样只有接收者才能够通过他们自己的私钥来验证其中的内容。

1.7.3　安全通道

1. SSL/TLS

SSL（Secure Sockets Layer，安全套接字层）是在网络上应用最广泛的加密协议实现。SSL 使用结合加密过程来提供网络的安全通信。

SSL 提供了一个安全的增强标准 TCP/IP 套接字用于网络通信协议。如表 1-2 所示，在标准 TCP/IP 协议栈的传输层和应用层之间添加了完全套接字层。SSL 的应用程序中最常用的是

Hypertext Transfer Protocol（HTTP，超文本传输协议），这个是互联网网页协议。其他应用程序，如 Net News Transfer Protocol（NNTP，网络新闻传输协议）、Telnet、 Lightweight Directory Access Protocol（LDAP，轻量级目录访问协议）、Interactive Message Access Protocol（IMAP，互动信息访问协议）和 File Transfer Protocol（FTP，文件传输协议），也可以使用 SSL。

注：目前还没有标准的安全的 FTP。

表 1-2　增强标准的 TCP/IP 层与协议

TCP/IP 层	协　　议
Application Layer	HTTP、NNTP、Telnet、FTP 等
Secure Sockets Layer	SSL
Transport Layer	TCP
Internet Layer	IP

SSL 最初是由网景公司在 1994 年创立的，现在已经演变成为一个标准。由国际标准组织 Internet Engineering Task Force（IETF）进行管理。之后 IETF 更名为 SSL 为 Transport Layer Security（TLS，传输层安全），并在 1999 年 1 月发布了第一个规范，版本为 1.0。TLS 1.0 对于 SSL 的最新版本 3.0 版本是一个小的升级。两者差异非常微小。TLS 1.1 是在 2006 年 4 月发布的，TLS 1.2 在 2008 年 8 月发布。

2. SSL 握手过程

SSL 通过握手过程在客户端和服务器之间协商会话参数，并建立会话。会话包含的主要参数有会话 ID、对方的证书、加密套件（密钥交换算法、数据加密算法和 MAC 算法等）以及主密钥（master secret）。通过 SSL 会话传输的数据，都将采用该会话的主密钥和加密套件进行加密、计算 MAC 等处理。

不同情况下，SSL 的握手过程存在差异。下面将分别描述以下三种情况下的握手过程：

- 只验证服务器的 SSL 握手过程；
- 验证服务器和客户端的 SSL 握手过程；
- 恢复原有会话的 SSL 握手过程。

只验证服务器的 SSL 握手过程如图 1-22 所示。

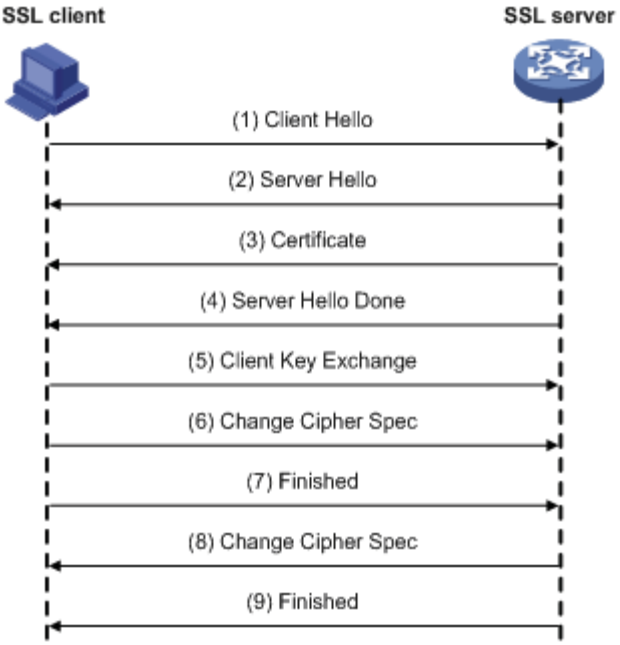

图 1-22　只验证服务器的 SSL 握手过程

如图 1-22 所示，只需要验证 SSL 服务器身份，不需要验证 SSL 客户端身份时，SSL 的握手过程为：

（1）SSL 客户端通过 Client Hello 消息将它支持的 SSL 版本、加密算法、密钥交换算法、MAC 算法等信息发送给 SSL 服务器。

（2）SSL 服务器确定本次通信采用的 SSL 版本和加密套件，并通过 Server Hello 消息通知给 SSL 客户端。如果 SSL 服务器允许 SSL 客户端在以后的通信中重用本次会话，则 SSL 服务器会为本次会话分配会话 ID，并通过 Server Hello 消息发送给 SSL 客户端。

（3）SSL 服务器将携带自己公钥信息的数字证书通过 Certificate 消息发送给 SSL 客户端。

（4）SSL 服务器发送 Server Hello Done 消息，通知 SSL 客户端版本和加密套件协商结束，开始进行密钥交换。

（5）SSL 客户端验证 SSL 服务器的证书合法后，利用证书中的公钥加密 SSL 客户端随机生成的 premaster secret，并通过 Client Key Exchange 消息发送给 SSL 服务器。

（6）SSL 客户端发送 Change Cipher Spec 消息，通知 SSL 服务器后续报文将采用协商好的密钥和加密套件进行加密和 MAC 计算。

（7）SSL 客户端计算已交互的握手消息（除 Change Cipher Spec 消息外所有已交互的消息）的 Hash 值，利用协商好的密钥和加密套件处理 Hash 值（计算并添加 MAC 值、加密等），并通

过 Finished 消息发送给 SSL 服务器。SSL 服务器利用同样的方法计算已交互的握手消息的 Hash 值，并与 Finished 消息的解密结果比较，如果二者相同，且 MAC 值验证成功，则证明密钥和加密套件协商成功。

（8）同样地，SSL 服务器发送 Change Cipher Spec 消息，通知 SSL 客户端后续报文将采用协商好的密钥和加密套件进行加密和 MAC 计算。

（9）SSL 服务器计算已交互的握手消息的 Hash 值，利用协商好的密钥和加密套件处理 Hash 值（计算并添加 MAC 值、加密等），并通过 Finished 消息发送给 SSL 客户端。SSL 客户端利用同样的方法计算已交互的握手消息的 Hash 值，并与 Finished 消息的解密结果比较，如果二者相同，且 MAC 值验证成功，则证明密钥和加密套件协商成功。

SSL 客户端接收到 SSL 服务器发送的 Finished 消息后，如果解密成功，则可以判断 SSL 服务器是数字证书的拥有者，即 SSL 服务器身份验证成功，因为只有拥有私钥的 SSL 服务器才能从 Client Key Exchange 消息中解密得到 premaster secret，从而间接地实现了 SSL 客户端对 SSL 服务器的身份验证。

验证服务器和客户端的 SSL 握手过程如图 1-23 所示。

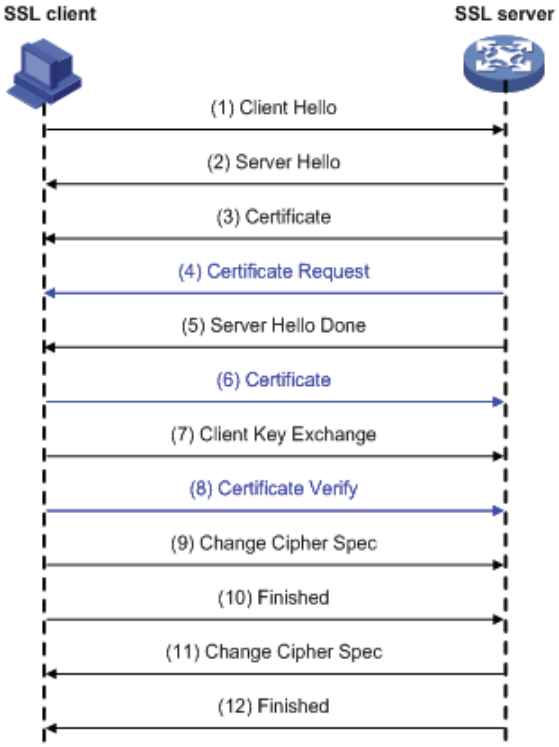

图 1-23　验证服务器的 SSL 握手过程

SSL 客户端的身份验证是可选的，由 SSL 服务器决定是否验证 SSL 客户端的身份。如图 1-23 中（4）、（6）、（8）部分所示，如果 SSL 服务器验证 SSL 客户端身份，则 SSL 服务器和 SSL 客户端除了交互“只验证服务器的 SSL 握手过程”中的消息协商密钥和加密套件，还需要进行以下操作：

（1）SSL 服务器发送 Certificate Request 消息，请求 SSL 客户端将其证书发送给 SSL 服务器。

（2）SSL 客户端通过 Certificate 消息将携带自己公钥的证书发送给 SSL 服务器。SSL 服务器验证该证书的合法性。

（3）SSL 客户端计算已交互的握手消息、主密钥的 Hash 值，利用自己的私钥对其进行加密，并通过 Certificate Verify 消息发送给 SSL 服务器。

（4）SSL 服务器计算已交互的握手消息、主密钥的 Hash 值，利用 SSL 客户端证书中的公钥解密 Certificate Verify 消息，并将解密结果与计算出的 Hash 值比较。如果二者相同，则 SSL 客户端身份验证成功。

恢复原有会话的 SSL 握手过程如图 1-24 所示。

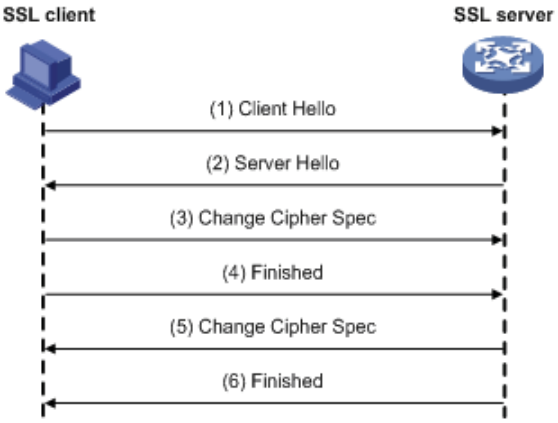

图 1-24　恢复原有会话的 SSL 握手过程

协商会话参数、建立会话的过程中，需要使用非对称密钥算法来加密密钥、验证通信对端的身份，计算量较大，占用了大量的系统资源。为了简化 SSL 握手过程，SSL 允许重用已经协商过的会话，具体过程为：

（1）SSL 客户端发送 Client Hello 消息，消息中的会话 ID 设置为计划重用的会话的 ID。

（2）SSL 服务器如果允许重用该会话，则通过在 Server Hello 消息中设置相同的会话 ID 来应答。这样，SSL 客户端和 SSL 服务器就可以利用原有会话的密钥和加密套件，不必重新协商。

（3）SSL 客户端发送 Change Cipher Spec 消息，通知 SSL 服务器后续报文将采用原有会话的密钥和加密套件进行加密和 MAC 计算。

（4）SSL 客户端计算已交互的握手消息的 Hash 值，利用原有会话的密钥和加密套件处理 Hash 值，并通过 Finished 消息发送给 SSL 服务器，以便 SSL 服务器判断密钥和加密套件是否正确。

（5）同样地，SSL 服务器发送 Change Cipher Spec 消息，通知 SSL 客户端后续报文将采用原有会话的密钥和加密套件进行加密和 MAC 计算。

（6）SSL 服务器计算已交互的握手消息的 Hash 值，利用原有会话的密钥和加密套件处理 Hash 值，并通过 Finished 消息发送给 SSL 客户端，以便 SSL 客户端判断密钥和加密套件是否正确。

3. HTTPS

HTTPS（Hyper Text Transfer Protocol over Secure Socket Layer）是基于 SSL 安全连接的 HTTP 协议。HTTPS 通过 SSL 提供的数据加密、身份验证和消息完整性验证等安全机制，为 Web 访问提供了安全性保证，广泛应用于网上银行、电子商务等领域。近年以来，在主要互联网公司和浏览器开发商的推动之下，HTTPS 在加速普及，HTTP 正在被加速淘汰。不加密的 HTTP 连接是不安全的，你和目标服务器之间的任何中间人都能读取和操纵传输的数据，比如 ISP 可以在你点击的网页上插入广告，你很可能根本不知道看到的广告是否是网站发布的。中间人能够注入的代码不仅仅是看起来无害的广告，他们还可能注入具有恶意目的的代码。2015 年，百度联盟广告的脚本被中间人修改，加入了代码对两个政府不喜欢的网站发动了 DDoS 攻击。这次攻击被称为网络大炮，网络大炮让普通的网民在不知情下变成了 DDoS 攻击者。而唯一能阻止大炮的方法是加密流量。

4. 在 Tomcat 中配置 SSL/TLS 以支持 HTTPS

由于 Tomcat 是市场占有率最高的 Java 开源 Servelet 容器，下面介绍如何在 Tomcat 中配置 SSL/TLS 以支持 HTTPS。

生成密钥和证书

Tomcat 目前只能操作 JKS、PKCS11、PKCS12 格式的密钥存储库。JKS 是 Java 标准的“Java 密钥存储库”格式，是通过 keytool 命令行工具创建的。该工具包含在 JDK 中。PKCS12 格式是一种互联网标准，可以通过 OpenSSL 和 Microsoft 的 Key-Manager 来操纵。

创建一个 keystore 文件来保存服务器的私有密钥和自签名证书。

在 Windows 下执行：

```
"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA
```

在 UNIX 下执行：

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA
```

执行该命令后，首先会提示你提供 keystore 的密码。Tomcat 默认使用的密码是 changeit（全部字母都小写），当然你可以指定一个自定义密码（如果你愿意）。同样，你也需要将这个自定义密码在 server.xml 配置文件内进行指定，稍后再予以详述。

接下来会提示关于证书的一般信息，比如组织、联系人名称，等等。当用户试图在你的应用中访问一个安全页面时，该信息会显示给用户，所以一定要确保所提供的信息与用户所期望看到的内容保持一致。

最后，还需要输入密钥密码（key password），这个密码是这一证书（而不是存储在同一密码存储库文件中的其他证书）的专有密码。keytool 提示会告诉你，如果按下回车键，则自动使用密码存储库 keystore 的密码。当然，除了这个密码，你也可以自定义自己的密码。如果选择自定义密码，那么不要忘了在 server.xml 配置文件中指定这一密码。

下面是详细步骤：

```
C:\Users\admin>"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA
输入密钥库口令：
再次输入新口令：
您的名字与姓氏是什么？
  [Unknown]:  waylau

您的组织单位名称是什么？
  [Unknown]:  waylau.com

您的组织名称是什么？
  [Unknown]:  waylau.com

您所在的城市或区域名称是什么？
  [Unknown]:  Hangzhou

您所在的省/市/自治区名称是什么？
  [Unknown]:  Zhejiang

该单位的双字母国家/地区代码是什么？
  [Unknown]:  china
CN=waylau, OU=waylau.com, O=waylau.com, L=hangzhou, ST=zhejiang, C=china
是否正确？
```

```
[否]: y

输入 <tomcat> 的密钥口令
        (如果和密钥库口令相同, 按回车):
```

如果操作全部正常，现在就会创建一个新的 JKS 密码存储库，该密码库包含一个自签名的证书。创建一个新的 JKS 密码存储库，该密码库包含一个自签名的证书。

该命令将在用户的主目录下创建一个新文件：.keystore。

要想指定一个不同的位置或文件名，可以在上述的 keytool 命令上添加-keystore 参数，后面跟到达 keystore 文件的完整路径名。还需要把这个新位置指定到 server.xml 配置文件上（见后文介绍）。

Windows：

```
"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA -keystore \path\
to\my\keystore
```

UNIX：

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA -keystore /path/to/my/keystore
```

修改配置

取消对 Tomcat 安装目录下/conf/server.xml 中“SSL HTTP/1.1 Connector”一项的注释状态，并指定 keystore 的路径和密码：

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
        maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
        keystoreFile="${user.home}/.keystore" keystorePass="changeit"
        clientAuth="false" sslProtocol="TLS" />
```

Tomcat 指定了 8443 端口为 HTTPS 访问端口。如果要隐藏端口号，就要把 Tomcat 的 HTTPS 端口设为 443。

若想把所有 HTTP 请求都转到 HTTPS 协议上，可以修改 Tomcat 的 conf 下的 web.xml，在 <welcome-file-list> </welcome-file-list>节点下方添加如下代码：

```
<security-constraint>
    <!-- Authorization setting for SSL -->
    <web-resource-collection >
        <web-resource-name >SSL</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
```

```
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

其中，<url-pattern>是配置文件过滤策略，比如只对.jsp 的请求自动转化为 HTTPS，配置如下：

```
<security-constraint>
    <web-resource-collection >
       <web-resource-name >SSL</web-resource-name>
       <url-pattern>*.jsp</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

在<url-pattern>中可以配置你希望自动转化的请求路径/*、login.html、login.jsp，等等。

虽然 SSL 协议的意图是尽可能有助于提供安全且高效的连接，但从性能角度来考虑，加密与解密是非常耗费计算资源的，因此将整个 Web 应用都运行在 SSL 协议下是完全没有必要的，开发者需要挑选需要安全连接的页面。对于一个相当繁忙的网站来说，通常只会在特定页面上使用 SSL 协议，也就是可能交换敏感信息的页面，比如：登录页面、个人信息页面、购物车结账页面（可能会输入信用卡信息），等等。应用中的任何一个页面都可以通过加密套接字来请求访问，只需将页面地址的前缀 http:换成 https:即可。

5. SSL VPN

SSL VPN 是以 SSL 为基础的 VPN 技术，利用 SSL 提供的安全机制，为用户远程访问公司内部网络提供了安全保证。与复杂的 IPSec VPN 相比，SSL 通过简单易用的方法实现了信息远程连通。任何安装浏览器的机器都可以使用 SSL VPN，这是因为 SSL 内嵌在浏览器中，它不需要像传统 IPSec VPN 一样必须为每一台客户机安装客户端软件。SSL VPN 通过在远程接入用户和 SSL VPN 网关之间建立 SSL 安全连接，允许用户通过各种 Web 浏览器，各种网络接入方式，在任何地方远程访问企业网络资源，并能够保证企业网络的安全，保护企业内部信息不被窃取。

一般而言，SSL VPN 必须满足最基本的两个要求：

- 使用 SSL 协议进行认证和加密；没有采用 SSL 协议的 VPN 产品自然不能称为 SSL VPN，其安全性也需要进一步考证。
- 直接使用浏览器完成操作，无须安装独立的客户端；即使使用了 SSL 协议，但仍然需要分发和安装独立的 VPN 客户端（如 Open VPN）不能称为 SSL VPN，否则就失去了

SSL VPN 易于部署、免维护的优点了。

SSL VPN 的典型组网模式如图 1-25 所示。

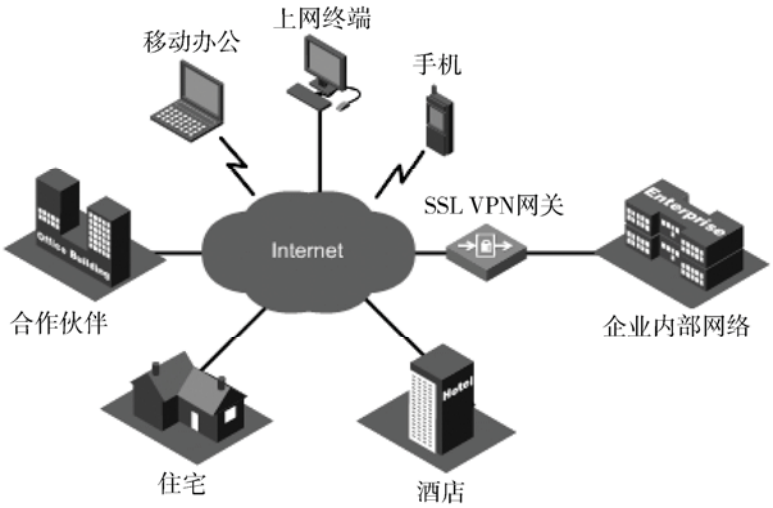

图 1-25　SSL VPN 的典型组网模式

1.7.4　访问控制

访问控制是指按用户身份及其所归属的某项定义组来限制用户对某些信息项的访问，或限制对某些控制功能的使用的一种技术。

访问控制的功能：

- 防止非法的主体进入受保护的网络资源；
- 允许合法用户访问受保护的网络资源；
- 防止合法的用户对受保护的网络资源进行非授权的访问。

1. 防火墙

防火墙指的是一个由软件和硬件设备组合而成、在内部网和外部网之间、专用网与公共网之间的界面上构造的保护屏障。这是一种获取安全性方法的形象说法，它是一种计算机硬件和软件的结合，使 Internet 与 Intranet 之间建立起一个安全网关（Security Gateway），从而保护内部网免受非法用户的侵入。防火墙主要由服务访问规则、验证工具、包过滤和应用网关 4 个部分组成，防火墙就是一个位于计算机和它所连接的网络之间的软件或硬件。该计算机流入/流出的所有网络通信和数据包均要经过此防火墙。

防火墙具备以下特性：

- 内部网络和外部网络之间的所有网络数据流都必须经过防火墙；
- 只有符合安全策略的数据流才能通过防火墙；
- 防火墙自身应具有非常强的抗攻击免疫力；
- 应用层防火墙具备更细致的防护能力；
- 数据库防火墙具有针对数据库恶意攻击的阻断能力。

2. 堡垒机

堡垒机，即在一个特定的网络环境下，为了保障网络和数据不受来自外部和内部用户的入侵和破坏，而运用各种技术手段实时收集和监控网络环境中每一个组成部分的系统状态、安全事件、网络活动，以便集中报警、记录、分析、处理的一种技术手段。

从功能上讲，它综合了核心系统运维和安全审计管控两大主干功能；从技术实现上讲，通过切断终端计算机对网络和服务器资源的直接访问，而采用协议代理的方式，接管了终端计算机对网络和服务器的访问。形象地说，终端计算机对目标的访问，均需要经过运维安全审计的翻译。打一个比方，运维安全审计扮演着看门者的工作，所有对网络设备和服务器的请求都要从这扇大门经过。因此运维安全审计能够拦截非法访问和恶意攻击，对不合法命令进行命令阻断，过滤掉所有对目标设备的非法访问行为，并对内部人员的误操作和非法操作进行审计监控，以便事后责任追踪。

安全审计作为企业信息安全建设不可缺少的组成部分，逐渐受到用户的关注，是企业安全体系中的重要环节。同时，安全审计是事前预防、事中预警的有效风险控制手段，也是事后追溯的可靠证据来源。

3. 拒绝服务

DoS（denial of service，拒绝服务）攻击是指通过向服务器发送大量垃圾信息或干扰信息的方式，导致服务器无法向正常用户提供服务的现象。对付一个或者多个资源的 DoS 往往比较高效，而要对付 DDoS（distributed denial of service，分布式拒绝服务）则要困难得多。

DDoS 攻击主要分为两类。

- **带宽耗竭的攻击**：往某个机器发送大量的消息，结果是正常的消息很难到达接收者。
- **资源耗竭的攻击**：使接收者把资源消耗在无用的消息上。

UDP 洪水攻击是指攻击者利用简单的 TCP/IP 服务，如 Chargen 和 Echo 来传送毫无用处的占满带宽的数据。通过伪造与某一主机的 Chargen 服务之间的一次 UDP 连接，回复地址指向开着 Echo 服务的一台主机，这样就生成在两台主机之间存在很多的无用数据流，这些无用数据流就会导致带宽的服务攻击。

SYN Flood 是当前最流行的 DoS 与 DDoS 的方式之一，这是一种利用 TCP 协议缺陷，发送大量伪造的 TCP 连接请求，使被攻击方资源耗尽（CPU 满负荷或内存不足）的攻击方式。

常见的防止 DDoS 的方式：

- **TCP 首包丢弃方案。**利用 TCP 协议的重传机制识别正常用户和攻击报文。当防御设备接到一个 IP 地址的 SYN 报文后，简单比对该 IP 是否存在于白名单中，存在则转发到后端。如不存在于白名单中，检查是否是该 IP 在一定时间段内的首次 SYN 报文，不是则检查是否重传报文，是重传则转发并加入白名单，不是则丢弃并加入黑名单。是首次 SYN 报文则丢弃并等待一段时间以试图接收该 IP 的 SYN 重传报文，等待超时则判定为攻击报文并加入黑名单。
- **缓存。**尽量由设备的缓存直接返回结果来保护后端业务。大型的互联网企业会有庞大的 CDN 节点缓存内容。
- **重发。**可以是直接丢弃 DNS 报文导致 UDP 层面的请求重发，也可以是返回特殊响应强制要求客户端使用 TCP 协议重发 DNS 查询请求。
- **判断博文内容。**一个 TCP 连接中，HTTP 报文太少和报文太多都是不正常的，过少可能是慢速连接攻击，过多可能是使用 HTTP 1.1 协议进行的 HTTP Flood 攻击。

4. 访问控制的模型

开发者需要在他们的软件和设备中实现访问控制功能，访问控制模型为之提供了模型。常见的访问控制的模型有三种。

- **自主访问控制（Discretionary Access Control，DAC）模型**：管理的方式不同形成不同的访问控制方式。一种方式是由客体的属主对自己的客体进行管理，由属主自己决定是否将自己客体的访问权或部分访问权授予其他主体，这种控制方式是自主的，我们把它称为自主访问控制。在自主访问控制下，一个用户可以自主选择哪些用户可以共享他的文件。Linux 系统中有两种自主访问控制策略，一种是 9 位权限码（User-Group-Other），另一种是访问控制列表 ACL（Access Control List）。
- **强制访问控制（Mandatory Access Control，MAC）模型**：用于将系统中的信息分密级和类进行管理，以保证每个用户只能访问到那些被标明可以由他访问的信息的一种访问约束机制。通俗地来说，在强制访问控制下，用户（或其他主体）与文件（或其他客体）都被标记了固定的安全属性（如安全级、访问权限等），在每次访问发生时，系统检测安全属性以便确定一个用户是否有权访问该文件。其中多级安全（MultiLevel Secure，MLS）就是一种强制访问控制策略。
- **基于角色的访问控制模型（Role Based Access Control，RBAC）模型**：管理员定义一系列角色（roles）并把它们赋予主体。系统进程和普通用户可能有不同的角色。设置对

象为某个类型，主体具有相应的角色就可以访问它。这样就把管理员从定义每个用户的许可权限的繁冗工作中解放出来。RBAC 有时称为基于规则的、基于角色的访问控制（Rule-Based Role-Based Access Control，RB-RBAC）。它包含了根据主体的属性和策略定义的规则动态地赋予主体角色的机制。例如，你是一个网络中的主体，你想访问另一个网络中的对象。这个网络定义好了访问列表的路由器的另一端，路由器根据你的网络地址或协议，赋予你某个角色，这决定了你是否被授权访问。

1.8　并发

计算机用户想当然地认为他们的系统在一个时间内可以做多件事。比如，用户一边用浏览器下载视频文件，一边可以继续在浏览器上浏览网页。可以做这样的事情的软件称为并发软件（concurrent software）。

计算机实现多个程序的同时执行，主要基于以下原因：

- **资源利用率。**某些情况下，程序必须要等待其他外部的某个操作完成，才能往下继续执行，而在等待的过程中，该程序无法执行其他任何工作。因此，如果在等待的同时可以运行另外一个程序，将无疑提高资源的利用率。
- **公平性。**不同的用户和程序对于计算机上的资源有着同等的使用权。一种高效的运行方式是通过粗粒度的时间分片（Time Slicing）使得这些用户和程序能共享计算机资源，而不是一个程序从头运行到尾，然后再启动下一个程序。
- **便利性。**通常来说，在计算多个任务时，应该编写多个程序，每个程序执行一个任务并在必要时相互通信，这比只编写一个程序来计算所有任务更加容易实现。

1.8.1　线程与并发

早期的分时系统中，每个进程以串行化方式执行指令，并通过一组 I/O 指令来与外部设备通信。每条被执行的指令都有相应的“下一条指令”，程序中的控制流就是按照指令集的规则来确定的。

串行编程模型的优势是直观性和简单性，因为它模仿了人类的工作方式：每次只做一件事，做完再做其他事情。例如，早上起床后，先穿衣，然后下楼，吃早饭。在编程语言中，这些现实世界的动作可以进一步被抽象为一组粒度更细的动作。例如，喝茶的动作可以被细化为：打开橱柜，挑选茶叶，将茶叶倒入杯中，查看茶壶的水是否够，不够要加水，将茶壶放在火炉上，点燃火炉，然后等水烧开，等等。在等水烧开这个过程中包含了一定程序的异步性。例如，在烧水过程中，你可以干等，也可以做其他事情，比如开始烤面包，或者看报纸（这就是另一个

异步任务），同时留意水是否烧开了。但凡做事高效的人，总能在串行性和异步性之间找到合理的平衡，程序也是如此。

线程允许在同一个进程中同时存在多个线程控制流。线程会共享进程范围内的资源，例如内存句柄和文件句柄，但每个线程都有各自的程序计数器、栈以及局部变量。线程还提供了一种直观的分解模式来充分利用操作系统中的硬件并行性，而在同一个程序中的多个线程也可以被同时调度到多个 CPU 上运行。

毫无疑问，多线程编程使得程序任务并发成为了可能。而并发控制主要是为了解决多个线程之间资源争夺等问题。并发一般发生在数据聚合的地方，只要有聚合，就有争夺发生，传统解决争夺的方式采取线程锁机制，这是强行对 CPU 管理线程进行人为干预，线程唤醒成本高，新的无锁并发策略来源于异步编程、非阻塞 I/O 等编程模型。

1.8.2 并发与并行

《The Practice of Programming》一书的作者 Rob Pike 对并发与并行做了如下描述：

> 并发是同一时间应对（dealing with）多件事情的能力；并行是同一时间动手做（doing）多件事情的能力。

并发（concurrency）属于问题域（problem domain），并行（parallelism）属于解决域（solution domain）。并行和并发的区别在于有无状态，并行计算适合无状态应用，而并发解决的是有状态的高性能；有状态要着力解决并发计算，无状态要着力并行计算，云计算要能做到这两种计算自动伸缩扩展。

1.8.3 并发带来的风险

多线程并发会带来如下的问题：

- **安全性问题。**在没有充足同步的情况下，多个线程中的操作执行顺序是不可预测的，甚至会产生奇怪的结果。线程间的通信主要是通过共享访问字段及其字段所引用的对象来实现的。这种形式的通信是非常有效的，但可能导致 2 种错误：线程干扰（thread interference）和内存一致性错误（memory consistency errors）。
- **活跃度问题。**一个并行应用程序的及时执行能力被称为它的活跃度（liveness）。安全性的含义是“永远不发生糟糕的事情”，而活跃度则关注于另外一个目标，即“某件正确的事情最终会发生”。当某个操作无法继续执行下去，就会发生活跃度问题。在串行程序中，活跃度问题形式之一就是无意中造成的无限循环（死循环）。而在多线程程序中，常见的活跃度问题主要有死锁、饥饿以及活锁。

- **性能问题。**在设计良好的并发应用程序中，线程能提升程序的性能，但无论如何，线程总是带来某种程度的运行时开销。而这种开销主要是在线程调度器临时关起活跃线程并转而运行另外一个线程的上下文切换操作（Context Switch）上，因为执行上下文切换，需要保存和恢复执行上下文，丢失局部性，并且 CPU 时间将更多地花在线程调度而不线程运行上。当线程共享数据时，必须使用同步机制，而这些机制往往会抑制某些编译器优化，使内存缓存区中的数据无效，以及增加贡献内存总线的同步流量。所以这些因素都会带来额外的性能开销。

1. 死锁（Deadlock）

死锁是指两个或两个以上的线程永远被阻塞，一直等待对方的资源。

下面是一个 Java 编写的死锁的例子。

Alphonse 和 Gaston 是朋友，都很有礼貌。礼貌的一个严格的规则是，当你给一个朋友鞠躬时，你必须保持鞠躬，直到你的朋友鞠躬回给你。不幸的是，这条规则有个缺陷，那就是如果两个朋友同一时间向对方鞠躬，那就永远不会完了。这个示例应用程序中，死锁模型是这样的：

```
public class Deadlock {
    static class Friend {
        private final String name;

        public Friend(String name) {
            this.name = name;
        }

        public String getName() {
            return this.name;
        }

        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s" + "  has bowed to me!%n", this.name,
bower.getName());
            bower.bowBack(this);
        }

        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s" + " has bowed back to me!%n", this.name,
bower.getName());
        }
```

```
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() {
                alphonse.bow(gaston);
            }
        }).start();
        new Thread(new Runnable() {
            public void run() {
                gaston.bow(alphonse);
            }
        }).start();
    }
}
```

当它们尝试调用 bowBack 时两个线程将被阻塞。无论是哪个线程，也永远不会结束，因为每个线程都在等待对方鞠躬。这就是死锁了。

上面例子的代码可以在 https://github.com/waylau/distributed-systems-technologies-and-cases-analysis 的 distributed-systems-java-demos 程序的 com.waylau.essentialjava.concurrency 包下找到。

2. 饥饿（Starvation）

饥饿描述了一个线程由于访问足够的共享资源而不能执行程序的现象。这种情况一般出现在共享资源被某些“贪婪”线程占用，而导致资源长时间不被其他线程可用。例如，假设一个对象提供一个同步的方法，往往需要很长时间返回。如果一个线程频繁调用该方法，其他线程若也需要频繁地同步访问同一个对象则通常会被阻塞。

3. 活锁（Livelock）

一个线程常常处于响应另一个线程的动作，如果其他线程也常常响应该线程的动作，那么就可能出现活锁。与死锁的线程一样，程序无法进一步执行。然而，线程是不会阻塞的，它们只是会忙于应对彼此的恢复工作。现实中的例子是，两人面对面试图通过一条走廊：Alphonse 移动到他的左侧给 Gaston 让路，而 Gaston 移动到他的右侧想让 Alphonse 过去，两个人同时让路，但其实两人都挡住了对方，他们仍然彼此阻塞。

下面就介绍几种解决并发问题的常用方法。

1.8.4　同步（Synchronization）

同步是避免线程干扰和内存一致性错误的常用手段。下面就用 Java 代码来演示这几种问题，以及如何用同步解决这类问题。

1. 线程干扰

下面描述当多个线程访问共享数据时错误是如何出现的。

考虑下面的一个简单的类 Counter：

```
public class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

其中的 increment 方法用来对 c 加 1；decrement 方法用来对 c 减 1。然而，多个线程中都存在对某个 Counter 对象的引用，那么线程间的干扰就可能导致出现我们不想要的结果。

线程间的干扰出现在多个线程对同一个数据进行多个操作的时候，也就是出现了“交错”。这就意味着操作是由多个步骤构成的，而此时，在这多个步骤的执行上出现了叠加。

Counter 类对象的操作貌似不可能出现这种“交错（interleave）”，因为其中的两个关于 c 的操作都很简单，只有一条语句。然而，即使是一条语句也会被虚拟机翻译成多个步骤。在这里，我们不深究虚拟机具体将上面的操作翻译成了什么样的步骤。只需要知道即使简单的 C++这样的表达式也会被翻译成三个步骤：

（1）获取 c 的当前值。

（2）对其当前值加 1。

（3）将增加后的值存储到 c 中。

表达式 c--也会被按照同样的方式进行翻译，只不过第二步变成了减 1，而不是加 1。

假定线程 A 中调用 increment 方法，线程 B 中调用 decrement 方法，而调用时间基本上相同。如果 c 的初始值为 0，那么这两个操作的“交错”顺序可能如下所示。

（1）线程 A：获取 c 的值。

（2）线程 B：获取 c 的值。

（3）线程 A：对获取到的值加 1，其结果是 1。

（4）线程 B：对获取到的值减 1，其结果是-1。

（5）线程 A：将结果存储到 c 中，此时 c 的值是 1。

（6）线程 B：将结果存储到 c 中，此时 c 的值是-1。

这样线程 A 计算的值就丢失了，也就是被线程 B 的值覆盖了。上面的这种“交错”只是其中的一种可能性。在不同的系统环境中，有可能是 B 线程的结果丢失了，或者是根本就不会出现错误。由于这种“交错”是不可预测的，线程间相互干扰造成的 bug 是很难定位和修改的。

2. 内存一致性错误

下面介绍通过共享内存出现的不一致的错误。

内存一致性错误发生在不同线程对同一数据产生不同的“看法”。导致内存一致性错误的原因很复杂，超出了本书的描述范围。庆幸的是，程序员并不需要知道出现这些原因的细节。我们需要的是一种可以避免这种错误的方法。

避免出现内存一致性错误的关键在于理解 happens-before 关系。这种关系是一种简单的方法，能够确保一条语句对内存的写操作对于其他特定的语句都是可见的。为了理解这点，我们可以考虑如下的示例。假设定义了一个简单的 int 类型的字段并对其进行初始化：

```
int counter = 0;
```

该字段由两个线程共享：A 和 B。假定线程 A 对 counter 进行了自增操作：

```
counter++;
```

然后，线程 B 打印 counter 的值：

```
System.out.println(counter);
```

如果以上两条语句是在同一个线程中执行的，那么输出的结果自然是 1。但是如果这两条语句是在两个不同的线程中，那么输出的结构有可能是 0。这是因为没有保证线程 A 对 counter 的修改对线程 B 来说是可见的。除非程序员在这两条语句间建立了一定的 happens-before 关系。

我们可以采取多种方式建立这种 happens-before 关系。使用同步就是其中之一，这点我们

将会在下面的小节中看到。

到目前为止，我们已经看到了两种建立这种 happens-before 的方式：

- 当一条语句中调用了 Thread.start 方法，那么每一条和该语句已经建立了 happens-before 的语句都和新线程中的每一条语句有这种 happens-before。引入并创建这个新线程的代码产生的结果对该新线程来说都是可见的。
- 当一个线程终止了并导致另外的线程中调用 Thread.join 的语句返回，那么此时这个终止了的线程中执行了的所有语句都与随后的 join 语句的所有语句建立了这种 happens-before。也就是说，终止了的线程中的代码效果对调用 join 方法的线程来说是可见的。

关于哪些操作可以建立这种 happens-before，更多的信息请参阅“java.util.concurrent 包的概要说明”（https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility）。

3. 同步方法

Java 编程语言中提供了两种基本的同步用语：同步方法（synchronized methods）和同步语句（synchronized statements）。同步语句相对而言更为复杂一些，我们将在下一小节中进行描述。本节重点讨论同步方法。

我们只需要在声明方法的时候增加关键字 synchronized 即可：

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

如果 count 是 SynchronizedCounter 类的实例，设置其方法为同步方法会有两个效果：

- 首先，不可能出现对同一对象的同步方法的两个调用的“交错”。当一个线程在执行一

个对象的同步方式的时候，其他所有调用该对象的同步方法的线程都会被挂起，直到第一个线程对该对象操作完毕。

- 其次，当一个同步方法退出时，会自动与该对象的同步方法的后续调用建立 happens-before 关系。这就确保了对该对象的修改对其他线程是可见的。

注意：构造函数不能是 synchronized——在构造函数前使用 synchronized 关键字将导致语义错误。同步构造函数是没有意义的。这是因为只有创建该对象的线程才能调用其构造函数。

警告：在创建多个线程共享的对象时，要特别小心对该对象的引用不能过早地“泄露”。例如，假定我们想要维护一个保存类的所有实例的列表 instances。我们可能会在构造函数中这样写到：

```
instances.add(this);
```

但是，其他线程可会在该对象的构造完成之前就访问该对象。

同步方法是一种简单的可以避免线程相互干扰和内存一致性错误的策略：如果一个对象对多个线程都是可见的，那么所有对该对象的变量的读写都应该是通过同步方法完成的（一个例外就是 final 字段，它在对象创建完成后是不能被修改的，因此，在对象创建完毕后，可以通过非同步的方法对其进行安全地读取）。这种策略是有效的，但是可能导致“活跃度问题”。这点我们会在后面进行描述。

4. 内部锁和同步

同步是构建在被称为“内部锁（intrinsic lock）”或者是“监视锁（monitor lock）”的内部实体上的。在 API 中通常被称为“监视器（monitor）”。内部锁在两个方面都扮演着重要的角色：保证对对象状态访问的排他性，建立对象可见性相关的 happens-before 关系。

每一个对象都有一个与之相关联动的内部锁。按照传统的做法，当一个线程需要对一个对象的字段进行排他性访问并保持访问的一致性时，它必须在访问前先获取该对象的内部锁，然后才能访问之，最后释放该内部锁。在线程获取对象的内部锁到释放对象的内部锁的这段时间，我们说该线程拥有该对象的内部锁。只要有一个线程已经拥有了一个内部锁，其他线程就不能再拥有该锁了。其他线程在试图获取该锁的时候被阻塞了。

当一个线程释放了一个内部锁，那么就会建立起该动作和后续获取该锁之间的 happens-before 关系。

5. 同步方法中的锁

当一个线程调用一个同步方法的时候，它就自动地获得了该方法所属对象的内部锁，并在方法返回的时候释放该锁。即使由于出现了没有被捕获的异常而导致方法返回，该锁也会被释放。

我们可能会感到疑惑：当调用一个静态的同步方法的时候会怎样？静态方法是和类相关的，而不是和对象相关的。在这种情况下，线程获取的是该类的类对象的内部锁。这样对于静态字段的方法来说，这是由和类的实例的锁相区别的另外的一个锁来进行操作的。

6. 同步语句

另外一种创建同步代码的方式就是使用同步语句。和同步方法不同，使用同步语句必须指明要使用哪个对象的内部锁：

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

在上面的示例中，方法 addName 需要对 lastName 和 nameCount 的修改进行同步，还要避免同步调用其他对象的方法（在同步代码段中调用其他对象的方法可能导致“活跃度”中描述的问题）。如果没有使用同步语句，那么将不得不使用一个单独、未同步的方法来完成对 nameList.add 的调用。

在改善并发性时，巧妙地使用同步语句能起到很大的帮助作用。例如，我们假定类 MsLunch 有两个实例字段，c1 和 c2，这两个变量绝不会一起使用。所有对这两个变量的更新都需要进行同步。但是没有理由阻止对 c1 的更新和对 c2 的更新出现交错——这样做会创建不必要的阻塞，进而降低并发性。此时，我们没有使用同步方法或者使用和 this 相关的锁，而是创建了两个单独的对象来提供锁。

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }
```

```
    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

采用这种方式时需要特别小心，我们必须绝对确保相关字段的访问交错是完全安全的。

7. 重入同步（Reentrant Synchronization）

回忆前面提到的：线程不能获取已经被别的线程获取的锁。但是线程可以获取自身已经拥有的锁。允许一个线程能重复获得同一个锁就称为重入同步（reentrant synchronization）。它是这样的一种情况：在同步代码中直接或者间接地调用了还有同步代码的方法，两个同步代码段中使用的是同一个锁。如果没有重入同步，在编写同步代码时需要额外小心，以避免线程将自己阻塞。

1.8.5 原子访问（Atomic Access）

下面介绍另外一种可以避免被其他线程干扰的做法的总体思路——原子访问。

在编程中，原子性动作就是指一次性有效完成的动作。原子性动作是不能在中间停止的：要么一次性完全执行完毕，要么就不执行。在动作没有执行完毕之前，是不会产生可见结果的。

通过前面的示例，我们已经发现了诸如 C++这样的自增表达式并不属于原子操作。即使是非常简单的表达式也包含了复杂的动作，这些动作可以被解释成许多别的动作。然而，的确存在一些原子操作：

- 对几乎所有的原生数据类型变量（除了 long 和 double）的读写以及引用变量的读写都是原子的。
- 对所有声明为 volatile 的变量的读写都是原子的，包括 long 和 double 类型。

原子性动作是不会出现交错的，因此，使用这些原子性动作时不用考虑线程间的干扰。然而，这并不意味着可以移除对原子操作的同步。因为内存一致性错误还是有可能出现的。使用 volatile 变量可以降低内存一致性错误的风险，因为任何对 volatile 变量的写操作都和后续对该变量的读操作建立了 happens-before 关系。这就意味着对 volatile 类型变量的修改对于别的线程来说是可见的。更重要的是，这意味着当一个线程读取一个 volatile 类型的变量时，它看到的不仅仅是对该变量的最后一次修改，还看到了导致这种修改的代码带来的其他影响。

使用简单的原子变量访问比通过同步代码来访问变量更高效，但是需要程序员更多细心的考

虑，以避免内存一致性错误。这种额外的付出是否值得完全取决于应用程序的大小和复杂度。