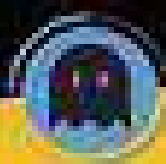


涵盖网络游戏整体架构，包括线程、Actor模型、对象回收机制与组件式编程



多人在线游戏 架构实战

基于C++的分布式游戏编程

彭 越 编著

〈 近 **50** 个真实案例 〉

〈 近 **80** 个流程图 〉

〈 完整的
客户端与服务端实例 〉

机械工业出版社
China Machine Press



掌握网络游戏整体架构，包括线程、Actor模型、对象回收机制与组件式编程



多人在线游戏 架构实战

基于C++的分布式游戏编程

彭欣 编著

〈近50个真实案例〉

〈近80个流程图〉

〈完整的
客户端与服务端实例〉



机械工业出版社
China Machine Press

版权信息

COPYRIGHT INFORMATION

书名：多人在线游戏架构实战：基于C++的分布式游戏编程

作者：彭放

排版：zm

出版社：机械工业出版社

ISBN：9787111667926

— · 版权所有 侵权必究 · —

前言

本书主要讲述大型多人在线游戏开发的框架与编程实践，以实际例子来介绍从无到有地制作网络游戏框架的完整过程，让读者了解网络游戏制作中的所有细节，最终我们会得到一个完整的、基于ECS（Entity Component System，实体组件系统）模式的、高效的分布式服务端框架，一个可以登录的客户端，以及一个用于测试的机器人工程。本书为读者呈现了近50个真实示例，提供了近80个流程图，以直观的方式还原游戏制作的过程，同时阐述在这些示例中运用的编程技巧、数据结构以及所采用的主流设计模式。

笔者从事游戏开发工作十余年，经历了从PC端游、网页游戏到手机游戏3个重要的游戏发展阶段。从行业知名大公司到创业团队，笔者曾就职于数个多元化的游戏制作团队，参与研发过多种类型的游戏，其间积累了相当丰富的游戏编程经验，本书就是总结这些经验编写而成的。

不同于其他的游戏编程图书，本书不使用伪代码，而是给出游戏制作中的具体实现。

本书致力于向游戏开发编程人员、学生或对游戏编程有兴趣的读者提供一整套游戏开发的基础框架——一种多进程、多线程、高效的分布式服务端解决方案，在这个基础框架之上，我们可以快速开发游戏业务逻辑。同时，本书的源代码库中提供了一套易于开发的、少耦合的客户端框架，用于验证分布式框架的功能。

本书讲解的是网络游戏框架，定位为多人在线游戏体系，这类游戏的特点是数以万计的玩家同时在游戏世界中，游戏互动性很高。对多人在线游戏进行细分，其中两个大的分类为MMO（Massively Multiplayer Online，大型多人在线）游戏和MOBA（Multiplayer Online Battle Arena，多人在线竞技类）游戏。如今，MMO和MOBA游戏基本占据了游戏

市场的半壁江山。

常见的MMO游戏是MMORPG（Massively Multiplayer Online Role-Playing Game，大型多人在线角色扮演游戏），代表作有《魔兽世界》，其玩法是玩家登录一个虚拟的游戏世界，进行任务、道具、交易等一系列交互操作。

大家熟悉的MOBA游戏有《英雄联盟》《王者荣耀》等，其玩法是玩家之间的竞技操作。

这两类游戏都是游戏中比较复杂的类型，也是本书中框架设计的目标类型。但不要被吓到，本书不但适合有经验的游戏行业从业人员学习，同样适合新手学习。如果读者有游戏行业从业的经验，那么应该了解游戏工作室的法则，即为了加快开发进度，游戏从业人员总是被固定在某一个领域或者某一系统开发中，例如游戏技能开发、AI功能开发。越是大公司，越遵循这条法则，因此游戏从业人员不得不根据策划提出来的需求，花大量的时间在某一区域中增加功能、修改Bug，从而忽视了游戏的整体框架。又或许出于代码安全考虑，公司不得不对一些员工屏蔽核心的功能，使得一线的游戏编程人员在整体框架上的知识储备总是缺少一环。

本书通过实战的方式向读者展示如何搭建一个既适合MMO又适合MOBA的基本游戏框架。虽然这两类游戏在逻辑上大相径庭，但在架构的时候仍有相通之处。

如果读者要了解本书最终的成果，可以预览介绍视频，笔者将介绍视频的地址放在了源代码目录中。

本书组织结构

本书共12章，各章内容介绍如下：

第1、2章主要介绍网络编程。网络编程是网络游戏的核心，毫不夸张地说，网络游戏是一种建立在协议之上的软件。由于系统底层不同，因此示例中会同时介绍基于Windows系统与基于Linux系统的不同网络编程模型。第2章会引入Google公司开发的第三方protobuf库来定义网络协议的内容，该库是目前常用的协议工具。

第3章讲解进程和线程，它们是构成后续逻辑的基础知识。同时，引入Actor模型并介绍该模型的使用方法。为了充分了解这些知识点，第4章给出了一个登录示例。在这一章中，我们的高性能服务框架会有一个基本雏形——由几个多线程的进程组成，可以自由地扩展进程，能均衡服务端的性能。这里需要澄清一个概念，本书所说的服务端并不一定是指物理机，可能是几台物理机，也可能是一台物理机上的几个进程相互协作的系统。

第5章讲解性能优化与对象池。在开始更复杂的编码之前，有必要对框架的性能做一次检查。在这一章中介绍Windows和Linux下的性能检查工具。在检查的过程中会引入一些提升性能的常用缓存数据结构和对象池来提升整体框架的效率。

第6章搭建ECS框架。我们要在旧的框架体系上做一些改动，引入ECS模式。在这一章中将详细介绍什么是ECS、它有什么特点以及如何让它在多线程上无冲突地运行。在此基础之上，第7章引入MySQL数据库，存储功能是游戏中的基础功能之一。

第8章深入学习组件式编程，基于ECS的思路，将抛弃面向对象的编程，真正转向组件式编程。为了便于读者理解，在这一章中制作了非常多的流程图，以梳理组件与框架是如何配合工作的。ECS和Actor模型的结合会让框架变得异常灵活。其表现在于，框架可以随时合并成一个进程以方便在开发阶段进行测试，也可以随着发布的需求变更为无数个进程，以便在必要时部署到多个物理机上。

基本框架搭建完成之后，第9~12章都是上层应用。在这几章中给出了分布式登录方案、跳转方案，还引入了内存数据库Redis。第9~11章将深入使用框架，让本书中的框架实现更多的功能，使之与真正意义上的游戏接近。最后一章将学习如何在框架的基础上创建一个自己的系统——移动系统，以及游戏人物是如何行走的。

正文虽然只有12章，但作为实战类图书，每章示例工程中的代码量都相当大，限于篇幅，本书只讨论重点代码与逻辑。游戏制作是一个非常复杂的过程，但是如果了解了基本思路，这又是一个极为简单的过程。

除了正文之外，笔者还准备了一个附录（见下载资源）。附录在源代码目录下，用于介绍Linux基础，包括如何在Windows上安装一个

Linux虚拟机，如何在其上搭建环境、编译代码，以及如何安装需要的工具与数据库。附录的内容与编码没有任何关系，但是如果不熟悉的话，那么很有可能运行不了本书提供的代码，建议开始学习正文内容之前阅读一下附录内容。当然，如果读者已经掌握了这些知识点，那么可以跳过。

本书重点放在服务端的框架上，每一章都会给出机器人工程对代码进行验证，在这个过程中，我们逐步使代码趋近于最终的框架。本书编写了很多示例，这些示例的一次又一次优化就是编程思路的一次又一次明晰，在这个过程中读者可以更加深入地了解编码背后的逻辑。

本书有近50个示例，前面的示例相对简单，后面的示例会越来越复杂，通过慢慢地增加功能，让读者能跟得上进度。为了突出重点，不会讲解每一行代码的作用，只挑出一些重要功能进行详细的代码解析和流程分析。

最后要说明的是，复制代码是编程学习中很大的弊病，虽然解决了学习者暂时的困难，但是编程思想是无法复制的，所以本书希望达到这样一个目的——告诉读者为什么这样编写代码。

学习是为了找到答案，更是为了学到思维方式。

本书涉及的编程语言

在游戏制作中，我们常用的编程语言是C++、C#和Java，脚本语言是Lua和Python。当然，这不是定式，用什么语言来制作游戏是根据当下的需求来确定的，有时可能不得不用两种或者两种以上的语言来完成整个游戏的制作。

客户端的常用引擎是Unity和UE。Unity的脚本语言为JavaScript和C#，UE采用了C++。值得一提的是，Unity和UE都具有跨平台性，也就是说，引擎已经为我们做好了输出不同平台的不同文件的准备，编程人员只需要专注于游戏实现。

为了提高效率，在服务端选择了C++这样重量级的语言，它既有高级语言的特性，又可以实现一些底层功能。对于游戏服务端，C++依然是主流选择。虽然比起Go和C#等相对容易的高级语言C++更难一些，指针使用不当会不够安全，但是C++有一个特性——灵活，灵活即自由。

在使用C++的时候，我们并不像传统那样需要造轮子，Linux开源社区有很多开源的库，需要时可以随时下载使用。不要被C++吓到，可以将重点放在思维上，而语言不过是我们使用的手段。

本书编写的服务端程序对于Windows系统与Linux系统是兼容的，虽然使用了Linux的开源库，但是也提供了跨平台编译它们的方案，在使用时相关章节给出了教程。

一般来说，最终的服务端程序会在Linux机器上运行，但对于不熟悉Linux的编程人员来说，Windows系统的界面以及Visual Studio工具可能更让人觉得亲切，鉴于此，本书中所有的工程都做了系统的兼容，以方便读者学习。

本书采用了C++14标准。虽然本书不会重点讲解语言特性，但C++14部分的代码不会成为读者阅读的障碍，在有特别语义的地方都会做出解释。我们将重点放在框架、编程思想和数据结构上，只是采用C++来实现这些思想与数据结构。

客户端选择容易上手的Unity，它的跨平台性以及可视化的综合开发环境使其在图形引擎中脱颖而出，成为近几年游戏客户端的主流开发工具。在本书的示例代码中，选择C#作为脚本语言。出于篇幅和侧重点的考虑，本书不会对客户端的编程思路做出阐述，但依然给出完整的客户端源代码。

客户端的资源（如场景地图、人物、动作）都来自九众互动公司，本书已获得该公司的授权，在此表示感谢。读者可以自由下载使用，但不可用于商业用途。

游戏的编码过程很复杂，涉及的功能和外围系统也很多，可能会同时使用多种语言。掌握了编程思想之后，语言之间是有共性的，学起来也很快，所以，即使本书涉及两种语言，也不必过于担心。

我们在选择框架或者学习语言的时候，总是很纠结该选什么样的语言。语言的选择一直是编程社区讨论的重点。对于游戏服务器实现语言的选择来说，有的采用C++、Java、C#、Go，也有完全基于Python的。每种语言各有利弊，但如果认真分析过这些代码，那么可以发现它们的逻辑与数据结构是大同小异的。真正重要的不是语言本身，我们遇到的每一个问题，采用任何一种框架、任何一种语言都有解决方案，可见重

点在于编程思想与数据结构。

源代码与工程目录结构

本书有两个源代码下载地址：

- (1) 服务器：<https://github.com/setuppf/GameBookServer.git>。
- (2) 客户端：<https://github.com/setuppf/GameBookClient.git>。

可以使用SVN或Git方式获取。如果是在Windows系统下，那么可以使用Git或者SVN工具获取源代码，使用Visual Studio打开工程；如果是在Linux系统下，那么可以直接使用Git或者SVN命令行获取源代码。

另外，还可以登录机械工业出版社华章公司的网站（www.hzbook.com）下载，首先搜索到本书，然后在页面上的“资源下载”模块下载即可。如果在下载过程中遇到问题，请发送电子邮件到booksaga@126.com，邮件主题为“多人在线游戏架构实战：基于C++的分布式游戏编程”。

本书所有源代码在CentOS 7上的GCC 9.1.0版本下测试通过。在Windows系统下，采用Visual Studio 2019，在Debug x86下测试通过，Windows系统下的Release版本没有进行相应配置，读者可根据Debug参数自行配置。注意，在Windows系统下只能编译为x86，原因是我们使用的Redis第三方库hiredis目前不支持64位Windows。但这无伤大雅，因为最终的代码是运行在Linux上的。

本书的每一个示例代码都经过了调试，消除了基本可见的Bug，但仍免不了有未知Bug，敬请谅解。

为了便于理解，所有的工程采用了相同的目录结构，清晰的目录结构能够使读者对于架构有更清晰的认识。我们为所有的工程标记了编号，目录对应的功能说明如下：

(1) 一般示例目录，例如01_01_network_first表示第1章的第1个工程例子。

(2) include目录，该目录是所有工程需要的头文件目录。因为不同操作系统要求包含的头文件不同，所以分成了common目录和windows

目录。common目录是两个操作系统共用的头文件目录。现在不必困扰这些头文件有什么作用、什么时候使用，在相关的章节会一一说明。

(3) libs目录，该目录是所有工程需要的库目录，区分了Windows和Linux版本。本书预先将所有需要的第三方库都编译好了。Windows版本是由Visual Studio 2019 Debug版本编译出来的，如果读者使用的不是这个版本，那么可能需要重新编译。Linux版本则是由CentOS 7下的GCC 9.1版本编译出来的。如果读者使用的编译工具与本书的工具不同，也不需要太担心，在使用这些库的时候会给出编译步骤。

(4) linux目录，该目录中的内容是Linux系统下需要的一些脚本。

(5) tools目录，该目录中的内容是需要用到的工具，是Windows系统下编译出来的protoc.exe文件和一些数据库脚本。

以上是对源代码目录的说明。对每个工程而言，目录结构都是固定的。在Linux系统下，进入目录，执行tree命令，就可以看到目录结构。如果没有tree工具，就执行安装命令“yum install-y tree”。进入最后一个示例目录，执行“tree-d”命令，目录结构如下。

```
└── bin                                // 编译后的可执行文件可以在这个目录下找到
└── res                                // 资源目录，存放着配置文件
    ├── log4
    └── resource
└── src                                // 源代码目录，这个目录由 3 个小目录组成
    ├── apps                            // 进程目录，每个进程一个目录
    │   ├── allinone
    │   ├── login
    │   .....
    ├── libs                            // 库目录
    │   ├── libplayer
    │   .....
    │   └── libserver
    │       └── protobuf
    └── tools                            // 工具目录
        └── robots
```


将这个工程与第2章中的工程对比，会发现其目录结构是一致的，只是复杂度不同。

第1章

网络编程基础

本章主要介绍网络编程的基础知识，单机游戏与网络游戏的本质区别在于有了网络层的介入，有了异步。本章的重点在于网络编程的基本概念和网络底层函数的使用。本章包括以下内容：

- 介绍网络编程的基本概念。
- 给出一个阻塞式的网络示例。
- 给出一个非阻塞式的网络示例。

单机游戏与网络游戏的区别

要了解网络游戏，需先从单机游戏出发，这里讨论的单机游戏是指PC端的游戏，而非XBOX这类游戏机上的游戏。

就国产游戏来说，1995年推出《仙剑奇侠传》之后，随后的几年，《仙剑奇侠传》系列、《轩辕剑》系列一度盛行，在2000年左右，单机游戏发展至巅峰。2001年1月，华义推出了网络游戏《石器时代》，直到2006年左右，巨人网络推出《征途》，第九城市代理的《魔兽世界》公测，将网络游戏推至巅峰，单机游戏才慢慢淡出市场。这一系列的单机游戏都有一个特点：没有服务器的概念，玩家不需要与别的玩家进行交互，所有数据、算法、存储都在本地完成。所以，在单机游戏的早期，很多游戏修改器应运而生，这些游戏修改器要么修改本地的存储文件，要么修改内存数据，以达到“无限金币”“超强武力值”等目的。

作为玩家，我们可以体验游戏、打开游戏界面、点击操作、得到反馈、继续游戏。下面从编程的角度来看看单机游戏的设计流程。

单机游戏的设计思路如图1-1所示。在游戏开始时，需要加载一些资源，这些资源包括地图信息、基本的图素、用户界面（UI）等。加载完成之后，用账号登录游戏，每个账号都有其数据，选择角色真正进入游戏。

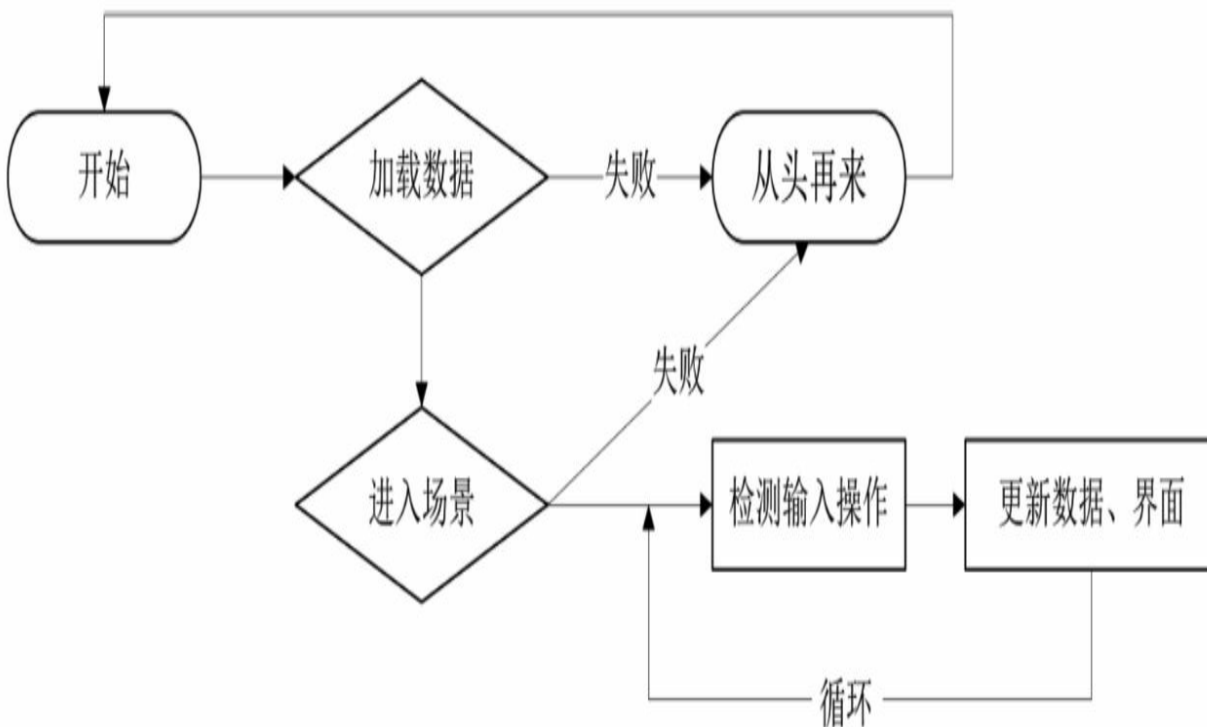


图1-1 单机游戏逻辑

从编码的角度来说，游戏一旦开始，运行的就是一个无限循环（Loop），我们可以认为它是一个while（true）{...}，除非离开游戏，否则这个循环会一直执行下去。循环的每一次执行称为一帧。在一帧中包括3个主要操作：

（1）检测玩家输入。

（2）根据输入更新内存数据，刷新场景、人物模型和界面。

（3）捕捉退出请求，如果有退出请求，这个循环就会被打破，游戏结束。

理论上来讲，一秒能产生的循环数越大，程序的反应就越灵敏。举例说明，假如输入够快，在0.3秒和0.6秒产生了两个输入，例如触发了两个执行技能请求，如果执行一帧需要0.5秒，在0秒进入第一帧，0.5秒之后才会发现0.3秒的输入，而0.6秒的输入则要等到执行第三帧，也就是1秒的时候才会被发现。假如执行一帧只需要0.1秒，则0.3秒和0.6秒的操作会很快被触发。

单机游戏不需要异步的过程，所有操作可以马上得到答案，不需要等待结果，类似于调用函数的过程。调用之后，函数必定马上会有一个返回值。数据都在本地内存中，方便读取，也方便判断。单机游戏创建角色是在本地进行的，不需要向外部请求数据。加载角色选择界面也不需要谁许可，只要能从本地文件中读到角色，就可以马上进入角色选择界面。而网络游戏在这一点上与单机游戏有很大的区别。

了解了单机游戏的流程之后，再来看看网络游戏。

网络游戏和单机游戏一样也有一个循环，只是多了一个网络层的处理。实际上因为异步的关系，逻辑会变得更加复杂，图1-2充分展示了这种不同。下面以登录为例来展示一下具体的不同之处，流程如图1-3所示。

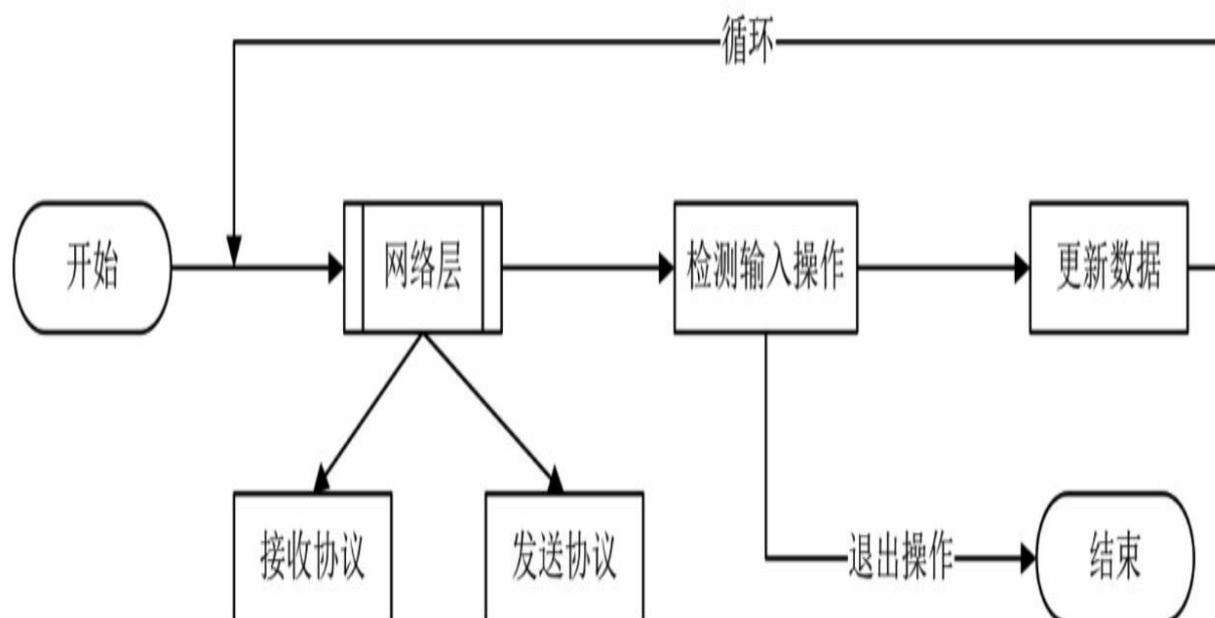


图1-2 网络游戏逻辑

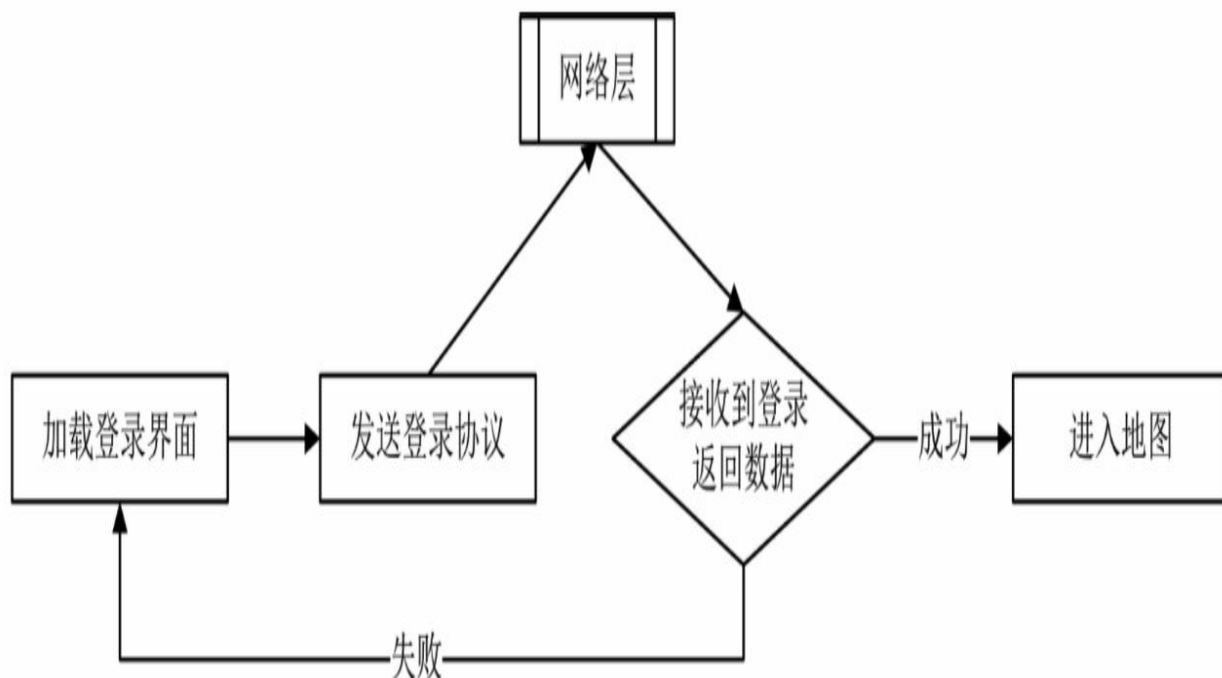


图1-3 客户端登录逻辑

在登录界面输入账号和密码之后，要经历一个异步的操作，客户端向服务端发送协议，等待服务端返回数据，由此来判断登录成功或者失败。网络游戏的客户端发出命令，只有等到服务端给它结果，它才会做出反应。客户端向服务端请求账号验证，这个请求数据不是一个函数可以完成的，这就是一个异步的过程。

第一个需要了解的概念是“异步”。与“异步”一起出现的概念有“同步”“并行”（并发）等，这几个概念往往比较容易混淆。

举一个加载的例子，如果需要加载几个不同的资源，先加载A，等到A加载完成后再加载B，加载完B，再加载C，直到所有资源加载完成，这就是同步操作。以同步方式实现加载，加载者必须等到加载完成之后才能继续后续的加载。

如果用3个线程同时加载A、B、C资源，由于线程之间是不会相互影响的，加载A的同时另外两个线程在加载B和C，这就是并行操作。但这种情况不能算是异步，因为在加载A时，要等待A加载完成之后退出线程。这个等待产生了阻塞。

那么，什么是异步呢？异步不会等待，也不会阻塞。

假设加载一个资源需要10秒，加载3个资源，在串行时需要30秒，在并行时需要10秒，异步也需要10秒。那么，异步与并行的差别在哪里呢？

以图1-4为例来看一下异步的流程。在第N帧发出了加载A命令，这一帧会马上结束，不需要等待10秒，直接进入下一帧，不关心A是否加载成功，当它加载成功之后，会在某一帧收到加载成功的回调。因为是异步，同时加载A、B时，得到结果的顺序可能是不一样的，可能A先发出加载命令，但是B的回调却先发生。

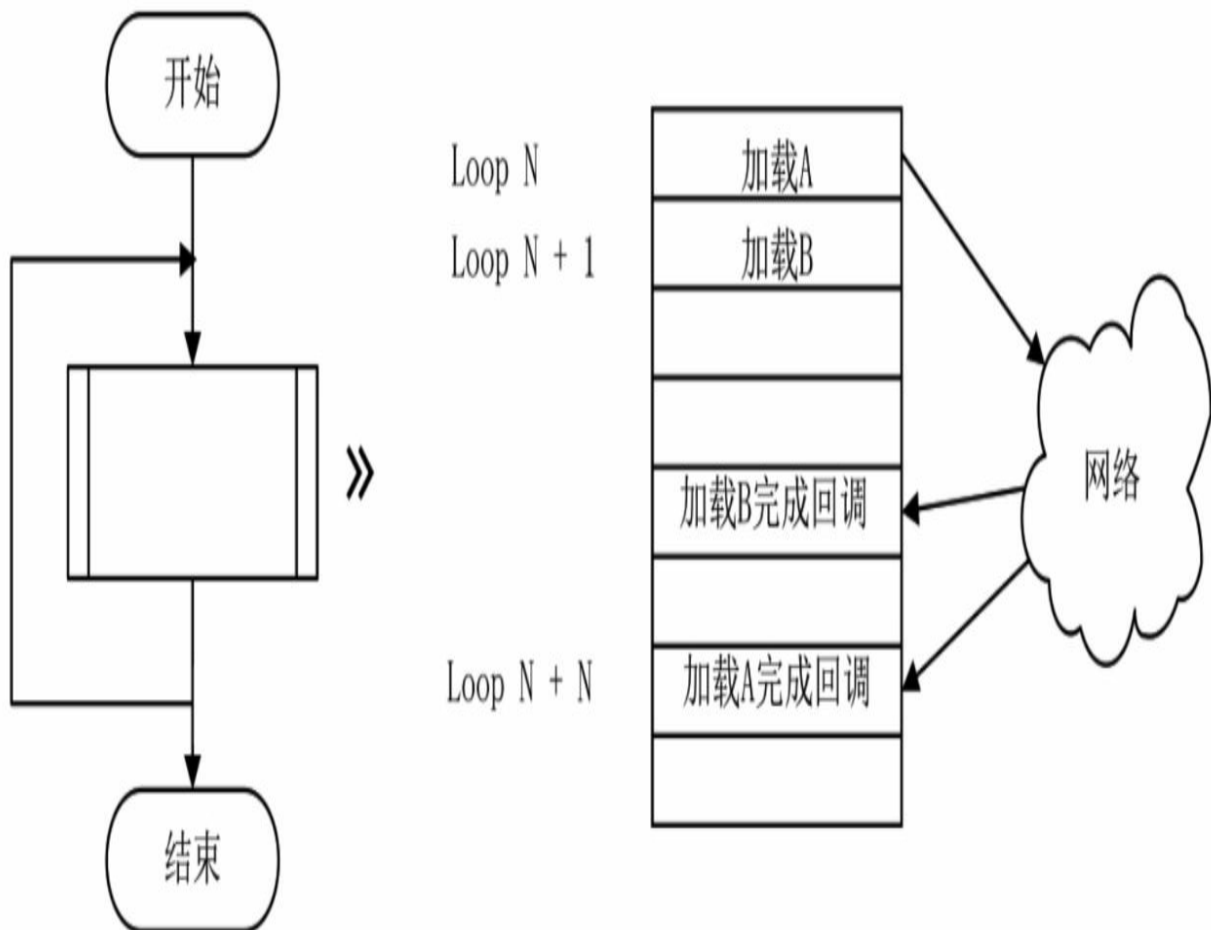


图1-4 异步加载

那么，程序是如何知道回调完成的呢？一般来说，有两种方式可以

知道A是否加载成功：

(1) 主动询问，每间隔一段时间询问A是否加载完成。

(2) 被动接收，一般在启动事件时会要求注册一个回调函数，事件完成时会主动调用回调函数，以标记事件完成。

所谓异步，简而言之，就是在不阻塞进程的情况下完成操作。

网络游戏大部分情况下都是异步的操作。例如，要分解背包中的一个道具。首先要发起一个分解道具的请求，这个请求从客户端发送到服务端，再把结果从服务端传回到客户端。假设它需要0.5秒，采用非异步的方式需要在发起请求之后等0.5秒，等收到服务端的结果再进行后续操作。

如果按这种方式编写游戏代码，效率是非常低下的。在这0.5秒内，进程将暂停在此处，图形引擎得不到刷新，输入操作在这0.5秒内暂停了，也得不到响应。

为了解决这个问题，需要将功能在这里中断。这意味着向服务端发起请求时，这个分解功能就中断了。当分解道具的协议从服务端传回到客户端时，再进行后续的处理。这就是一个异步的操作。

对于编码来说，单机游戏在一个函数中可以完成的操作需要拆分成两步或者更多步。简而言之，单机游戏和网络游戏的一个根本不同就是后者因为网络层介入而产生的异步。接下来将研究网络，包括数据是如何发送的，又是如何接收的。

理解IP地址

在开始真正的网络编程之前，需要理解一些基本概念，IP地址就是其中之一。

在工作和生活中安装和使用WiFi时，经常听人说起IP地址，相信大家一定不会陌生。但它究竟是什么，可能有些读者还不太清楚。IP地址对大众而言就是类似于127.0.0.1这种字符串，但对于编程者来说，我们必须了解这个字符串的本质是一个32位的整数。IP地址具体地确定了一台计算机，它是这台计算机在整个网络中的ID，就像人们使用的身份证一样。

我们看到的IP地址通常是4个十进制数，以“.”隔开，每个十进制数不超过255。

读者想过为什么它不能超过255吗？之前讲过IP地址的本质是一个32位的整数，也就是4字节。为了便于记忆，把每个字节用数字的方式呈现出来，以“.”分隔来显示。也就是说，在127.0.0.1这个串中，每个数值都表示一个字节，一个字节当然不可能超过255。

在使用网络通信的时候，除了IP之外，还需要一个端口。例如，常用的HTTP的默认端口就是80。当我们访问网站的时候，其实就是通过80端口与主机进行通信的。有了IP地址和端口，就可以指定一个确定且唯一的通信链路。

随着互联网的发展，网络变得越来越庞大，之前定义的32位IP地址已经不够用了，所以新的长达128位的IP地址出现了，即IPv6，而旧的32位地址称为IPv4。

理解TCP/IP

在互联网上，任意两台计算机都必须通过网络协议进行通信。游戏中常用的网络通信协议为TCP/IP（Transmission Control Protocol/Internet Protocol）。

协议是什么呢？简单来说就是有格式的数据。A机向B机发送一串字符，这串字符经过层层包装发送到B机，B机根据约定的格式一步步分解成最初的字符串。使用网络通信工具进行网络通信或者浏览网络视频时，通信信息或视频数据从A机到B机，大部分人理解的情况可能如图1-5左侧所示，但实际上复杂度远远超过我们的想象，真实情况更趋近于图1-5右侧所示。

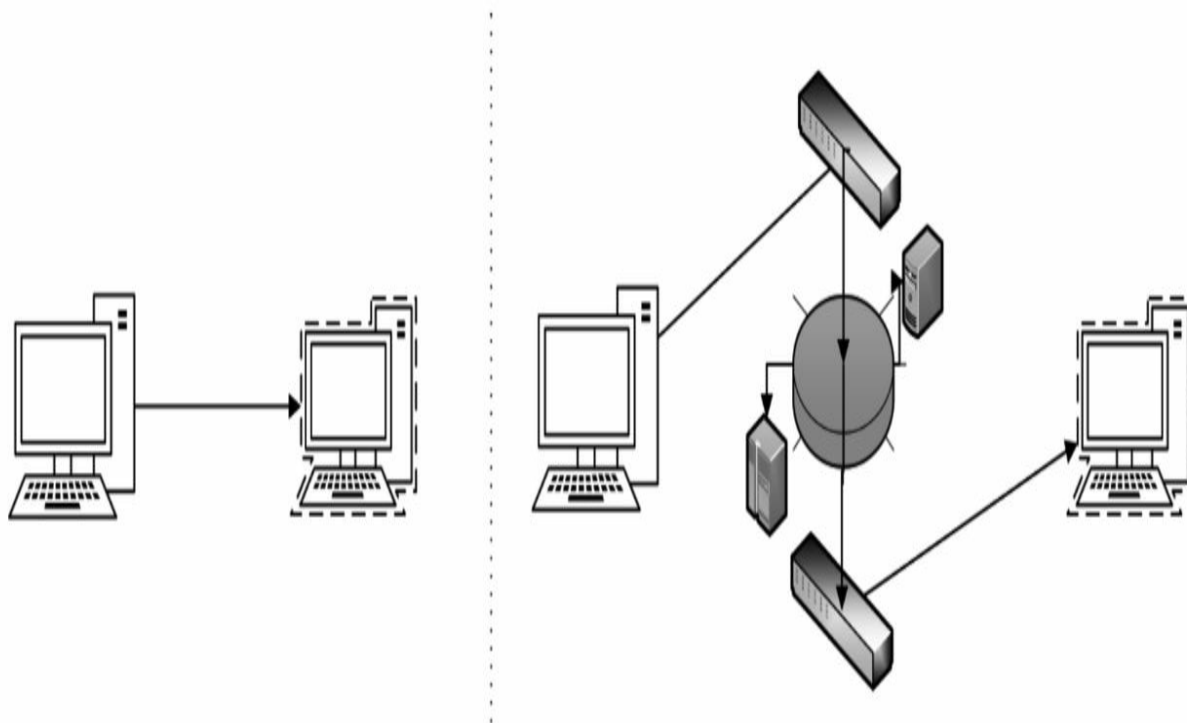


图1-5 网络拓扑

数据通过层层局域网或者交换机才能到达目标机器。这么长的传输距离，为了保证数据有效地传送到目标机器，设计出了网络通信协议。

常用的网络通信协议有TCP、UDP两种。TCP用于可靠的长连接，UDP一般用于速度比可靠性更重要的情况，例如传输视频流时，直播一般采用的也是UDP。UDP允许有数据的丢失，TCP则不允许。本书主要讲解的是TCP/IP，因为游戏数据是要基于数据稳定性的，TCP对于没有发送成功的包会重复发送，UDP则会直接丢弃。试想一下，看视频时，画面卡顿几秒不会有什么感觉，但在竞技游戏中，少发送一个技能则可能使我们输掉整场比赛。

理论上，开放系统互连参考模型（Open System Interconnection Reference Model，OSI参考模型）为7层。TCP/IP并不完全匹配这7层，它有4层协议。

图1-6简洁地说明了TCP/IP四层协议中的基本结构。链路层囊括网络层的数据，而网络层又囊括传输层的数据。像洋葱一样，数据发送时

一层一层被包裹起来，到了发送目的地，像剥洋葱一样，又一层一层剥离，最终得到应用层需要的数据。在这个过程中，首先需要理解“层”的概念，表1-1详细说明了每个协议层级的作用。

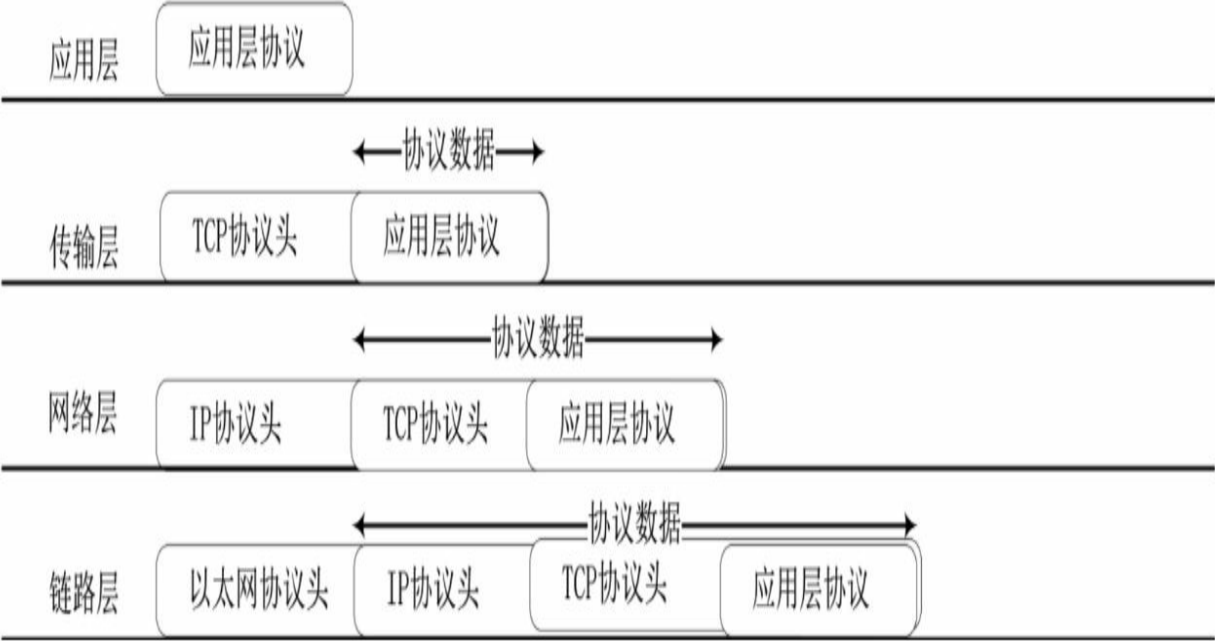


图1-6 网络通信协议层级

表1-1 协议层级说明表

层 级	说 明
应用层	应用协议包括 HTTP、FTP、自定义协议等。例如，我们浏览一个网站，输入网站地址并按回车键后，就会生成 HTTP 请求协议。对于游戏应用来说，应用层协议就是自定义的协议数据格式
传输层	主要负责数据在两台机器上传递的可靠性。常用的传输协议为 TCP（Transmission Control Protocol，传输控制协议）和 UDP（User Datagram Protocol，用户数据报协议），协议的数据标识存储在协议头中。传输层将应用层的协议加上 TCP 或 UDP 协议头，组织成了新的传输层协议
网络层	在网络层有 3 个重要的协议，分别是 IP、ARP 和路由协议。其中 IP 实现两个基本功能：寻址和分段。ARP 即地址解析协议，简单来说，它发挥了一个重要的作用：IP 与物理地址的转换。在使用计算机时，接触比较多的是 IP 地址（例如 192.168.0.1），但是在物理设备传输时，真正用到的却是 MAC 物理地址。这就是 ARP 存在的原因，它将两个地址进行了关联。数据到了网络层之后，在传输层协议的基础上又增加了网络层的协议头
链路层	以太网协议，实现物理设备之间的数据传递。在这一层中，物理设备之间传递的数据增加了以太网协议头

简单理解这几个层级之后，下面从应用与操作系统的层面来看协议在这几个层级的处理。发送数据时，如图1-7所示，数据从左侧自上而下叠加，而到了目标主机，数据是自下而上剥离的，最终到了应用层面，就变成了当初发送的数据。因为本书偏重于游戏编程，所以对于 TCP/IP 的底层协议不进行过多解释。如果读者对网络协议感兴趣，可以阅读一些相关的图书。

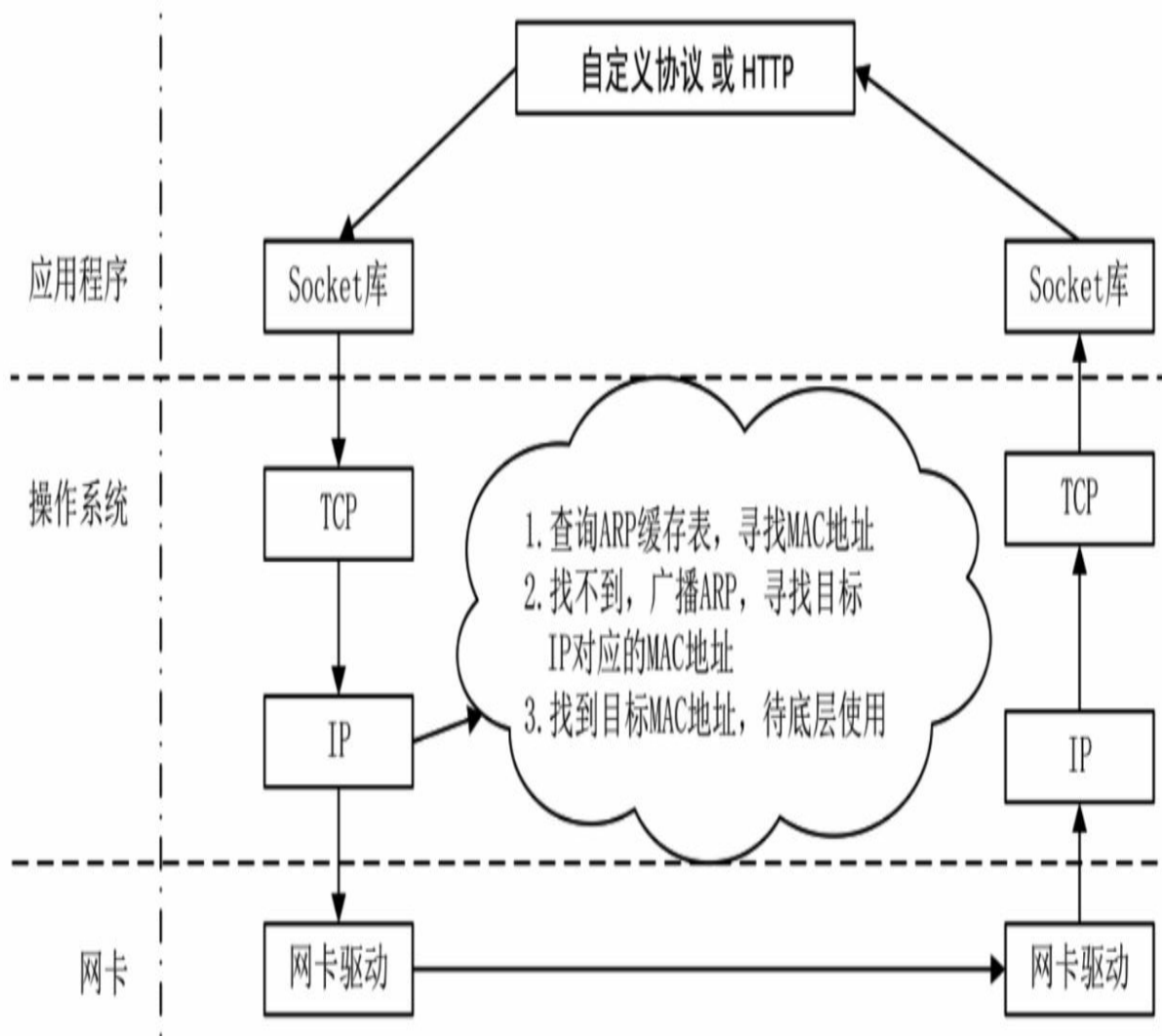


图1-7 操作系统与网络

作为一本编程图书，我们重点关心的是自定义协议与Socket库的相关操作，将目光锁定在数据应用层，而对于整个TCP/IP只需要有大概的了解即可。后续章节将由浅入深地介绍几个经典的网络编程模型。对于网络游戏来说，网络编程属于系统底层逻辑的部分，因此本书会花费一整章来细细讲解，以便读者充分了解并在以后好好使用，为上层逻辑打好基础。

阻塞式网络编程

下面就从简单的网络模型入手来实现一个简单的网络程序。要达到的目的如下：

- (1) 客户端与服务端建立网络通信。
- (2) 完成通信之后，客户端向服务端发起一条协议。
- (3) 服务端收到协议，并转发给客户端。
- (4) 客户端收到协议，打印出来。

在这个例子中，将展示网络编程的基本功能——接收和发送，处理服务端与客户端对于Socket的不同表现，实现服务端与客户端的收发协议流程。

1.4.1 工程源代码

该工程的源代码在本书源代码库的01_01_network_first目录下。先来执行工程，看看结果如何（注：在不少中文版的集成开发环境中把英文版中的“Project”翻译成“项目”，因此工程和项目在这种语境下指同一个概念，例如工程文件就是指项目文件）。工程中提供了两种打开方式，一种是在Windows系统下的Visual Studio工程文件，另一种是在Linux系统下的CMake文件。如果读者还不了解CMakeLists.txt文件的定义，那么建议先阅读附录中的CMake部分。本书提供的所有源代码均有这两种打开方式。

如果在使用Windows编译源代码时出现SDK版本不一致的问题，那么右击“解决方案”，选择“重定解决方案目标”。产生这个问题的原因是工程原来指定的SDK与本地环境中的SDK版本不一致。再次提醒，编译目标为“debug, x86”。

现在看看在Linux上如何执行本例。进入工程目录，执行脚本make-all.sh，编译的步骤已经在这个脚本中写好了，本书所有的工程都采用该方式编译。

```
[root@localhost 01_01_network_first]# ./make-all.sh
```

执行make-all脚本时，它会将该工程上的所有可执行程序都进行编译，本例中生成了两个文件：clientd和serverd。为了便于调试，所有的库文件源代码都是直接编译到执行文件中的，不再生成中间的静态库文件。

这是我们第一次使用make-all.sh脚本，每个工程都会有该脚本，用于批量编译。读者可使用“vim ./make-all.sh”命令查看该脚本。

这里做一个简短的说明，在make-all.sh脚本中提供了两个参数，默认情况下，采用Debug模式编译代码，如果执行命令“./make-all.sh release”，就编译Release版本。除此之外，还可给定clean参数，即执行“./make-all.sh clean”，目的是清除CMake生成的临时文件，重新生成Makefile文件。

在脚本中提供了一个build函数，该函数的目的是对给定目录下的所有工程进行编译。以src/libs目录为例，函数build对libs目录下的src/libs/network/目录进行了编译。这个目录是网络库工程，其下有一个已经写好的CMakeLists.txt文件。该文件与附录中讨论的文件格式大同小异，有3个地方值得注意。

(1) 编译文件名

```
set(MyProjectName networkd)
```

指定一个编译文件名。当属性CMAKE_BUILD_TYPE为Debug时，输出文件加了d字符，以方便区分Debug和Release版本。CMake提供的STREQUAL函数用于字符串比较。

(2) 输出目录

```
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "../.../libs")
```

设置属性CMAKE_ARCHIVE_OUTPUT_DIRECTORY。该属性指定了静态库生成的目录。

工程生成的可执行文件放在工程目录下的bin目录中，库文件放在工程目录下的lib目录中。不论是Windows系统还是Linux系统都遵从该规则。

(3) 生成文件

在CMakeLists.txt文件的最后使用了add_library指令，而不是add_executable指令。

```
add_library(${MyProjectName} STATIC ${SRCS})
```

add_executable生成可执行文件，add_library则生成一个库。关键字STATIC表示要生成一个静态库，需要生成动态库时，关键字改为SHARED即可。

下面看看第一个例子的执行结果，执行make-all.sh编译完成之后，进入bin目录。

```
[root@localhost 01_01_network_first]# cd bin/
[root@localhost bin]# ls
clientd  serverd
```

在bin目录下生成了serverd和clientd两个可执行文件。先运行serverd，再运行clientd。

在表1-2中展示了服务端和客户端的打印数据。进程serverd开始运行之后，就会进入网络监听状态，当clientd启动后，serverd收到一个连接请求，打印“accept one connection”，双方连接成功之后，clientd首先发出数据“ping”，serverd收到之后返回一条相同的数据，最后clientd收到serverd发出的数据。这个简单的例子完成了一个来回的数据发送与处理。看看流程图可能会更容易厘清思路，如图1-8所示，图中标注了数据流转的4个步骤。

表1-2 阻塞式网络通信运行结果

服 务 端	客 户 端
[root@localhost bin]# ./serverd	[root@localhost bin]# ./clientd
accept one connection	::send.ping
::recv.ping	::recv.ping
::send.ping	

- ①客户端发送ping数据。
- ②服务端收到ping数据。
- ③服务端收到ping数据的同时返回一个ping给客户端。

④客户端收到ping数据。

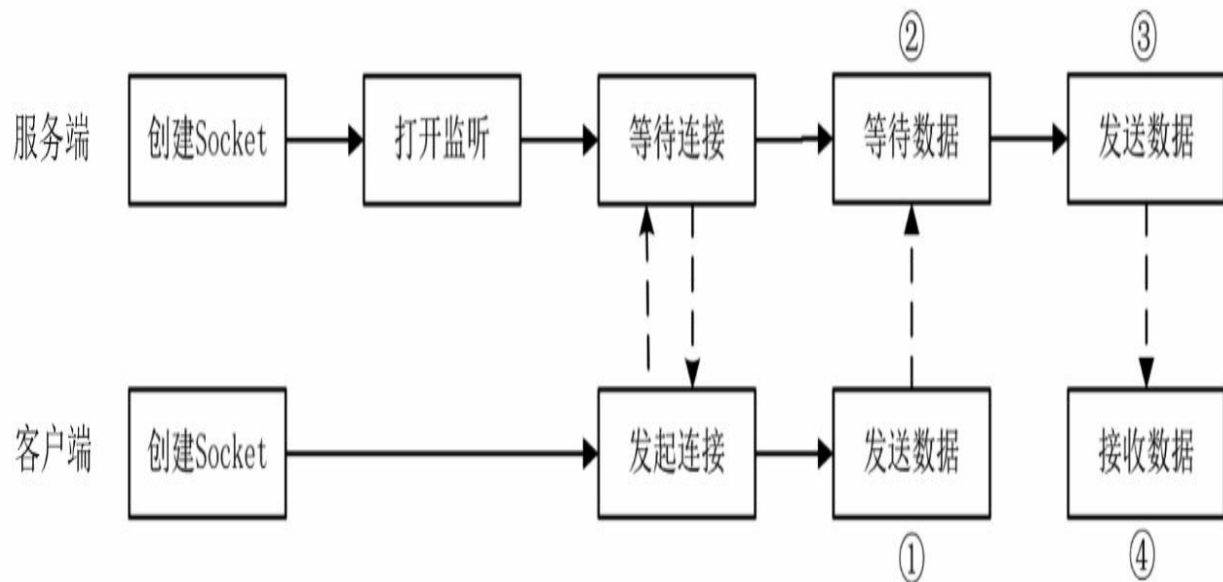


图1-8 阻塞式网络通信流程

网络监听和连接是如何实现的呢？无论是服务端还是客户端，首先要做的事情都是创建一个Socket（套接字）。那么Socket是什么呢？下面通过代码分析来简要说明。

1.4.2 服务端代码分析

服务端的主要代码在server.cpp文件中。对比源代码进行查看，需要掌握以下几个关键点。

关键点1: Socket初始化

代码的第一行为创建Socket做初始化准备：

```
_sock_init( );
```

首先要了解什么是Socket。简单来说，Socket定义了IP地址上的一个通信连接。例如，同一台计算机向同一个服务端发起两个网络连接，这就是两个通信连接，不论在客户端还是服务端都会产生两个不同的Socket。Socket实际上就是一个ID，也可以用通道来理解这种通信。打个比方，你有一个手机号码，当你打电话给别人的时候，你与对方建立了一个通信通道。Socket也是类似的，每次通信开始，通信双方建立了Socket，这个Socket被分配了一个ID，一个固定的ID固定了通道，避免收到错误消息。

_sock_init() 定义在network工程的network.h文件中。在Windows系统下的宏定义为：

```
#define _sock_init( ) { WSADATA wsaData; WSAStartup( MAKEWORD(2, 2
```

在Windows系统下，需要初始化执行WSAStartup函数。在Linux系统下，调用：：socket函数之前不需要执行任何操作，所以定义了一个空宏。

关键点2: 创建Socket

初始化操作完成之后需要创建一个Socket。创建失败会调用宏_sock_err来显示其错误。创建代码如下：

```
SOCKET socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (socket == INVALID_SOCKET) {
        std::cout << "::socket failed. err:" << _sock_err() << st
        return 1;
    }
```

函数：`::socket`是底层函数，这个函数调用的细节放在后面来讲解。它在两个系统下略有不同。在Linux系统下，位于`/usr/include/sys/socket.h`文件中，其定义如下：

```
extern int socket (int __domain, int __type, int __protocol) __THR
```

从函数返回值可以看出，生成的Socket为int类型。

而在Windows系统下，返回值为SOCKET类型。`socket()`函数定义如下：

```
SOCKET WINAPI socket(_In_ int af, _In_ int type, _In_ int protocol
typedef UINT_PTR SOCKET;
```

在Windows系统下，SOCKET类型被重定义为一个UINT类型，也就是说，如果编译的版本为32位，SOCKET类型就为unsigned int，64位则为unsigned__int64，总之，SOCKET本身也是一个数值类型。

为了兼容Linux和Windows这两个系统，工程中在Linux下定义了两个宏：SOCKET和INVALID_SOCKET。

```
#define SOCKET int
#define INVALID_SOCKET -1
```

虽然两个系统中的定义略有不同，但：`::socket`函数的表现却是相同的。如果调用：`::socket`函数失败，在Linux系统下的返回值为-1，在Windows系统下返回一个宏定义INVALID_SOCKET。

除了定义不同之外，两个系统显示出错信息的函数也有差别，工程中定义了_sock_err宏来处理。关于这个宏，在Windows系统和Linux系统下的定义不同，其定义如下：

```
#ifndef WIN32
#define _sock_err( ) WSAGetLastError()
#else
#define _sock_err( ) errno
#endif
```

关键点3：绑定IP与端口

创建Socket之后需要指定IP地址和端口实现绑定操作，本例中指向了本机127.0.0.1的2233端口。

```
sockaddr_in addr;
memset(&addr, 0, sizeof(sockaddr_in));
addr.sin_family = AF_INET;
addr.sin_port = htons(2233);
::inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr.s_addr);
if (::bind(socket, reinterpret_cast<sockaddr *>(&addr), sizeof(addr)) < 0)
{
    std::cout << "::bind failed. err:" << _sock_err() << std::endl;
    return 1;
}
```

在绑定过程中使用了sockaddr_in结构，该结构中指定了协议族、IP地址和端口。

之前讨论过IP地址可以唯一标识一台计算机，但一台计算机可能有多个IP地址，至少可以有一个对外的地址和一个对内的地址。日常中，设置IP地址为192.168.0.120，这是一个内网地址，127.0.0.1是一个特定的描述，指向本机，也是一个内网地址。在本例中，打开127.0.0.1的2233端口，开放的范围只是本机，也就是说这个服务端只能由本机上执行的clientd对它进行连接。如果这里填写的IP地址是192.168.0.120，那么开放的范围是整个局域网，离开了局域网就不能访问了。如果计算机还有一个公网IP地址，调用::bind函数绑定的是一个公网IP地址，那么在任何地方、任何计算机上都可以访问这个IP地

址开放的端口。

特别说明：如果在Linux下反复测试时遇到了错误“: : bind failed.Err: 98”，则是因为之前绑定的端口没有被释放，系统有一定的回收时间。为了快速释放，我们可以输入“ss-lnpt”命令找到端口对应的PID，使用kill指令杀掉进程。

```
[root@localhost bin]# ss -lnpt
State      Recv-Q Send-Q Local Address:Port      Peer Address:Port
LISTEN     0      10    127.0.0.1:2233        :*users:(("logind",pid=63
[root@localhost bin]# kill -9 6367
```

关键点4: 监听网络

绑定好IP地址和端口之后，还需要打开对Socket的监听，服务端的工作才算完成。其代码如下：

```
int backlog = GetListenBacklog();
if (::listen(socket, backlog) < 0) {
    std::cout << "::listen failed." << _sock_err() << std::endl;
    return 1;
}
```

对于服务器来说，这是必不可少的一步，调用了底层函数：`listen`。只有打开了监听，我们才可以敏锐地察觉是否有客户端对该端口发起了通信请求。参数backlog指定了请求缓存列表可以有多长。

关键点5: 等待连接

有了监听就可以接收连接了，接收连接的代码如下：

```
struct sockaddr socketClient;
socklen_t socketLength = sizeof(socketClient);
int newSocket = ::accept(socket, &socketClient, &socketLength);
```

在本例中，程序会在：`accept`函数阻塞住。如果在：`accept`后面一行打一个断点，会发现断点不会被触发。`accept`函数的功能是接收一个请求，如果没有就会一直等待。所以，在运行了serverd还没有运行clientd之前，服务端处于阻塞状态，它在等待一个连接请求。

在一些网络编程图书中会讨论TCP的3次握手，在本程序中，我们并没有看到3次握手，而是通过：`accept`函数就收到了连接。这并不是说3次握手不存在或者没有完成，而是这3次握手过程已经在底层完成了。

关键点6：接收数据

当：`accept`函数接收到有新的连接到来时，双方通信建立完成，可以开始发送数据。代码中调用了底层：`recv`函数接收数据，再调用：`send`函数发送接收到的数据。这部分的代码如下：

```
char buf[1024];
memset(&buf, 0, sizeof(buf));
auto size = ::recv(newSocket, buf, sizeof(buf), 0);
if (size > 0) {
    std::cout << "::recv." << buf << std::endl;
    ::send(newSocket, buf, size, 0);
    std::cout << "::send." << buf << std::endl;
}
```

在上面的代码中，接收数据放在一个长度为1024字节的缓存中，接收到什么数据就发送什么数据回去。

关键点7：关闭Socket

关闭Socket调用了两个宏：

```
_sock_close(socket);
_sock_exit();
```

在Windows和Linux系统下做了不同的处理。在Windows系统下初始

化时调用了WSAStartup函数，结束时则需要调用WSACleanup函数。宏定义如下：

```
#ifndef WIN32
#define _sock_exit( )
#define _sock_close( sockfd ) ::close( sockfd )
#else
#define _sock_exit( )      { WSACleanup(); }
#define _sock_close( sockfd ) ::closesocket( sockfd )
#endif
```

特别说明：在Linux下关闭Socket时，有时使用：`::close`函数，有时使用：`::shutdown`函数。这两者有什么区别呢？

可以做一个实验，将：`::close`函数替换为：`::shutdown`函数。在生成和关闭Socket处进行打印，从打印信息中可以看出，使用：`::shutdown`函数，关闭过的Socket即使关闭了，也不会重用。每次有新的连接到来时，就需要用到新的Socket，其值会在之前的值上加1。这是因为：`::shutdown`函数会关闭TCP连接，但不释放Socket。而：`::close`函数会将套接字计数减1，当计数==0时，会自动调用：`::shutdown`函数。看上去在关闭连接时使用：`::close`才是正确的，但为什么还是有人直接使用：`::shutdown`呢？因为使用：`::close`并不能真正断开连接，它只是计数减1，在某些情况下，可能需要直接断开连接，所以调用：`::shutdown`函数关闭网络。

而使用：`::close`函数，在某些情况下，Socket的状态会变为CLOSE_WAIT状态，CLOSE_WAIT实际上就是等待关闭状态。

出现这种情况的原因比较复杂，有一种情况是客户端想关闭，但服务端可能还在读或写，就产生了等待。在网络编程中，这是我们需要特别注意的一个问题。

1.4.3 客户端代码分析

客户端的源代码与服务端略有不同，相对来说步骤简单一些。客户端的主要逻辑在client.cpp文件中，需要掌握以下几个关键点。

关键点1: Socket初始化

在客户端开始时，同样初始化和创建了Socket，与服务端原理一致。

```
_sock_init();
SOCKET socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (socket == INVALID_SOCKET) {
    std::cout << "::socket failed. err:" << _sock_err() << std::endl;
    return 1;
}
```

关键点2: 网络连接

对于客户端来说，并不需要执行::bind绑定函数，也不需要监听端口。客户端需要执行的是调用::connect函数，它向一个指定的IP和端口发起连接操作。函数::connect会用到sockaddr_in结构。调用代码如下：

```
sockaddr_in addr;
memset(&addr, 0, sizeof(sockaddr_in));
addr.sin_family = AF_INET;
addr.sin_port = htons(2233);
::inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr.s_addr);
if (::connect(socket, (struct sockaddr *)&addr, sizeof(sockaddr))
    std::cout << "::connect failed. err:" << _sock_err() << std::endl;
    return 1;
}
```

在客户端，使用sockaddr_in结构的初始化工作与服务端一样。客

客户端调用了服务端没有涉及的底层函数：`connect`，向一个指定的地址（也就是在服务端开放的地址）发起了一个连接。

关键点3：发送数据

客户端与服务端一致，都是调用底层函数：`send`发送数据。使用下面的代码发送一个"ping"字符串：

```
std::string msg = "ping";  
::send(socket, msg.c_str(), msg.length(), 0);
```

关键点4：接收数据

客户端发送数据之后，陷入等待操作中，函数：`recv`等待接收数据。

```
char buffer[1024];  
memset(&buffer, 0, sizeof(buffer));  
::recv(socket, buffer, sizeof(buffer), 0);  
std::cout << "::recv." << buffer << std::endl;
```

1.4.4 系统差异

在前两小节中，对Linux和Windows两个系统进行了有区别的编码。除了Socket的定义之外，还有两个大的区别：

（1）虽然网络API是底层函数，但在Windows系统下创建Socket之前需要调用WSAStartup函数，而退出的时候需要调用WSACleanup函数。两个函数的定义如下：

```
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA);  
int WSACleanup();
```

（2）获取错误的方式也略有不同。在Linux系统下使用errno变量，在Windows下使用WSAGetLastError函数，如果执行网络API时出错，该函数就会返回一个错误码，定义如下：

```
int WSAGetLastError();
```

1.4.5 网络底层函数说明

前面举了一个简单的例子，用到的底层函数是网络编程的基础，本书之后的所有示例都是基于这些网络底层函数来完成的。为了加深理解，下面对这些底层函数逐一进行详细的说明。

1.函数：： socket

客户端与服务端在初始化时，均使用了：： socket函数。每个网络通信必有一个Socket。函数的参数说明见表1-3，原型如下：

```
int socket(int family, int type, int protocol);
```

表1-3 socket函数参数

参 数 名	说 明
family	常用值为 AF_INET、AF_INET6、AF_UNSPEC，分别表示 IPv4、IPv6、不指定，AF 为 Address Family 的缩写
type	常用值为 SOCK_STREAM、SOCK_DGRAM。该参数用于指明要创建的 Socket 的类型。SOCK_STREAM 表示需要创建的是一个有序、可靠的数据流，而 SOCK_DGRAM 则表示创建非连续的数据报
protocol	常用值为 IPPROTO_UDP、IPPROTO_TCP。该参数为 Socket 指定协议的类型。IPPROTO_UDP 表示采用 UDP，IPPROTO_TCP 则表示采用 TCP

在本节的例子中，不论是在服务端还是客户端，生成Socket时均采用了AF_INET、SOCK_STREAM、IPPROTO_TCP这3个参数，即采用IPv4协

议，以SOCK_STREAM数据流发送时采用可靠的TCP。

正常情况下，调用该函数会返回一个大于零的正数，即Socket值。这个值是唯一的，由系统分配。如果A与B建立了连接，那么对于A来说，在连接没有中断的情况下，它一定是一个定值，且不与其他Socket值相同。Socket这个单词在英文里的翻译是插槽、插座，在网络术语中一般翻译成套接字，可以将其理解为如果占用了一个插槽，其他人是不可能再使用的。

在Windows下，生成的Socket值是随机大于1000的值，在Linux下，它是从个位数开始累计的。在Linux下，Socket值也被称为描述符。这和Linux的内核结构有关联，这里不需要深究，只需要知道Socket也称为描述符即可。一旦Socket创建成功，返回的Socket值就会成为网络通信的重要依据。需要注意的是，Socket值是可以被复用的，也就是说，如果最开始分配了123给A，随后A断线，C上线，C也有可能分配到123。

2.函数：：bind

：：bind是服务端必不可少的一个函数，其作用是指定IP和端口开放给客户端连接，客户端则没必要调用该函数。函数的参数说明见表1-4，函数的原型如下：

```
int bind(int sockfd, const sockaddr *address, int address_len);
```

表1-4 bind函数参数

参 数 名	说 明
sockfd	Socket 值
address	Socket 地址
address_len	地址结构长度

参数sockfd即调用：`socket`函数之后得到的Socket值。

sockaddr是通用的套接字地址结构，在代码中还使用了sockaddr_in结构，这两者在这里没有什么差别，长度一样，可以相互转换，sockaddr_in是Internet环境下套接字的地址形式。参数address指定了Socket需要绑定的地址信息，这些信息中包括IP和端口。

该函数返回负数就表示出错了。如果试图绑定一个已经在使用的端口，调用：`bind`就会失败。

3.函数：`listen`

`listen`是服务端调用的函数，用于对IP地址和端口的监听。函数的参数说明见表1-5，函数的原型如下：

```
int listen(int sockfd, int backlog);
```

表1-5 `listen`函数的参数说明

参 数 名	说 明
sockfd	Socket 值
backlog	等待连接队列的最大长度

关于连接队列的最大长度，可以使用系统的宏SOMAXCONN。在Linux下，它的定义在/usr/include/bits/socket.h文件中，默认为128。值backlog的意义在于，当一个连接请求到来时，另一个连接请求可能同时到来，系统需要缓存其中之一，backlog是系统处理连接的缓冲队列的长度。虽然TCP有3次握手，但是目前来看这个过程还是相当快的，所以5~10个缓存已经足够使用。

该函数返回负数就表示出错了。

4.函数：：connect

：：connect是对一个已知地址进行网络连接的函数。一旦客户端调用：：connect函数，就会触发TCP的3次握手协议，3次握手完成之后，在服务端会调用：：accept函数。

在调用：：connect函数时，如果失败，就不能对当前Socket再次调用：：connect函数，正确的做法是关闭Socket再次调用：：connect函数。该函数的参数与：：bind函数是一致的。函数的原型如下：

```
int connect( int sockfd, const sockaddr *address, int address_len)
```

该函数返回负数就表示出错了。

5.函数：：accept

该函数用于监听端口，若：：accept收到数据，则一定有一个新的连接被发起。函数的参数与：：bind函数是一致的。函数的原型如下：

```
int accept(int sockfd, sockaddr* address, int* address_len);
```

函数：：accept返回的值是一个新的Socket值。调用：：accept时，我们传入了一个Socket值，这个值可以称为监听Socket，而返回的这个新值就是客户端连接服务端的连接Socket。这两个值是有区别的。服务端有且仅有一个监听Socket，却可以有无数个连接Socket。

6.函数：：send和：：recv

函数：：send和：：recv是一对用于发送和接收数据的函数，除了这两个函数之外，网络底层还提供了其他发送和接收数据的函数，适用于不同的场合，这里只介绍我们使用的这一对函数。函数的参数说明见表1-6，函数的原型如下：

```
int send(int sockfd, const char *buf, int len, int flags)
int recv(int sockfd, char *buf, int len, int flags)
```

表1-6 发送、接收函数参数

参 数 名	说 明
sockfd	Socket 值
buf	发送、接收缓存指针
len	缓存长度
flags	标志位，一般为 0

在4个参数中，需要着重说明的是buf参数。对于：`send`函数而言，发送的缓冲buf是const指针，而len则是发送数据的长度。

对于：`recv`函数而言，buf是接收数据的缓存，len是该缓存的长度。假设服务端向客户端发送了2024字节的数据，但客户端接收buf的长度只有1024，len的长度也只能为1024，即：`recv`函数一次只会读取系统底层网络缓冲中的1024字节，放入buf缓冲中。这个概念非常关键，会引发粘包、拆包的问题。网络数据并不是我们想象中一条一条规整地发送过来的，有可能接收的1024字节里面有3个协议数据，也有可能接收的1024字节只是某个协议的一部分，需要多次读取。在后面的例子中，我们会详细讲解这些情况该如何处理。

发送和接收函数调用失败返回非正数，若成功，则返回发送、接收的字节长度。对于：`recv`函数来说，若返回0，则表示在另一端发送了一个FIN结束包，网络已中断。但：`recv`函数在返回负数时，也并不都意味着网络出错而需要断开网络，这在后面用到的时候再讲解。

1.4.6 小结

我们已经对网络通信有了一定的了解，客户端和服务端在初始化时略有不同，但收发数据的流程是相同的，本例采用了阻塞式的收发数据方式，所有的代码都是在阻塞模式下进行的。函数：`:: accept`、`:: send`和`:: recv`都处于阻塞模式下。

所谓阻塞，就是一定要收到数据之后，后面的操作才会继续。客户端调用：`:: connect`函数连接到服务端，发送数据之后一直阻塞在：`:: recv`函数上，直到收到服务端传来的数据才退出。服务端同样是阻塞的，在：`:: accept`函数处等待连接进来，如果没有就一直等待，接收到一个连接之后，再次阻塞，等待：`:: recv`函数返回数据。

作为服务端，采用阻塞模式显然不够高效。一般来说，服务端需要同时处理成千上万个通信，不能因为一个连接而阻塞另一个连接的收发数据进程。在实际情况下，更常用的是非阻塞模式。接下来以一个实例来说明非阻塞模式是如何工作的。

非阻塞网络编程

本节提供一个非阻塞功能的例子，要达到的目的如下：

- (1) 客户端与服务端建立网络通信。
- (2) 建立通信之后，客户端发起3个线程，分别向服务端发送不同数据。这些数据是ping_0、ping_1和ping_2。
- (3) 服务端收到数据，将相同数据转发给客户端。
- (4) 客户端收到数据，验证并打印结果。

本例在基础收发数据的功能上增加了非阻塞的设置，虽然看似只有细微改变，但其逻辑会变得非常不一样。

1.5.1 工程源代码

工程的源代码在本书源代码库的01_02_network_nonblock目录下，使用make-all脚本进行编译之后，在01_02_network_nonblock/bin目录下会生成clientd和serverd两个文件，先启动serverd，再启动clientd。先来看一下这个非阻塞工程的执行结果，见表1-7。

表1-7 非阻塞式网络通信运行结果

客 户 端	服 务 端
[root@localhost bin]# ./clientd	[root@localhost bin]# ./serverd
::send.ping_2 socket:3 // 1组	new connection.socket:4 // 1组
::send.ping_1 socket:4 // 3组	::recv.ping_2 socket:4 // 1组
::recv.ping_2 socket:3 // 1组	::send.ping_2 socket:4 // 1组
::send.ping_0 socket:5 // 2组	new connection.socket:4 // 2组
::recv.ping_0 socket:5 // 2组	::recv.ping_0 socket:4 // 2组
::recv.ping_1 socket:4 // 3组	::send.ping_0 socket:4 // 2组
	new connection.socket:4 // 3组
	::recv.ping_1 socket:4 // 3组
	::send.ping_1 socket:4 // 3组

在Linux下，按Ctrl+C组合键可退出服务端进程。执行本例时，每一次产生的结果都不一样。因为是线程操作，客户端发送数据的前后关系不确定，所以服务端收到数据的顺序也不确定。为了便于区分，将数据按到达服务端的时间分成了1组到3组，每一组数据在同一个Socket通

道上。虽然这里只有几行简单的数据打印，但有几个关键点需要理解。

关键点1: Socket值的重用

从表1-7中可以看到，客户端首先发送了两条数据，在Socket值为3和4的通道中分别发送了ping_2和ping_1。

服务端首先收到了ping_2的数据。在打印信息时显示了一个Socket值，在Linux上，Socket的值是线性增加的，因为程序在启动时会用到前面的几个描述符，所以当前可用描述符是从4开始的。这意味着服务端建立第一个连接时，它的Socket值为4，数据是ping_2，服务端收到了该数据并发送了相同的数据给客户端。接着服务器收到了ping_0的数据，Socket值依然是4。

这里读者一定有疑问，不是说不同的通道值不一样吗？为什么两个Socket却用了一样的值？这是因为Socket是可以重用的，如果关闭了描述符为4的Socket，当有新的连接到服务端时会将描述符4重新分配给它。

从上面的数据中可以看出，当服务端收到ping_2并将它发送出去之后，会马上关闭Socket 4，这时又收到ping_0，Socket 4被再次分配。

关键点2: Socket值是进程级数据

认真观察会发现，客户端发送ping_2的时候，在客户端用的Socket值为3，为什么服务端接收到的Socket值却是4？客户端在发送ping_1的时候也使用了Socket 4，服务端和客户端使用了相同的Socket 4值，为什么没有出错？

这里需要区分一个概念，Socket值是在进程中独立的，不是通用的。它不像端口，就某个端口而言，一台物理机就只有一个，同一时间不可重用。但同一时间不同的进程可能存在相同的Socket值的通道。在客户端的Socket 3和在服务端的Socket 3不是同一个通道，只是值相同而已。

关键点3: 网络数据的无序与有序

在客户端，从打印结果可以看出，先发送了ping_1，再发送了ping_0。但是在服务端，收到数据的时候却正好相反，先收到了ping_0，后收到了ping_1，这和发送数据的顺序不一致。网络数据的收发是无序的，两个连接即使同时发送数据，也有可能这次你先到，下次我先到。但同一个Socket连接，如果先发了Msg0，再发送Msg1，在TCP下，服务端收到的数据一定是有序的，必定是先收到Msg0，再收到Msg1。

在网络不稳定的时候，在TCP机制下，包丢失会重发。如果Msg0发送失败引起了重发，Msg1很有可能比Msg0先到达目标机器，这种情况是有可能存在的，但完全没必要担心，因为在TCP底层，有一套可靠的机制对收到的消息进行重新排序。如果出现错误，Socket连接就会抛出异常，也就是常说的玩家断线。

1.5.2 服务端代码分析

下面分析这个非阻塞的源代码，看看它是如何实现的，与阻塞的代码又有什么不同。还是先从服务端代码开始。

关键点1: Socket初始化

与前一个例子相比，服务器创建Socket、绑定IP地址、监听Socket的流程并没有发生变化，只看重点代码：

```
_sock_init();
SOCKET socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
...
_sock_nonblock(socket);
```

在服务端，Socket创建成功之后调用了一个新宏：`_sock_nonblock`。该宏设置Socket的属性，把阻塞模式变为非阻塞模式，在Windows下和Linux下的调用函数各不相同。宏对比如下（该宏定义在network.h文件中）：

```
#ifndef WIN32
#define _sock_nonblock( sockfd ) { int flags = fcntl(sockfd, F_GET
#else
#define _sock_nonblock( sockfd )
    { unsigned long param = 1; ioctlsocket(sockfd, FIONBIO, (unsi
#endif
```

在Windows下，函数*ioctlsocket*的目的是对Socket执行命令操作。在本例中，将Socket设置为非阻塞。函数*ioctlsocket*的说明可以在MSDN网上查到。原型如下：

```
int ioctlsocket(SOCKET sock, long cmd, u_long *argp);
```

在Linux下，调用*fcntl*，其原型如下：

```
int fcntl(int sock, int cmd, ...);
```

在Linux下，Socket被认为是一个文件描述符，可以用read、write、open等IO操作来操作它。而fcntl是对文件描述符进行属性操作的函数。在上面的宏定义中，首先传入参数F_GETFL，取得文件描述符的属性并保存到flags变量中，再在取得的属性之上增加一个O_NONBLOCK属性，即非阻塞模式。

关键点2：如何接收连接请求与收发数据

在上一个阻塞例子中，如果使用断点调试，就会发现在调用：`accept`、`::recv`和`::send`函数时会一直卡住，等待数据。现在，在非阻塞模式下，不论有没有数据，这些函数都会马上返回。在主线程中，不知道什么时候能收到`::accept`、`::recv`数据，因此做了个死循环。先浏览一下代码：

```
while (true) {
    SOCKET newSocket = ::accept(socket, &socketClient, &socketLeng
    if (newSocket != INVALID_SOCKET) {
        _sock_nonblock(newSocket); // 接收到一个新的连接请求，设为非阻
        sockets.push_back(newSocket);
    }
    // 遍历所有连接，调用::recv函数，查看是否有数据到来
    // 如果有，就调用::send函数将接收到的数据发送回去
    auto iter = sockets.begin();
    while (iter != sockets.end()) {
        SOCKET one = *iter;
        auto size = ::recv(one, buf, 1024, 0);
        if (size > 0) {
            ::send(one, buf, size, 0);
            iter = sockets.erase(iter);
            ...
        }
        ...
    }
}
```

在循环中，收到新的连接便把新的Socket值保存起来。每一帧不停

地主动询问在这个Socket上是否有数据，如果接收到数据，就发送一段相同的数据到客户端。

修改一下代码，把while循环去掉，再执行一下，会发现程序开始执行就退出了。因为所有的底层函数都不会再等待数据，没有数据就直接返回了。

如果在while循环中加一个计数并打印出来，就会发现一秒可以执行无数次的检查。非阻塞模式保证了每个Socket的收发都在同时进行，互不影响。

1.5.3 客户端代码分析

在客户端，为了保证多组Socket互不影响地发送数据，用到了线程。将每个连接包装成一个ClientSocket类，其定义在client_socket.h文件中。

每一个ClientSocket类的功能类似于1.4节中阻塞的例子，发送数据，阻塞等待数据到来，接收到数据，然后退出。图1-9展示了客户端与服务端通信的整体流程。

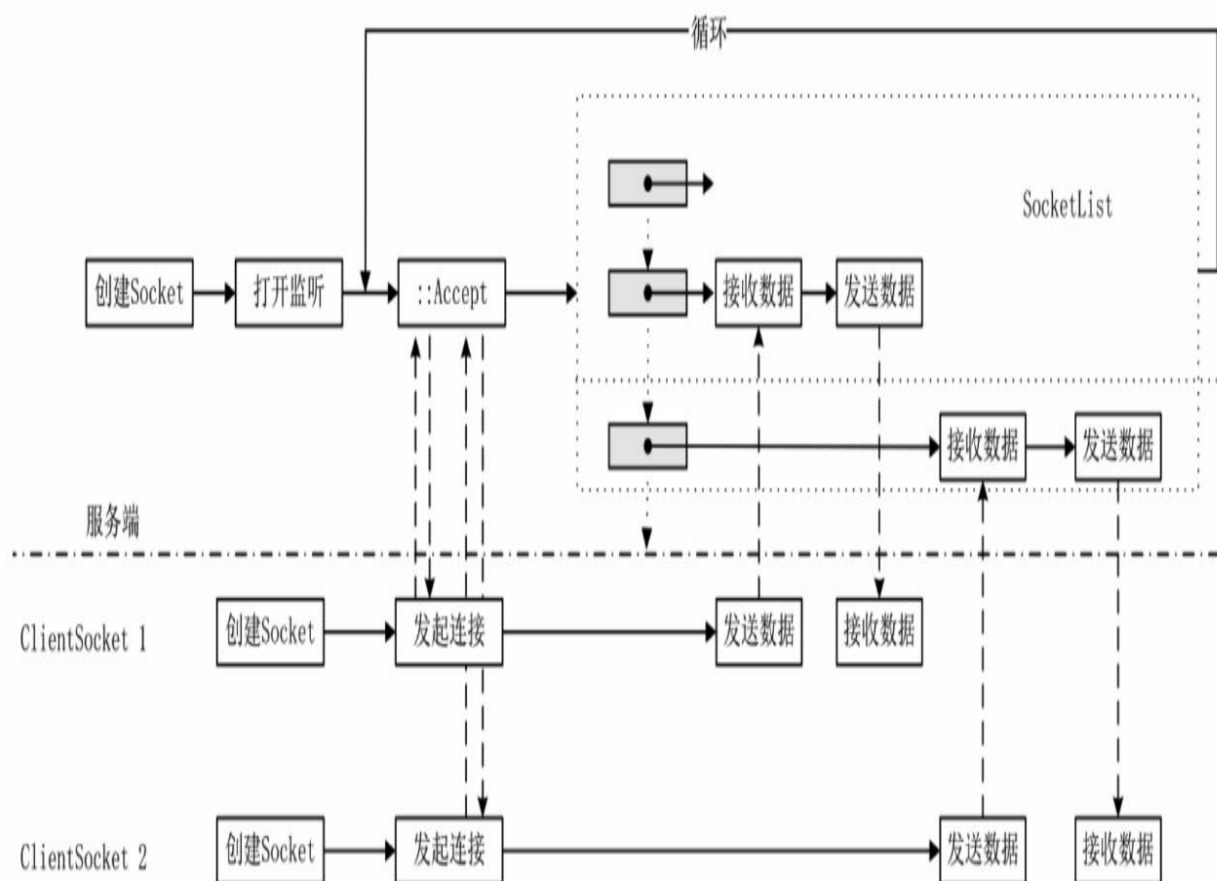


图1-9 非阻塞式网络通信流程

关键点1: ClientSocket类

为了让客户端互不影响，在client.cpp文件的main函数里创建了多个ClientSocket实例，每个实例都是在线程中执行的，即使在本线程中是阻塞的，也不会影响其他线程。ClientSocket类一共有4个函数，其中1个是构造函数。定义如下：

```
class ClientSocket {
public:
    ClientSocket(int index);    // 开启了一个线程
    void MsgHandler();          // 阻塞式的Socket收发数据流程
    bool IsRun() const;         // 收发数据是否已完成
    void Stop();                // 结束线程
private:
    bool _isRun{ true };
    ...
};
```

ClientSocket类的构造函数实现如下：

```
ClientSocket::ClientSocket(int index) : _curIndex(index) {
    _thread = std::thread([index, this]() {
        _isRun = true;
        this->MsgHandler();
        _isRun = false;
    });
}
```

在创建ClientSocket类时创建了一个线程，在线程中调用了MsgHandler函数。

函数MsgHandler的思路和前一个例子中的一样，即创建Socket，发送一条数据，再等待接收一条数据，使用的还是阻塞模式，这里不再重复。处理完数据之后，线程退出。

总之，ClientSocket类使用了线程的方式发送数据。

关键点2：主线程逻辑

在主线程中创建了3个ClientSocket实例，使用while不断循环查看

每个ClientSocket实例是否还处于IsRun运行模式下，如果已经完成，就调用ClientSocket的Stop函数，直到全部ClientSocket类都退出，程序结束。

```
int main(int argc, char *argv[]) {
    std::list<std::shared_ptr<ClientSocket>> clients;
    // 启动3个线程
    for (int index = 0; index < 3; index++) {
        clients.push_back(std::make_shared<ClientSocket>(index));
    }
    // 遍历当前所有ClientSocket类，如果已完成数据收发，就关闭线程，剔除数组
    while (!clients.empty()) {
        auto iter = clients.begin();
        while (iter != clients.end()) {
            auto pClient = (*iter);
            if ((*pClient).IsRun()) {
                ++iter;
                continue;
            }
            pClient->Stop();
            iter = clients.erase(iter);
        }
    }
    return 0;
}
```

主线程并不关心ClientSocket类中做了些什么，只是关心它的生命周期，每个线程结束了，整个程序就认为结束了。

特别说明：std::thread是C++标准线程库，在创建线程时使用了一个Lambda匿名函数，std::thread执行匿名函数。如果读者没有接触过Lambda匿名函数，就要学习一下Lambda匿名函数与常规代码的对比。下面的代码中，TestThread1使用了匿名函数，TestThread2则使用的是常规代码。

```
void TestThread1() {
    bool isrun = true;
    auto thread = std::thread([index, &isrun]() {
        printf("call test1.\n");
        isrun = false;
    });
}
```

```

    });
    while (isrun) {
        while (thread.joinable()) {
            thread.join();
        }
    }
}

void TestThread2() {
    bool isrun = true;
    auto thread = std::thread(&CallFunc, &isrun);
    while (isrun) {
        while (thread.joinable()) {
            thread.join();
        }
    }
}

void CallFunc(bool* pIsRun) {
    printf("call test2.\n");
    *pIsRun = false;
}

```

两个函数中代码的目的完全一样，一个使用Lambda匿名函数给线程注册了一个调用函数，而另一个则编写了一个常规函数注册到线程中。Lambda匿名函数使我们的代码显得更为简洁。

1.5.4 小结

在本例中，服务端使用了非阻塞模式接收数据。简单来说，就是你接收你的数据，我发送我的数据，互不影响。有数据到来就处理，至于对方是何时发过来的，接收数据方并不关心。

本例采用了主动询问数据的方式在服务端判定有没有数据直接调用：`recv`函数，每一个循环中都要对每一个Socket调用：`recv`函数，当返回值大于0时，认为有数据到来。

总结

至此，通过两个示例我们掌握了网络的基础编程知识，了解了什么是阻塞与非阻塞，并且了解了一些底层网络函数的使用。

假设在服务端，我们的机器每秒可以调用1000个循环，也就是说10个连接就有上万次函数调用，每一个循环中都要对每一个Socket调用：`recv`函数，而这些调用中可能只有极少数会真正有数据到达。

对于少量连接，也许我们还可以这样操作，一旦服务端连接增多，采用这样的方式效率就会降低。有没有一种方案，在没有数据的时候就跳过调用，而有数据的时候通知调用这些函数呢？这就是第2章要讲解的内容。

第2章

网络IO多路复用

本章介绍更为高效的网络编程方案，让系统底层去判断Socket是处于可读还是可写状态，在可读可写时通知上层，也就是我们常用的网络IO处理——IO多路复用。本章将介绍两个模型：Select和Epoll。在介绍这两个模型的同时还会引入自定义协议的内容。本章包括以下内容：

- Select模型示例。
- Epoll模型示例。
- 使用protobuf定义协议。

Select网络模型

Select是一个在Windows和Linux系统下通用的模型。该模型名为Select主要是因为它以名为：`select`的函数为驱动，该函数以不同的返回值来决定后续如何操作。简单来说，其主要的原理就是监视描述符（Socket）的变化来进行读写操作，且在必要时抛出异常。

在前面非阻塞的例子中，每一帧对每个Socket值进行了：`recv`函数调用，通过返回值来判断是否有数据到来，该方案的命中概率很低，而Select模型则提供了一种集合方式，让我们一次可以知道一个或者多个Socket是否有数据的变化。函数：`select`的主要功能是通过传入的参数告诉底层系统，我们关心的可读、可写、异常的Socket。该函数被调用之后会反馈给我们传入的这些Socket是否处于可读、可写状态。

2.1.1 ：：select函数说明

调用：：select函数，在Windows和Linux两个系统下的头文件不一致，其对比见表2-1。

表2-1 ：：select函数头文件

Windows 头文件	Linux 头文件
#include <WinSock2.h>	#include <sys/select.h>
	#include <sys/time.h>

函数：：select的原型在两个系统中是一致的，参数说明见表2-2。函数原型如下：

<pre>int select(int nfd, fd_set *readfd, fd_set *writefd, fd_set *ex</pre>

表2-2 ：：select函数参数列表

参 数	说 明
nfd	最大描述符加 1
readfd	fd_set 类型集合
writefd	fd_set 类型集合
exceptfd	fd_set 类型集合
timeout	超时

参数nfd在Windows下没有实质的作用，在Linux下则是Socket值集合内的最大值加1。这一点很好理解，Linux是线性增加的，最大值对于底层来说是有效的，而Windows是随机的，最大值即使传到底层也起不了什么作用。

中间3个参数都是fd_set集合，fd_set集合是装满Socket的容器。对于readfds中存储的Socket，系统会检测是否可读，如果可读，就表示有数据到达。对于writefds中存储的Socket，系统会检测是否可写，如果可写，就可以发送我们想要发送的数据。而对于exceptfds中存储的Socket，系统会检测是否有异常。

下面介绍这个函数一些值得注意的点。

关键点1：超时

每次调用：`select`函数的时候都会将进程变为一个极短时间的阻塞状态，参数timeout就是控制这个阻塞时长的。该参数是一个时间结构，其定义如下：

```
struct timeval {
    long    tv_sec;           /* seconds */
    long    tv_usec;         /* and microseconds */
};
```

它有3种设置方式：

(1) 设为`nullptr`，则表示永远等下去。参数中传入的Socket集合中，可读、可写或异常至少有一个返回值，函数：`select`才会有数据返回，这相当于让：`select`进入了阻塞状态。

(2) timeout有值时，表示等待一段时间。在等待的时间内，参数中传入的Socket集合中只要有一个准备好，函数就会有返回，返回值大于0。

(3) 完全不等待，把timeval中的秒和微秒都设为0。在完全不等待的状态下，返回值可能等于0，也可能大于0。等于0时，表示每一个监听的Socket都没有事件发生；大于0时，则有1个或多个Socket有事件

发生。

关键点2: fd_set集合

fd_set集合其实就是一组数组。有4个宏可以操作集合，表2-3罗列了操作函数。

表2-3 fd_set集合的操作函数

函 数 名	说 明
FD_CLR(socket, *set)	从集合 set 中把当前 socket 值删掉
FD_ISSET(socket, *set)	判断 socket 是否在给定的 set 集合中，如果在，就返回非零值
FD_SET(socket, *set)	将 socket 加入给定的 set 集合中
FD_ZERO(*set)	将给定的 set 集合初始化为空

参数readfds、writefds、exceptfds都是fd_set类型集合，从名字就可以分辨出其功能，readfds表示关心读的集合，writefds表示关心写的集合，exceptfds是异常集合。如果在调用：：select时只关心是否可读，不关心是否可写，那么可把参数writefds设为nullptr。

这3个参数是输入参数，同时也是输出参数。

需要注意的是，这3个集合互不影响，也就是说readfds中的数据只检测可读。对于同一个Socket值，既想检测可读又想检测可写，那么这个值需要分别压入readfds和writefds两个集合。

关键点3: 返回值

调用：：select之后，返回值大于0时，还保留在readfds、writefds、exceptfds三个参数中的值是有事件发生而需要处理的

Socket。在readfds中就是可读的，在writefds中就是可写的，在exceptfds中就是发现了异常。举例说明：

- 在readfds集合中传入{11, 12}。
- 在writefds集合中传入{11, 13, 14}。
- 在exceptfds集合中传入{11, 13}。

参数nfds传入15（14+1），这是最大的值14加1的结果。上面的参数表示，我们关心值为11、12的Socket是否可读，11、13、14是否可写，再检查一下11、13是否有异常。调用：`select`函数之后，这3个集合的数据只会留下有事件的Socket值，即返回readfds的集合可能为{11}，writefds可能为{13}，exceptfds可能没有值，即表示Socket 11有可读数据，Socket 13可以写入数据。

2.1.2 工程源代码

下面用实例代码来说明Select模型的使用方法。工程源代码位于本书02_01_network_select目录中，在这个工程中，我们的目的是：

(1) 以Select模型为基础实现一个有网络收发功能的服务端，收到客户端的数据之后，同时发送给客户端，收多少发多少。

(2) 以Select模式为基础建立一个测试robots工程。robots工程可以指定多个线程，每个线程模拟客户端向服务端发送多条随机长度的数据，完成指定次数后退出。

随机长度的数据可以验证在收发包时的拆分长度是否正确。

虽然本节内容是由：`select`函数引出的，但在本例中，我们抽象出了网络层，重建了数据结构、收发数据的相关逻辑。本节的信息量比起第1章相对较多，在分析代码之前先来看看结果，使用`make-all`脚本进行编译。由于打印数据过多，我们挑选一些重要信息。在服务端打印数据如下：

```
[root@localhost bin]# ./serverd
recv size:433 msg:upqytazmzlsfimoisnqt...
recv size:296 msg:detdwpbndarrsguubcpe...
```

在robots机器人端打印数据如下：

```
[root@localhost bin]# ./robotsd 1 2
send. size:433 msg:upqytazmzlsfimoisnqt...
recv. size:433
send. size:296 msg:detdwpbndarrsguubcpe...
recv. size:296
```

为了便于查看，在robots机器人端开启了1个线程，每个线程发送了两条数据。从上面的数据可以看出，robots发送了两条数据，长度分别为433和296，服务端收到之后，转发了相同的内容。随后，robots收到了正确长度的数据。按`Ctrl+C`组合键可退出服务端进程。

如果想测试多线程的结果，那么可以在启动robots时更改线程数量，如要启动3个线程，命令行为“./robotsd 3 2”，即开启3个线程，每个线程发送两条数据。

本节的源代码采用了面向对象的思想，比之前的例子更加复杂。有两个关键类NetworkConnector和NetworkListen，分别用于网络连接和网络监听，它们都派生于Network基类。图2-1列出了网络库network工程中重要的类以及它们的关系。

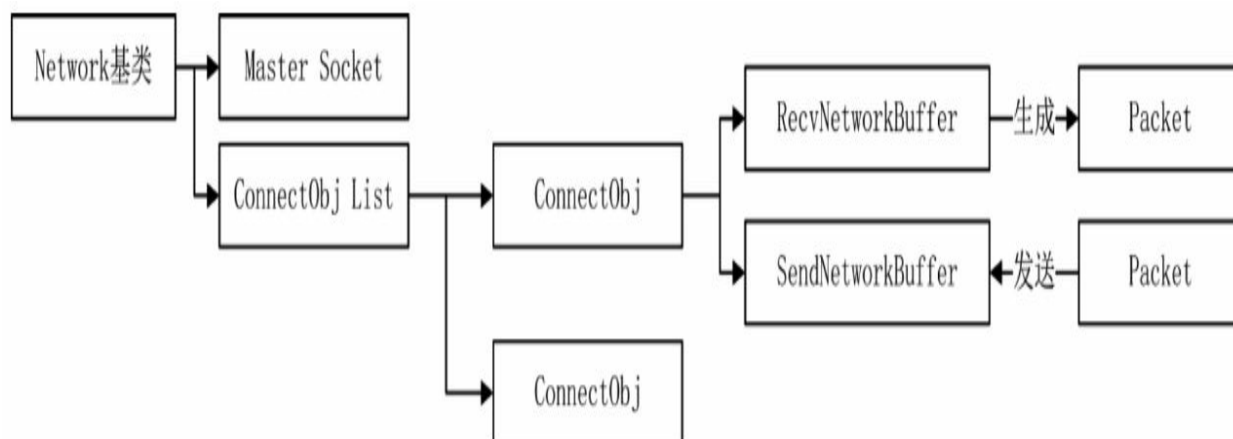


图2-1 网络层数据结构

在前面几个例子中，可以发现监听与连接除了初始化调用不一致以外，在收发数据上是一致的，将这部分功能提出来放在Network基类中。NetworkListen类有它自己的Accept函数，用于接收新的请求。而NetworkConnector类实现Connect连接函数。在面向对象编码时有一个重要的编码规则，即为对象找到共性。不论是监听还是连接类，最初都需要创建一个Socket，为了便于区分，命名为MasterSocket。

每一个网络连接用一个ConnectObj类来维护。在监听端会产生无数个ConnectObj。在发起连接的NetworkConnector类中，目前只有一个ConnectObj实例。

每个ConnectObj都维护着接收缓存和发送缓存，对应图2-1中的RecvNetworkBuffer和SendNetworkBuffer。接收缓存接收到数据之后，将在逻辑层生成Packet类（也称为协议类），供上层使用。而发送缓存发送逻辑层事先准备好的Packet类。客户端发送一个Packet，服务端还

原一个Packet。Packet类是一个非常重要的概念，后面的章节还会详细地介绍它。现在可以这样理解，从网络缓存中得到的二进制数据，从ConnectObj中取出来时已经变成了一个有意义的Packet类。

我们的目标是将网络层的数据锁在Network工程中，对于调用NetworkConnector或者NetworkListen的上层而言，看不到底层的复杂数据，使用Packet来传递数据，有数据到来的时候被封装成Packet类，而发送数据的时候也封装成Packet类。

2.1.3 网络基类：Network

先来看看封装网络功能的Network类，该类的定义位于network库工程的network.h文件中。Network类继承自IDisposable。后面所有类都会继承这个基类，以在退出时方便清理资源。IDisposable类只有一个虚函数。

```
class IDisposable {
public:
    virtual ~IDisposable( ) = default;
    virtual void Dispose( ) = 0;
};
class Network : public IDisposable {
public:
    ...
    bool Select( );
protected:
    // 不论是连接类还是监听类，都有一个_masterSocket
    // 连接时用来存储连接的Socket，监听时存储监听Socket
    SOCKET _masterSocket{ -1 };
    // 对于监听类来说，有无数个ConnectObj类，保存每个连接的数据
    // 对于连接类来说，这里只有一个ConnectObj类
    std::map<SOCKET, ConnectObj*> _connects;
    // 3个fd_set集合，分别用来存储可读、可写、异常的描述符
    fd_set readfds, writefds, exceptfds;
};
```

清楚了定义之后，来看看比较重要的Network::Select函数。该函数调用了系统层面的::select函数。其实现代码如下：

```
bool Network::Select( ) {
    // 初始化所有集合，FD_ZERO 是系统定义的宏，用来清空集合
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);
    FD_ZERO(&exceptfds);
    // 将用于监听或用来连接的Socket放入集合中
    FD_SET(_masterSocket, &readfds);
    FD_SET(_masterSocket, &writefds);
    FD_SET(_masterSocket, &exceptfds);
    // fdmax的作用是保存最大的Socket描述符
```

```

SOCKET fdmax = _masterSocket;
// 遍历当前所有的连接, 将所有当前有效的Socket加入3个集合
// 同时统计最大Socket值
for (auto iter = _connects.begin(); iter != _connects.end(); +
    if (iter->first > fdmax)
        fdmax = iter->first;
    FD_SET(iter->first, &readfds);
    FD_SET(iter->first, &exceptfds);
    if (iter->second->HasSendData()) {
        FD_SET(iter->first, &writefds);
    } else {
        if (_masterSocket == iter->first)
            FD_CLR(_masterSocket, &writefds);
    }
}
// 所有数据收集完成之后调用::select函数
struct timeval timeout;
timeout.tv_sec = 0;
timeout.tv_usec = 50 * 1000;
int nfd = ::select( fdmax + 1, &readfds, &writefds, &exceptfd
if ( nfd <= 0 )
    return true;
// 如果返回值大于0, 就表示在3个集合中一定有数据
// 遍历所有的连接, 看看是否有数据改变
auto iter = _connects.begin( );
while ( iter != _connects.end( ) ) {
    if ( FD_ISSET(iter->first, &exceptfds ) ) {
        // 检查是否有异常, 发现异常后关闭Socket
        iter->second->Dispose();
        delete iter->second;
        iter = _connects.erase(iter);
        continue;
    }
    // 检查是否有数据需要读取
    if ( FD_ISSET(iter->first, &readfds ) ) {
        if ( !iter->second->Recv( ) ) {
            ... // 若失败, 有异常, 则关闭Socket
            continue;
        }
    }
    // 检查是否可以发送数据
    if (FD_ISSET(iter->first, &writefds)) {
        if (!iter->second->Send()) {
            ... // 若失败, 有异常, 则关闭Socket
            continue;
        }
    }
}
++iter;

```

```
    }  
    return true;  
}
```

在这个函数中，将所有Socket值都注册到了集合中，通过调用：`select`来得到有事件发生的集合。读取数据时，不再需要一直不停地执行：`recv`函数。通过底层：`select`函数的返回值来判断是否有读事件发生，当Socket上有可读数据时，调用`ConnectObj::Recv`函数来读取数据。在发送数据时，检测一下是否可写，调用`ConnectObj::Send`函数来发送数据。

在本例中，Socket被封装到`ConnectObj`对象中，也就是说一个`ConnectObj`对象一定对应一个正在通信的Socket，如果Socket断开，这个`ConnectObj`对象就会被销毁。

至于`ConnectObj`类如何接收和发送数据的细节，我们会在后面的章节中讲解。总之，可以将其理解为通过`ConnectObj`类的`Send`和`Recv`这两个函数实现数据的收发。

2.1.4 NetworkListen分析

在Network基类之上建立了一个新类NetworkListen用于服务端的监听。该类定义在network_listen.h文件中。NetworkListen类的定义如下：

```
class NetworkListen :public Network {
public:
    bool Listen(std::string ip, int port);
    bool Update();
protected:
    virtual int Accept();
};
```

关键点1: Listen函数

在NetworkListen类中实现了监听类特有的Listen函数，该函数的主要任务是创建Socket并监听，其流程和前面例子中创建Socket的方式大同小异。

```
bool NetworkListen::Listen(std::string ip, int port) {
    _masterSocket = CreateSocket();
    if (_masterSocket == INVALID_SOCKET)
        return false;
    sockaddr_in addr;
    memset(&addr, 0, sizeof(sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    ::inet_pton(AF_INET, ip.c_str(), &addr.sin_addr.s_addr);
    if (::bind(_masterSocket, (sockaddr*)&addr, sizeof(addr)) <
        std::cout << "::bind failed. err:" << _sock_err() << std::
        return false;
    }
    int backlog = 10;
    if (::listen(_masterSocket, backlog) < 0) {
        std::cout << "::listen failed." << _sock_err() << std::end
        return false;
    }
    return true;
```

```
}
```

在该函数中，将创建的用于监听的Socket存到了MasterSocket变量中，然后调用：`bind`函数进行IP地址与端口的绑定，最后调用了：`listen`函数实现对网络层的监听。

关键点2: `Accept`函数

另一个监听类特有的函数是`Accept`函数，实现如下：

```
int NetworkListen::Accept( ) {
    struct sockaddr socketClient;
    socklen_t socketLength = sizeof( socketClient );
    int rs = 0;
    while ( true ) {
        // 调用::accept获取新连接
        SOCKET socket=::accept( _masterSocket, &socketClient, &socketLength );
        if ( socket == INVALID_SOCKET )
            return rs;
        // 为新建的Socket设置参数
        SetSocketOpt( socket );
        // 新建一个ConnectObj类，用来管理当前连接，同时将obj实例加入队列中
        ConnectObj* pConnectObj = new ConnectObj( this, socket );
        _connects.insert( std::make_pair( socket, pConnectObj ) );
        ++rs;
    }
    return rs;
}
```

值得注意的是，在`Accept`函数中使用了一个`while`来调用底层：`accept`函数，为什么这里要使用一个`while`？要知道，一瞬间可能不止一个连接请求。在这一帧，我们需要取尽所有新连接。

关键点3: `SetSocketOpt`函数

在上面的代码中，创建Socket的时候调用了`SetSocketOpt`函数为Socket增加属性。其代码如下：

```
void Network::SetSocketOpt( int socket ) {
    // 1.端口关闭后马上重新启用
    bool isReuseaddr = true;
    setsockopt( socket, SOL_SOCKET, SO_REUSEADDR, (void*)&isReusea
    // 2.发送、接收timeout
    int netTimeout = 3000; // 1000 = 1秒
    setsockopt( socket, SOL_SOCKET, SO_SNDTIMEO, (void *)&netTimeo
    setsockopt( socket, SOL_SOCKET, SO_RCVTIMEO, (void *)&netTimeo
    ...
    // 3.非阻塞
    _sock_nonblock( socket );
}
```

该函数内主要使用了底层：：setsockopt函数来设置Socket的状态。在Linux系统下，：：setsockopt函数定义在/usr/include/sys/socket.h文件内。函数说明见表2-4与表2-5，其函数原型如下：

```
int setsockopt (int fd, int level, int optname, const void *optval
```

若函数执行成功则返回0，若有错误则返回-1。

表2-4 setsockopt参数表

参 数 名	说 明
fd	Socket 值
level	欲设置的网络层，SOL_SOCKET 表示在套接字级别上设置选项
optname	该参数代表欲设置的选项，具体数据见表 2-5
optval	该参数代表欲设置的值
optlen	该参数为 optval 的长度

表2-5 optname常用值表

参 考 值	说 明
SO_DEBUG	打开或关闭调试模式
SO_REUSEADDR	允许在绑定过程中地址重复使用
SO_SNDBUF	设置送出的暂存区大小
SO_RCVBUF	设置接收的暂存区大小
SO_KEEPALIVE	定期确定连线是否已终止

表2-5只是列出了参数optname的一些常用值。在设置Socket属性时，SO_SNDBUF和SO_RCVBUF用于设置TCP的网络数据缓冲区大小。在本节的例子中并没有设置，而是使用了默认值。其默认值在Linux下可以查看。

```
[root@localhost ipv4]# cat /proc/sys/net/ipv4/TCP_rmem
4096      87380      6291456
[root@localhost ipv4]# cat /proc/sys/net/ipv4/TCP_wmem
4096      16384      4194304
```

这3个值分别是最小值、默认值和最大值。

每个数据包经过TCP层一系列复杂的步骤，最终到达系统接收缓冲区，我们调用底层函数：`recv`就是从这个缓冲中读取数据的。发送数据也是一样，将数据发送到系统发送缓冲区中，就一直在等待分片送出。在网络底层对于发送数据的大小是有限制的，这不属于本书的讨论范围，但我们需要知道，数据在网络中传输时绝不是按照应用程序组织的数据一条一条地发送的。SO_SNDBUF和SO_RCVBUF这两个属性一般可以保持默认值，除非对于协议数据有特别的需要，可以使用：`setsockopt`函数来修改这两个属性的值。

关键点4: Update函数

Update函数每帧调用，本书中将它称为帧函数。在其中调用了基类的Network::Select函数，对进程上的所有网络连接和用于监听的MasterSocket进行检查，查看每个Socket上是否有事件发生。

```
bool NetworkListen::Update( ) {
    // 调用Select函数对所有的Socket进行检测
    bool br = Select( );
    // 如果发现监听的Socket有可读消息，那么一定是有一个连接
    if (FD_ISSET(_masterSocket, &readfds)) {
        Accept( );
    }
    return br;
}
```

基类的Network::Select函数会对所有连接状态下的Socket收集读写事件，NetworkListen类只需要对底层::select函数返回值中的readfds集合进行检查，监听MasterSocket上是否有事件发生。一旦有事件发生，就可能有一个或多个连接请求。此时，需要做的是调用NetworkListen::Accept函数，若成功，则新的网络连接被建立。

得益于基类的支持，整个NetworkListen类看上去非常简单，对外开放的只有两个函数：一个是Listen函数；另一个是每帧调用的Update函数。

2.1.5 Server流程详解

现在监听类NetworkListen已经准备好了，来看看它的使用方式。为了使用它，在服务端生成了一个Server类。Server类定义在server工程中，定义如下：

```
class Server : public NetworkListen {
public:
    bool DataHandler();
protected:
    int Accept() override;
    ...
};
```

虽然在NetworkListen类中提供了收发数据，但作为一个基础库，NetworkListen是不会进行具体的数据处理的。Server类的作用就是实现这一功能，我们来看看它是如何处理数据的：

```
bool Server::DataHandler() {
    for (auto iter = _connects.begin(); iter != _connects.end(); ++
        ConnectObj* pConnectObj = iter->second;
        HandlerOne(pConnectObj);
    }
    ...
    return true;
}

void Server::HandlerOne(ConnectObj* pConnectObj) {
    // 收到客户端的消息，马上原样发送出去
    while (pConnectObj->HasRecvData()) {
        Packet* pPacket = pConnectObj->GetRecvPacket();
        if (pPacket == nullptr) {
            break;        // 数据不全，下帧再检查
        }
        // 原样转发
        std::string msg(pPacket->GetBuffer(), pPacket->GetDataLength());
        std::cout << "recv msg:" << msg.c_str() << std::endl;
        pConnectObj->SendPacket(pPacket);
        ...
    }
}
```

在前面讲过二进制数据留在了Network底层，逻辑层出去的数据是Packet数据。在Server::HandlerOne函数中，向ConnectObj类请求Packet包。对于Server工程而言，它并不知道底层有多么复杂，唯一的动作就是不断询问ConnectObj类是否接收到了Packet数据。再看看main函数对Server类的调用：

```
int main(int argc, char *argv[]) {
    ...
    Server server;
    // 调用监听函数将底层创建的Socket绑定，监听全都隐藏起来，只有一个看起来异
    if ( !server.Listen( "127.0.0.1", 2233 ) )
        return 1;
    while ( isRun ) {
        if ( !server.Update( ) ) // 底层数据更新
            break;
        // 通过更新网络层的数据，数据从网络底层缓冲区中加载到了我们的缓冲区中
        // 使用DataHandler函数对这些数据进行处理
        server.DataHandler( );
    }
    server.Dispose( ); // 退出前清理 NetworkListen
    return 0;
}
```

在上面的源代码中，调用Server::Listen打开了监听，隐藏了网络层的复杂操作。其后要做的是不断地检测底层网络数据的事件，这个功能由Server::Update来完成。Server::Update也就是NetworkListen::Update。回顾一下这个函数做了些什么，它收集了当前内存中所有处理连接状态的Socket，将这些Socket放到3个集合中，由::select函数去查看它们在当前帧下是否有事件发生，如果有读就读，有写就写。同时，内存中还有一个用于监听端口的MasterSocket，一旦发现监听端口有数据，就生成一个新的Connectobj并保存在内存中，以便在下一帧可以检查它的读写事件。

底层Update函数实现底层的数据收发，而上层逻辑在有数据时从读缓冲区RecvNetworkBuffer中取得Packet数据，同时将需要发送的数据组织为Packet放到发送缓冲区SendNetworkBuffer中，这样就借由Packet类为中转实现了底层数据与上层逻辑的隔离。

需要说明的是，在第N帧发送数据写入SendNetworkBuffer时，它的真正发送时间在第N+1帧。在第N+1帧，调用Update函数，若遍历ConnectObj对象时发现其中一个实例的发送缓冲区中有数据，则取出数据并进行真正的发送。图2-2清楚地展示了服务端的收发数据流程。

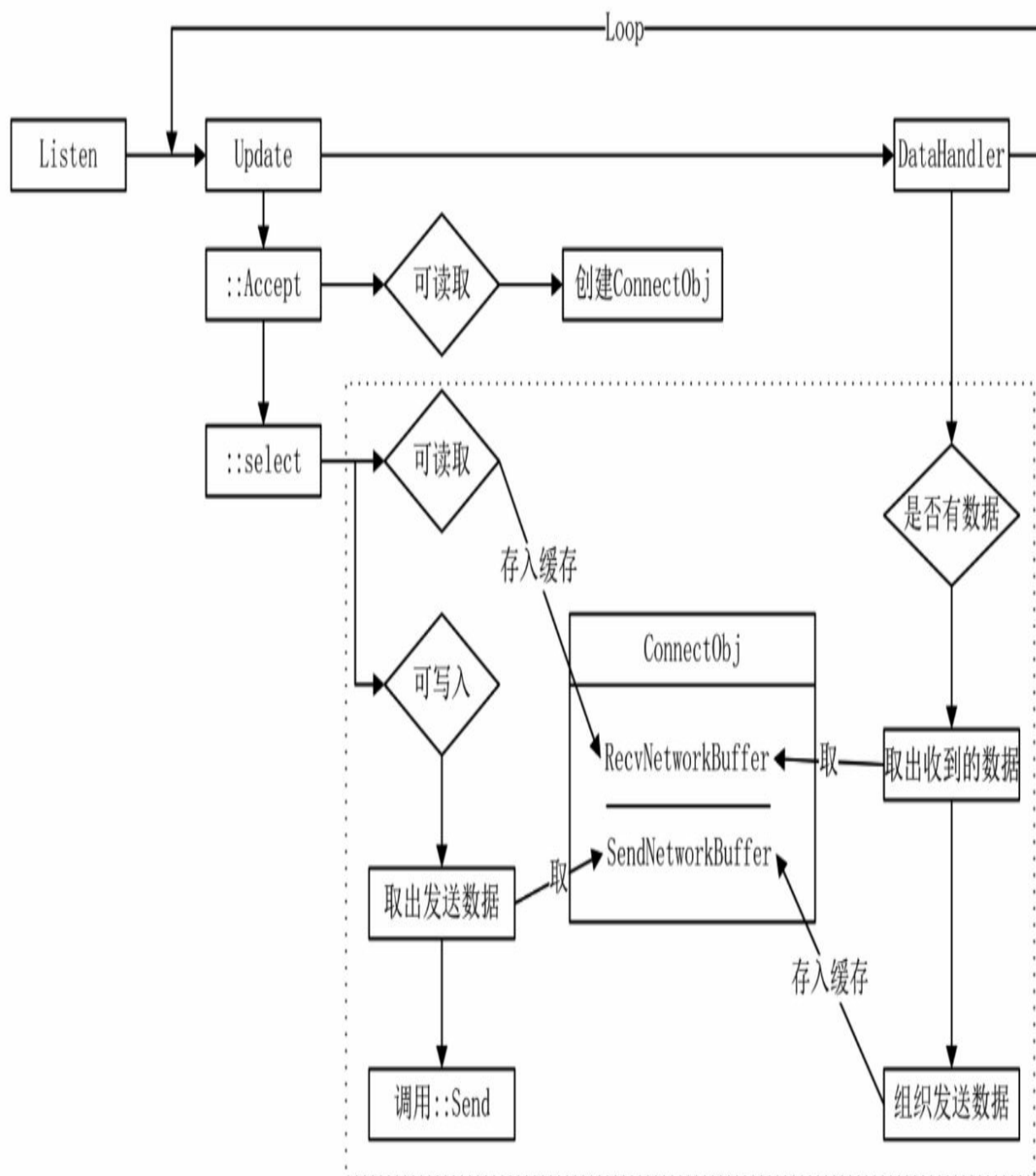


图2-2 Select模式下服务端的流程图

2.1.6 NetworkConnector 分析

客户端调用的是连接类NetworkConnector，其定义在network_connector.h文件中。

```
class NetworkConnector : public Network {
public:
    bool Connect(std::string ip, int port);
    bool Update();
    ...
}
```

先来看看连接类中的两个核心函数：

(1) Connect函数，对给定的IP地址和端口进行连接。

(2) Update函数，每帧调用，和NetworkListen监听类的原理一致。该函数调用基类的Network::Select函数对网络连接进行检查。

```
bool NetworkConnector::Connect(std::string ip, int port) {
    _ip = ip;
    _port = port;
    _masterSocket = CreateSocket();
    if (_masterSocket == INVALID_SOCKET)
        return false;

    sockaddr_in addr;
    memset(&addr, 0, sizeof(sockaddr_in));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(_port);
    ::inet_pton(AF_INET, _ip.c_str(), &addr.sin_addr.s_addr);
    int rs = ::connect(_masterSocket, (struct sockaddr *)&addr, s
    if (rs == 0) {
        // 成功
        ConnectObj* pConnectObj = new ConnectObj(this, _masterSo
        _connects.insert(std::make_pair(_masterSocket, pConnectO
    }
    return true;
}

bool NetworkConnector::Update() {
```

```

const bool br = Select();
if (!IsConnected()) {
    // 有异常出现
    if (FD_ISSET(_masterSocket, &exceptfds)) {
        Dispose(); // 异常, 关闭当前Socket, 重新连接
        Connect(_ip, _port);
        return br;
    }
    if (FD_ISSET(_masterSocket, &readfds) || FD_ISSET(_masterS
        int optval = -1;
        socklen_t optlen = sizeof(optval);
        const int rs = ::getsockopt(_masterSocket, SOL_SOCKET,
        if (rs == 0 && optval == 0) {
            ConnectObj* pConnectObj = new ConnectObj(this, _ma
            _connects.insert(std::make_pair(_masterSocket, pCo
        } else {
            Dispose(); // 异常, 关闭当前Socket, 重新连接
            Connect(_ip, _port);
        }
    }
}
return br;
}

```

代码中有两处调用创建ConnectObj的代码, 第一次是在Connect函数中, 如果成功, 就创建ConnectObj对象。但情况并不总是这么理想, 在某些情况下, 调用底层::connect函数并不会马上确认连接成功, 从发起一个连接到确认连接成功会有一个时间间隔。在等待时间里, 如果该Socket上有事件发生而没有任何错误, 就表示该连接成功。

一般情况下, 监听类监听到新的连接时会触发一个读操作, 连接类连接成功之后会触发一个写操作。

但是, 即使我们收到了写操作, 也必须确认一下是否出错。在没有出错的情况下, 才认为是真正成功连接。其中, 检查Socket状态的函数是::getsockopt, 它和我们前面讲的::setsockopt是一对函数: 一个用于取值, 另一个用于设值。

在上面的Update函数中, 调用::getsockopt函数从Socket层面取出错值, 若没有出错, 则连接成功。连接成功之后会生成一个ConnectObj对象, 用于数据收发。

2.1.7 测试流程详解

为了验证，server工程准备了一个模拟客户端操作的robots工程用于测试。这个工程使用了线程，每个线程负责处理一个Client类，Client类继承自NetworkConnector类，是一个连接类。

```
class Client :public NetworkConnector {
public:
    explicit Client(int msgCount, std::thread::id threadId);
    void DataHandler();
    ...
}
```

单个Client类的工作原理类似于服务端的Server类，Client类承担了具体的数据处理工作。在本例中，数据的发起是由Client类开始的。

客户端每次向服务端发送的数据都是随机的，范围为10~512。客户端使用网络基类的方式与服务端是一模一样的，图2-3清楚地画出了客户端数据的流程图。具体发送的数据在Client类的DataHandler函数中，实现如下：

```
void Client::DataHandler() {
    if (_isCompleted)
        return;
    if (!IsConnected())
        return;
    if (_index < _msgCount) {
        // 发送数据
        if (_lastMsg.empty()) {
            _lastMsg = GenRandom();
            Packet* pPacket = new Packet(1);
            pPacket->AddBuffer(_lastMsg.c_str(), _lastMsg.length());
            SendPacket(pPacket);
            delete pPacket;
        } else {
            if (HasRecvData()) {
                Packet* pPacket = GetRecvPacket();
                if (pPacket != nullptr) {
                    const std::string msg(pPacket->GetBuffer(), pP
                    std::cout << "recv msg. size:" << pPacket->Get
```

```

        ...
        delete pPacket;
    }
}
}
...
}

```

在上面的代码中，进出Network基类的唯一数据是Packet类，收发都是这个类，所以发送数据时创建了Packet类。发送的数据是从GenRandom函数中取得的一个随机的字符串。

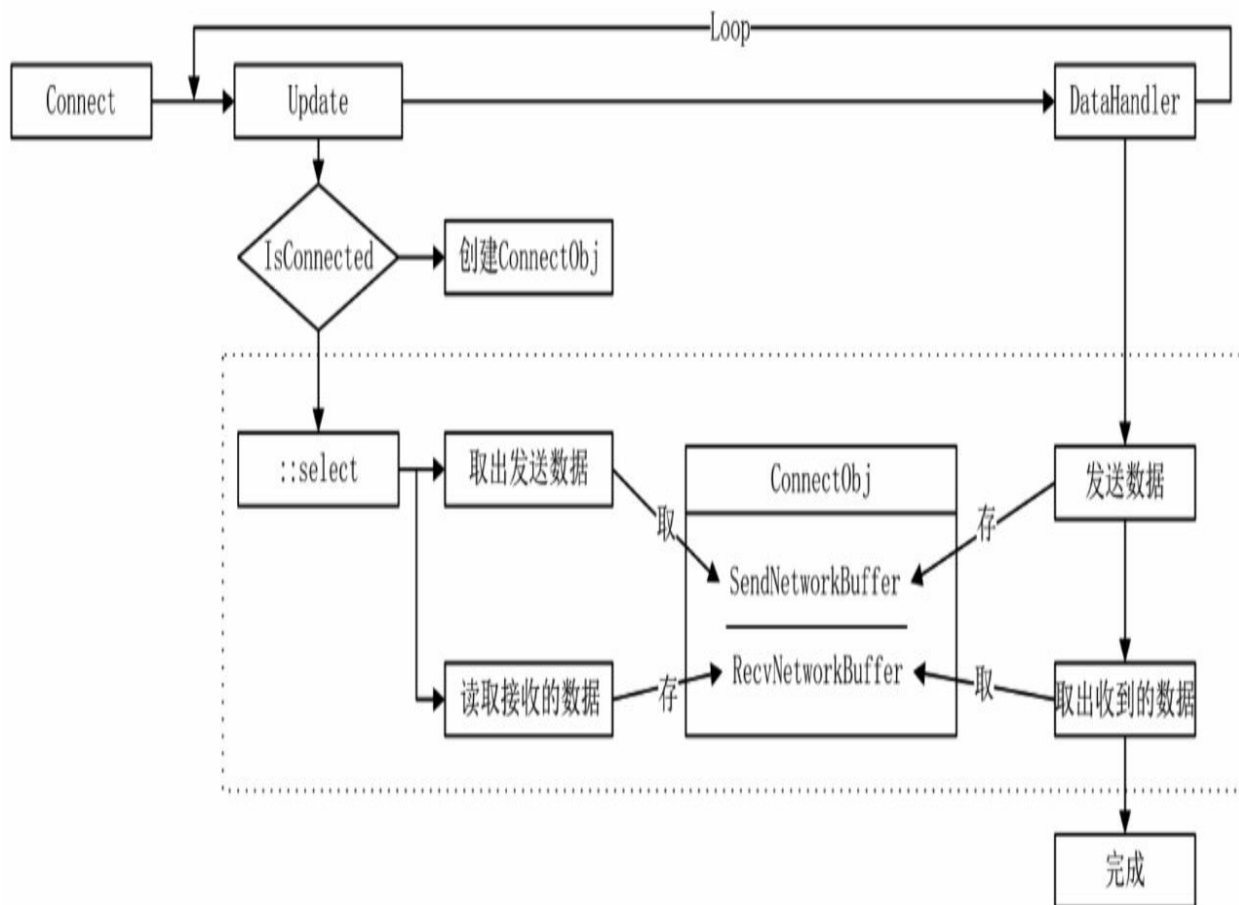


图2-3 Select模式下单个Client流程图

工程robots采用了多线程，线程类是由ClientThread类来负责的，具体实现可参见源代码，这里就不再过多讨论了。

现在从整体上来理解一下服务端与客户端的逻辑。在这个例子中创建了一个监听类和一些连接类，监听端或连接端为每一个Socket生成一个ConnectObj，而ConnectObj实现了数据的收发。怎么收发的，上层逻辑并不需要知道，只是在上层逻辑需要的时候，这些收发的数据会变成一个个Packet类来使用。

这是第一个面向对象的例子，在本书随后的所有例子中，网络底层逻辑都会沿用这一编程思路。在后续的章节中因为需求不同，代码会有一些变动，但其思路基本一致。因此，本章花点篇幅来讲解几个主要类的工作原理，主要是Packet、ConnectObj、RecvNetworkBuffer、SendNetworkBuffer四个类。我们首先要了解的是ConnectObj类。

下面深入看看ConnectObj到底是如何收发数据的，又是如何得到Packet数据的。

2.1.8 ConnectObj分析

为了突出网络模型的代码，在前面的章节中没有阐述数据是如何发送与接收的。只是简单使用了：`::Send`发送和：`::Recv`接收。本小节将对其背后的逻辑进行梳理。每当有一个连接被建立时，无论是在监听端还是连接端都有一个ConnectObj类被生成。下面是ConnectObj的类定义：

```
class ConnectObj : public IDisposable {
    ...
    bool HasRecvData( ) const;                // 是否有接收
    Packet* GetRecvPacket( ) const;           // 将接收到的
    bool HasSendData( ) const;                // 是否需要发
    void SendPacket( Packet& packet ) const;   // 发送Packet中的
protected:
    const int _socket;
    RecvNetworkBuffer* _recvBuffer{ nullptr };
    SendNetworkBuffer* _sendBuffer{ nullptr };
}
```

在这个类中维护了两个Buffer：`SendNetworkBuffer`用于发送数据，`RecvNetworkBuffer`用于接收数据。两个收发Buffer的主要功能是缓存从网络收到的数据以及缓存发送到网络底层的数据。从逻辑层发送和接收Packet的函数实现如下：

```
Packet* ConnectObj::GetRecvPacket() const {
    return _recvBuffer->GetPacket();
}
void ConnectObj::SendPacket(Packet& packet) const {
    _sendBuffer->AddPacket(packet);
}
```

上面的代码中，ConnectObj类调用了缓冲区的函数。为什么我们需要缓冲区？这是不少人想问的问题。

在前面为Socket设置属性的时候提到了缓冲区，`SO_SNDBUF`和`SO_RCVBUF`设置了系统中TCP网络数据缓冲区的大小，这个缓冲区与程序

中的缓冲区有什么区别呢？

当有数据从网络到来时，数据是被系统写入SO_RCVBUF这个缓冲区的，这个缓冲区有大小限制，字节占满缓冲区就会覆盖之前的数据。因此，需要应用程序有一个自己的读缓冲区，有数据时马上转存这些在系统缓冲区中的数据，以免它被后来的数据覆盖。

SO_SNDBUF是写缓冲区，是上层应用向网络发送数据的系统层缓冲区。举例说明，有100KB的数据需要发送，假如底层缓冲区最大为10KB，虽然需要发送100KB，但是调用：`send`函数会返回10KB，函数的返回值是已发送的长度，也是系统发送缓冲区的剩余长度。剩下90KB的数据需要存放在应用程序的缓冲区中，等待下一次写事件的到来。

当我们发送的数据过于频繁或者系统缓冲区较小时，正确的处理方式应该是不断向底层询问是否可以写入，分批将后面的数据依次写入。

以上就是读写缓冲区的意义所在。关于缓冲区的内容极为枯燥，但它又极为重要，重要到错一个字节的数据，整个程序就会全面瓦解。下面来详细看看这些缓冲区的工作原理。

2.1.9 Buffer分析

无论是SendNetworkBuffer还是RecvNetworkBuffer，都基于NetworkBuffer类，而NetworkBuffer继承自Buffer类。在Buffer类中，变量char*_buffer用来存储数据，当缓冲区满时，其长度会自动增加。其基类定义在base_buffer.h文件中。Buffer类的定义如下：

```
class Buffer :public IDisposable {
public:
    virtual unsigned int GetEmptySize();
    void ReAllocBuffer(unsigned int dataLength);
protected:
    char* _buffer{ nullptr };
    unsigned int _beginIndex{ 0 }; // buffer数据的开始位与结束位
    unsigned int _endIndex{ 0 };
    unsigned int _bufferSize{ 0 }; // 总长度
}
```

变量BeginIndex和EndIndex确定了数据处于这个缓冲区的什么位置。Buffer变量是这个缓冲区的头指针，BufferSize是缓冲区的长度，这个长度不是一成不变的，在不够用时会进行扩容操作，也就是Buffer::ReAllocBuffer函数的功能。在使用Buffer类时可能会有两种场景：

(1) 从一个二进制数据中读取一段数据存入Buffer中。例如后面要讲到的Packet类，对于Packet来说，读取一旦生成就不会改变。也就是说，一个Packet中的Buffer开始有5字节的长度，那么一直是5字节的长度，直到Packet被销毁或者Packet被重用，数据才被抹去。

(2) 前面提到的ConnectObj中的缓冲区，比第一种情况复杂得多，它的数据总是源源不断地到来，再源源不断地取出。数据长度在每一帧都可能不一样。

下面介绍这两种处理方式的不同，本书把这两种情况称为“一般缓存”和“环形缓存”。

关键点1：一般缓存

对于一般缓存来说，BeginIndex一定是从0开始的，并且不会变更。BeginIndex表示读取数据的开始，而EndIndex则是写入数据的开始，结合图2-4进行理解。

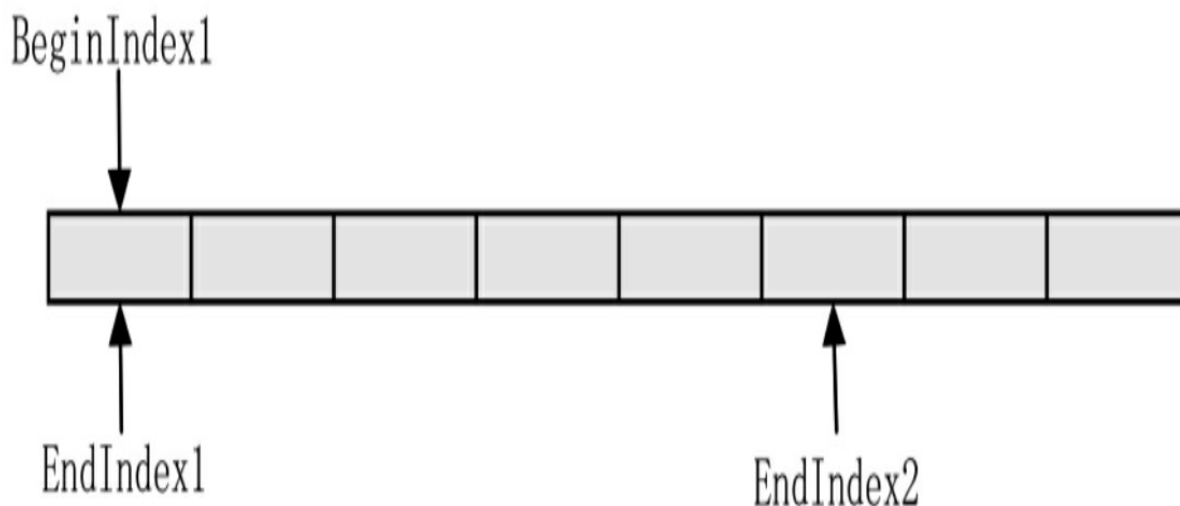


图2-4 一般缓存

一开始，BeginIndex和EndIndex都位于缓冲区的开始处，当收到数据之后，EndIndex开始移动，它移到了EndIndex2的位置，也就是说从EndIndex到BeginIndex之间所有的字节都是有数据的。举个例子，从网络层收到一个100字节长度的协议，这100字节会完全复制到Packet的缓冲区中供上层逻辑使用。当发现 $EndIndex > BeginIndex$ 的时候，就可以认为一定有数据等待读取。

关键点2：环形缓存

比起Packet，NetworkBuffer要复杂得多。因为它既有写又有读。数据读完了还需要抹去，以便后来的数据有地方可以写入。在环形数据中，EndIndex是可写数据的开始点，BeginIndex是可读数据的开始点。

图2-5可以形象地说明这一流程，最开始的时候，BeginIndex和EndIndex都位于缓冲区的最开始，连续收到好几条数据之后，EndIndex发生了移动。这时，上层逻辑发现有数据到来了，从Buffer缓冲区中取出一个协议，取出的操作导致BeginIndex发生了移动。有效的数据段变

成了从BeginIndex2到EndIndex2。如果按这个逻辑不停地写、不停地读，BeginIndex和EndIndex最终就会移动到这个缓冲区的尾部。这就有了环形缓存存在的必要。

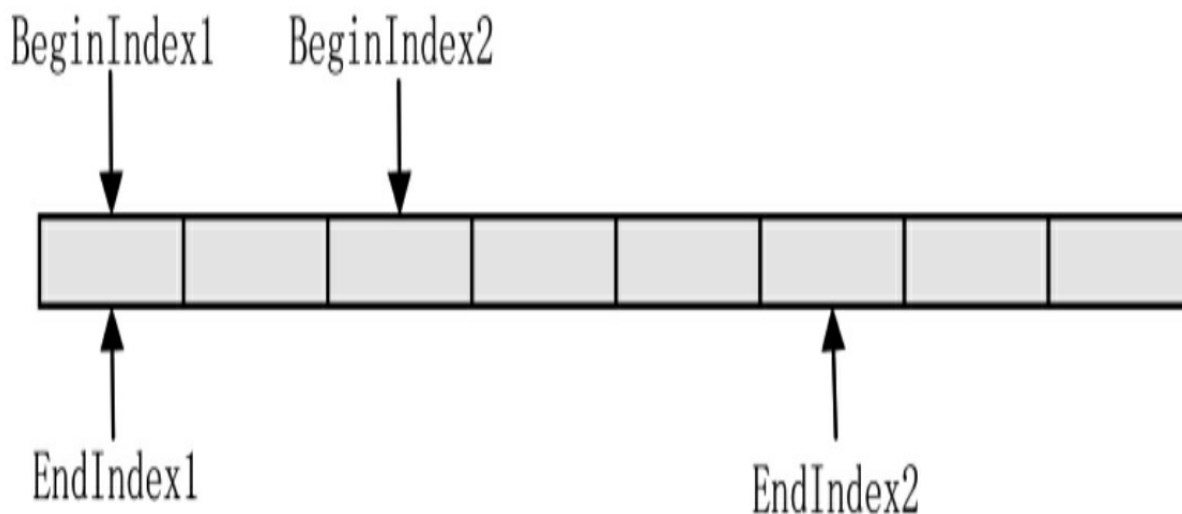


图2-5 环形缓存

当BeginIndex移动到尾部的时候，其实缓冲区的前半部分是空的，没有被使用，可以让整个Buffer形成一个环，让读写指针回到Buffer的最开始。对于环形数据来说，我们需要注意以下几种情况：

(1) BeginIndex在前，EndIndex在后，这时EndIndex-BeginIndex为有效数据，如图2-6左侧的上半部分所示。

(2) EndIndex在前，BeginIndex在后，这时有效数据被拆分成了两部分，一部分在尾部，一部分在头部，如图2-6左侧的下半部分所示。

(3) 如果BeginIndex和EndIndex指向同一位置，如图2-6右侧所示，就有两种情况：一种情况是完全没有数据；另一种情况是数据全满。EndIndex可能已经绕过尾部，从头部又回到了BeginIndex位置，这是一种极端情况。

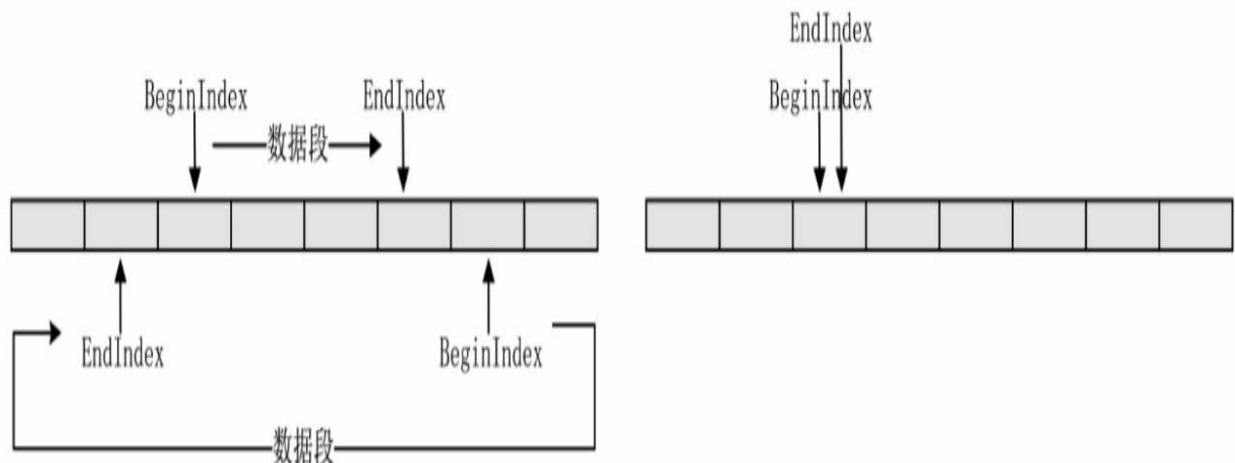


图2-6 环形缓存下标位置

在环形缓存中，边界值是一个十分敏感的问题，不能有一个字节的偏差，在使用和计算时要特别小心。当然，网络数据缓存的方法不止环形缓存这一种方式。这里有另一种方法可供参考，这种方法可分成以下步骤：

(1) 做一个char*的缓存池。例如，先缓存100组长度为1024的Buffer，不够用时再创建。

(2) 写入数据时，从缓存池中取一个长度为1024的Buffer将数据写入，直到写满1024字节之后，再从缓存池中取出一个新的Buffer继续写。将这些有数据的Buffer存到一个list中。

(3) 取出数据时，遍历list，每读完一个1024长度的Buffer，将它放回缓存池中。

该方法不用处理环形数据的重用问题，比较简单直观，也不用考虑空间不够的问题，因为缓存池是无限大的，不够就创建。其缺点在于不停地做取出和放回的操作。

回到例子中，不论是环形缓存还是一般缓存，空间不够时会自动增加长度。相关代码在Buffer::ReAllocBuffer中，该函数是一个内存拷贝函数，细节就不再多讲了。

2.1.10 RecvNetworkBuffer分析

下面看看Network类是如何处理环形缓存中的数据。首先从RecvNetworkBuffer类开始，这个类定义在network_buffer.h文件中。环形缓存要比一般缓存多一个变量，即有效数据的长度。

NetworkBuffer类的定义如下：

```
class NetworkBuffer : public Buffer {
...
protected:
    // 在环形缓存中，极端情况下_endIndex可能与_beginIndex重合
    // 重合时有两种可能：一种是没有数据，另一种是满数据
    unsigned int _dataSize; // 有效数据
}
```

关键点1：写入数据

前面讲过在：：select函数调用之后，如果一个Socket有数据就会触发一个读事件，当这个事件发生之后会调用ConnectObj：：Recv函数。下面是它的实现：

```
bool ConnectObj::Recv() const {
    char *pBuffer = nullptr;
    while (true) {
        // 总数据空间不足一个Packet头的大小时，扩容
        if (_recvBuffer->GetEmptySize() < (sizeof(PacketHead) + si
            _recvBuffer->ReAllocBuffer());
        }
        // 从缓冲区中取得写位置存于pBuffer中，返回可写长度
        const int emptySize = _recvBuffer->GetBuffer(pBuffer);
        const int dataSize = ::recv(_socket, pBuffer, emptySize, 0
        if (dataSize > 0) {
            // 从底层读到dataSize长度的数据写入缓冲区中
            // 修改缓冲区的下标位置
            _recvBuffer->FillDate(dataSize);
        } else if (dataSize == 0) {
            return false;
        } else {
```

```

        const auto socketError = _sock_err();
        ...
        return false;
    }
}

```

分析上面的代码，底层：`recv`函数的参数中需要一个缓存的地址，其目的是将网络底层的数据传到这个缓存，这里直接将`RecvNetworkBuffer`类中的缓存传递给它，同时给了一个可用长度。需要注意这个长度并不等于缓存的最大长度，因为缓存中可能还有数据没有来得及读取。`recv`函数调用之后会返回一个长度，该长度是本次操作从网络底层写入`RecvNetworkBuffer`中的数据的长度。得到已写长度之后调用了`FillDate`函数，这个函数只有一个用途，就是修改`EndIndex`的位置。`EndIndex`的位置向后偏移，表示偏移位前的字节已经有数据了。

当然这是空间足够的时候，如果空长度只有两个字节，而需要有4字节的写入，应该如何处理呢？当空长度不够时需要注意两种情况，如图2-7所示。如果只是单纯地以总长度减去`EndIndex`的值，对于图2-7的左图来说，这是一个真实长度，而对于图2-7的右图来说，总长度减去`EndIndex`的值并不是真实的空长度，可写的空间被分成了前后两段。



图2-7 缓存数据读取

因此，在`RecvNetworkBuffer`类中提供了两个函数：`GetBuffer`提供可写的地址，而`GetWriteSize`提供可写的长度。

```

int RecvNetworkBuffer::GetBuffer(char*& pBuffer) const {
    pBuffer = _buffer + _endIndex;
    return GetWriteSize();
}

```

```

unsigned int NetworkBuffer::GetWriteSize() const {
    if (_beginIndex <= _endIndex) {
        return _bufferSize - _endIndex;
    } else {
        return _beginIndex - _endIndex;
    }
}

void NetworkBuffer::FillDate(unsigned int size) {
    _dataSize += size;
    // 移动到头部
    if ((_bufferSize - _endIndex) <= size) {
        size -= _bufferSize - _endIndex;
        _endIndex = 0;
    }
    _endIndex += size;
}

```

不论是环形缓存还是一般缓存，向缓冲区中写入数据时都是从EndIndex位置开始的。BeginIndex是读取位置，EndIndex是写入位置，这是恒定不变的。虽然写入位置是固定的，但是长度有差异。如图2-7的右图所示，如果写入4字节，那么总空间正好有4字节，但尾部却只有2字节。这时，只能按2字节写入，函数GetWriteSize返回的是2。写入2字节后，FillDate函数会调整EndIndex的位置到最开始，再次调用GetWriteSize函数时还有可用空间，这时还可以继续写入。

在使用ConnectObj::Recv()函数处理数据时，用while循环写入，将数据全部写完。在开始写之前判断可写空间的长度是否大于一个协议头的长度，如果不够，就需要扩容一次。以图2-7左图的数据为例，写入2字节之后，空间已经被占满，如果要写入4字节数据，第2个while时就会自动扩容。

关键点2：读取数据

现在网络层的数据被写入了RecvNetworkBuffer中，下一步如何将这些在缓冲区中的二进制数据变成有意义的结构呢？这里我们要讨论协议格式。

本书中自定义的协议格式是“总长度+PacketHead+协议体”，我们将这些数据封装在Packet类中。关于协议部分，在后面的章节会有详细的阐述，总之就是每次从客户端发来的数据是有格式的，格式分成了3

段，将这3段数据放在Packet类中。注意，Packet类与ConnectObj类是不同的，Packet类是ConnectObj类接收的一个又一个数据。

```
using TotalSizeType = unsigned short;
Packet* RecvNetworkBuffer::GetPacket() {
    // 数据长度是否够一个协议体的总长度
    if (_dataSize < sizeof(TotalSizeType)) {
        return nullptr;
    }
    // 1. 读出整体长度
    unsigned short totalSize;
    MemcpyFromBuffer(reinterpret_cast<char*>(&totalSize), sizeof(T
    // 协议体长度不够，等待
    if (_dataSize < totalSize) {
        return nullptr;
    }
    RemoveDate(sizeof(TotalSizeType));
    // 2. 读出PacketHead
    PacketHead head;
    MemcpyFromBuffer(reinterpret_cast<char*>(&head), sizeof(Packet
    RemoveDate(sizeof(PacketHead));
    // 3. 读出协议
    Packet* pPacket = new Packet(head.MsgId);
    const auto dataLength = totalSize - sizeof(PacketHead) - sizeo
    while (pPacket->GetTotalSize() < dataLength) {
        pPacket->ReAllocBuffer();
    }
    MemcpyFromBuffer(pPacket->GetBuffer(), dataLength);
    pPacket->FillData(dataLength);
    RemoveDate(dataLength);
    return pPacket;
}
```

在上面的代码中，按照格式依次取出数据并拼接到Packet类中，其中有两处返回nullptr都是因为长度不够。当前缓冲区的数据大于我们需要取出的协议体，才认为一个完整的协议已经到达。不要以为一条协议一定是同一时间到达的，它可能在网络底层就被分片了。如果没有收到一个完整的协议，就继续等待。

数据取完之后，调用RemoveDate函数抹去已读数据，它和FillData是一对函数，FillData操作EndIndex，标记EndIndex之前的数据是有效数据，而RemoveDate是移动BeginIndex，数据已经被读取就无效了，

BeginIndex的新位置才是有效数据的开始位置。

读取数据时，有极大的可能遇到一块数据一部分在头，一部分在尾，这时需要特别处理。复制完了尾部的数据，还要绕到头部继续复制数据：

```
void RecvNetworkBuffer::MemcpyFromBuffer(char* pVoid, const unsigned
    const auto readSize = GetReadSize());
    if (readSize < size) {
        // 1.复制尾部数据
        ::memcpy(pVoid, _buffer + _beginIndex, readSize);
        // 2.复制头部数据
        ::memcpy(pVoid + readSize, _buffer, size - readSize);
    } else {
        ::memcpy(pVoid, _buffer + _beginIndex, size);
    }
}
```

现在我们对网络数据的接收流程有了更深层次的理解。在之前的流程图中，标记为取出或存入数据时，其实就是对缓存中的下标进行移动，从而实现数据的复制。

对于RecvNetworkBuffer缓存来说，底层：`select`函数一直在监听是否有数据可以写入，一旦发现有新的数据，就将数据复制到缓存中，并操纵EndIndex的移动。而逻辑层则一直在操纵BeginIndex，取出数据，生成便于逻辑层使用的协议Packet。如图2-8所示，函数Select操纵EndIndex写入数据，在ConnectObj中调用GetPacket函数，操纵BeginIndex不断地读取数据。

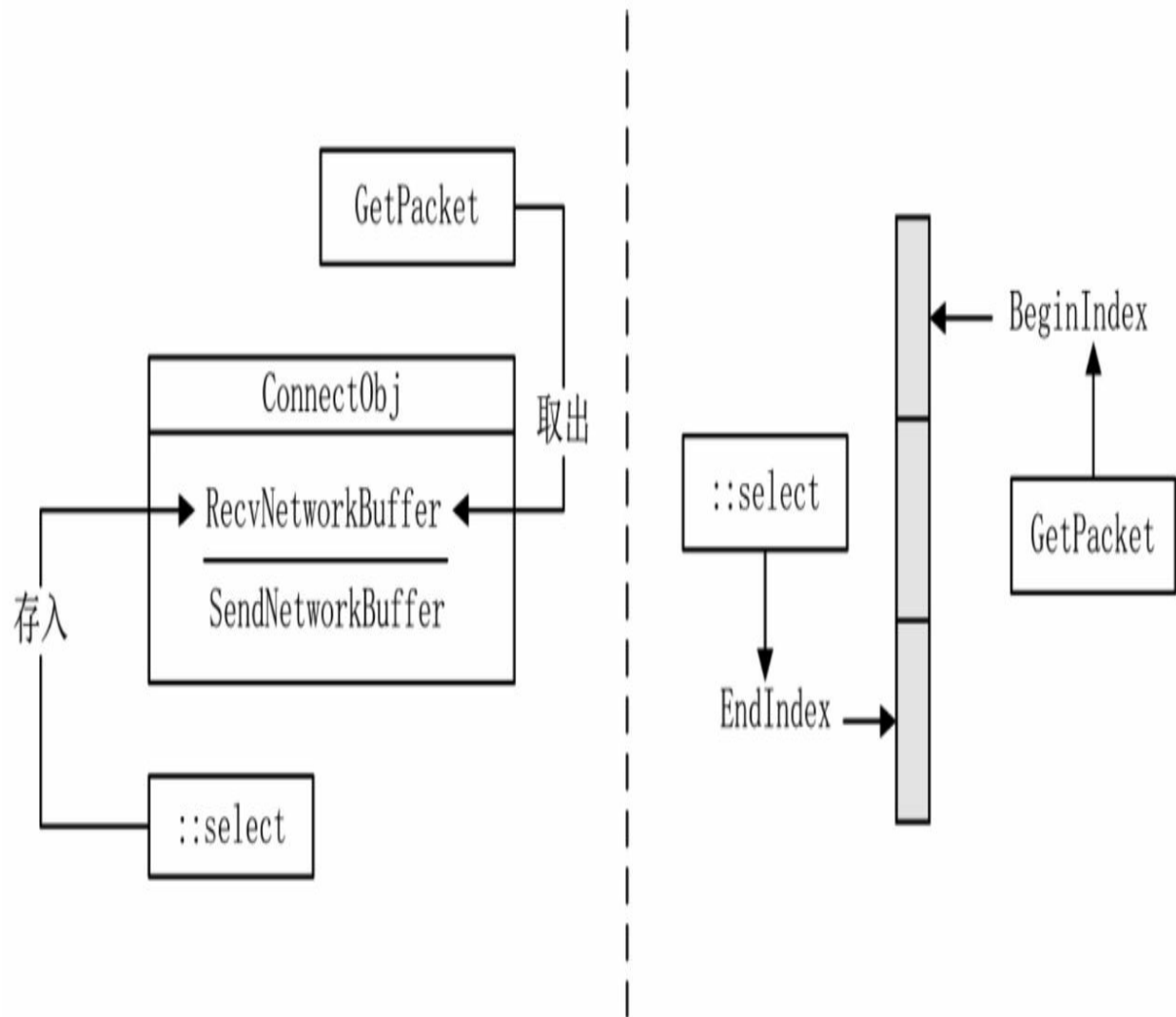


图2-8 ConnectObj接收数据的流程

2.1.11 SendNetworkBuffer分析

相对来说发送数据比接收数据更容易理解一些。要发送一个数据之前，必定要创建一个Packet，在发送缓冲区中，Packet类被序列化为二进制。首先写入的是它的协议头，其次是协议体。以下是相关函数：

```
void ConnectObj::SendPacket(Packet* pPacket) const {
    _sendBuffer->AddPacket(pPacket);
}

void SendNetworkBuffer::AddPacket(Packet* pPacket) {
    const auto dataLength = pPacket->GetDataLength();
    TotalSizeType totalSize = dataLength + sizeof(PacketHead) + si
    // 长度不够，扩容
    while (GetEmptySize() < totalSize) {
        ReAllocBuffer();
    }
    // 1.整体长度
    MemcpyToBuffer(reinterpret_cast<char*>(&totalSize), sizeof(Tot
    // 2.头部
    PacketHead head;
    head.MsgId = pPacket->GetMsgId();
    MemcpyToBuffer(reinterpret_cast<char*>(&head), sizeof(PacketHe
    // 3.数据
    MemcpyToBuffer(pPacket->GetBuffer(), pPacket->GetDataLength())
}
```

当需要发送协议时组织一个Packet，调用ConnectObj::SendPacket就可以发送出去。和接收数据一样，ConnectObj将这个功能转移到了SendNetworkBuffer类中。

而SendNetworkBuffer所做的事情只是将这个Packet类按3段格式序列化为二进制，放到发送缓冲区中。从上面的代码可以看到，先将协议头放进去，再放入了协议体。与接收缓冲区一样，在这个过程中调整了EndIndex的位置。数据被放到缓冲区后就是待发送状态，真正的发送还是在::select函数中。在::select函数中调用了ConnectObj类的Send函数，真正向系统发送缓存写入数据，该函数的实现如下：

```
bool ConnectObj::Send() const {
    while (true) {
```

```

char *pBuffer = nullptr;
const int needSendSize = _sendBuffer->GetBuffer(pBuffer);
// 没有数据可发送
if (needSendSize <= 0) {
    return true;
}
const int size = ::send(_socket, pBuffer, needSendSize, 0)
if (size > 0) {
    _sendBuffer->RemoveData(size);
    // 下一帧再发送
    if (size < needSendSize) {
        return true;
    }
}
if (size == -1) {
    const auto socketError = _sock_err();
    return false;
}
}
}

```

再梳理一下这个流程，当逻辑层需要发送数据时，会将其封装成Packet对象，并将这个Packet对象压入SendNetworkBuffer缓存中，这时并没有真正地发送数据，只是待发送状态，在Network对ConnectObj进行写入检查时，发现有数据才真正地发送数据。

为了对环形缓存做一个测试，本节的例子中新建了一个名为test的工程，在默认长度只有10字节的情况下，对环形缓存数据的扩容和边界值做了一些测试，测试时打开Network库中的TestNetwork宏重新编译。本书中不再对测试代码做出分析，有兴趣的读者可以自行学习。

2.1.12 Packet分析

之前已经对ConnectObj、RecvNetworkBuffer、SendNetworkBuffer三个类有了一定程度的了解，那么Packet类在这里起什么作用呢？先来看看图2-9，每一段二进制数据都是由多个Packet数据组成的。

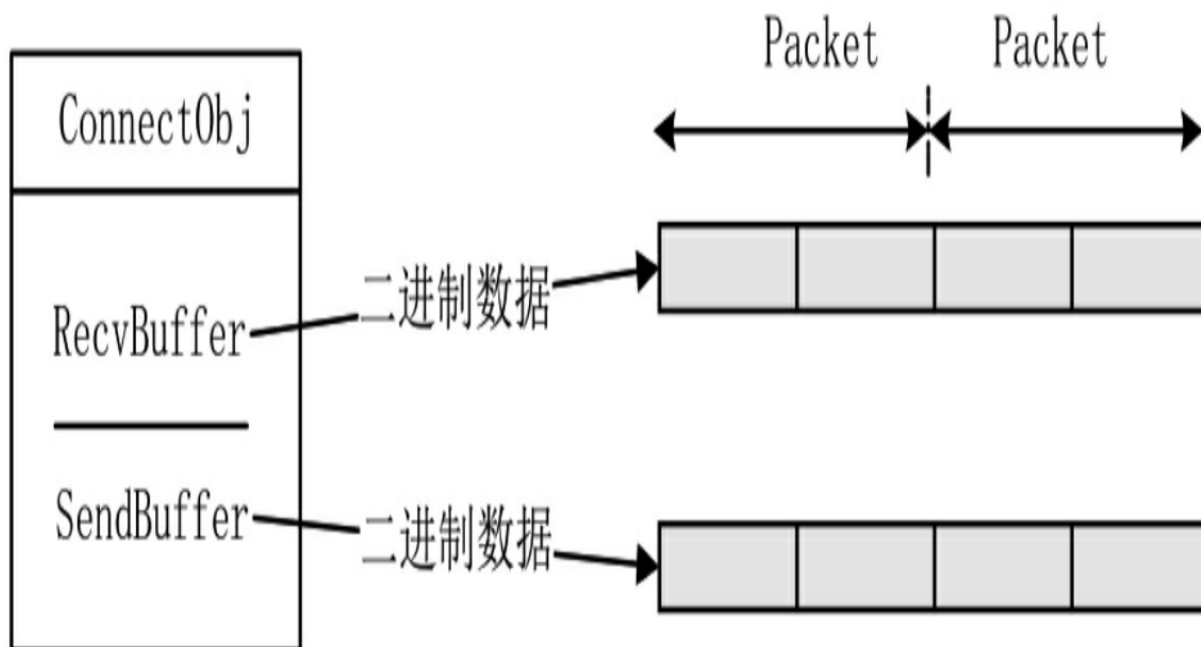


图2-9 缓存与Packet

数据从系统缓存到应用进程的缓存时都是以二进制数据存在的。在框架中，Network类向上层传递数据的时候，如果依然以二进制数据传递会非常不方便，所以Packet类由此而产生。Packet是一个内存对象，简单来说Packet让缓存中的二进制数据有了具体的意义。

在本例中，将协议数据类定义在packet.h文件中。其定义如下：

```
class Packet : public Buffer {
public:
    Packet();
    Packet(const int msgId);
    void AddBuffer(const char* pBuffer, const unsigned int size);
    ...
}
```



```
private:
    int _msgId;
};
```

在这个类中，有两个地方值得我们注意：一个是Buffer基类，它可以支持数据的输入/输出操作，这里用它来暂存协议体；另一个是协议号，创建Packet时要求输入一个MsgId来标记当前协议号。而Packet类提供的AddBuffer函数则是将一个需要发送的字符串放到Buffer中，以便在底层发送。

在前面的章节中，我们总是提到发送数据和接收数据。前面的例子中收发数据仅仅是字符串，但实际情况下，收发数据可能非常复杂，既有字符串，又有整数或浮点数。现在，我们理解了底层的收发流程，可以将目光放到应用层数据上来。在应用层，这些从网络层发来的数据统称为协议。

所谓协议，在游戏中即为两方约定的数据。既然是约定的数据，必定有一个格式。

常用的协议有文本协议和二进制协议。所谓文本协议，简单来说就是一个字符串。我们常用的HTTP就是文本协议。而二进制协议则是由数据结构组成的协议，通常由两部分组成：协议头和协议体。

关键点1：协议头是什么

在network工程的packet.h文件中就会看到自定义的协议头。

```
#pragma pack(push)
#pragma pack(4)
struct PacketHead {
    unsigned short MsgId;
};
#pragma pack(pop)
```

这就是常说的协议头，服务端和客户端都遵从这个协议头的结构。

在网络传输过程中，连续发送5条10KB的消息，在逻辑上认为它是

一条一条发送的，但在真实的网络传输过程中却不是严格按照一条一条数据到达接收端的，可能一次收到5KB，也可能一次就收到15KB，这就是网络编程中常说的粘包问题。那么我们如何判断收到了一个完整协议呢？为了解决这个问题，需要在逻辑层手动为它加上一个协议头，这个协议头的定义是自由的，但最重要的一个数据是size，表示本协议的大小。

本节例子中的协议遵循一个格式，即sizeof (unsigned short)+sizeof (PacketHead)+协议体数据。第一个unsigned short表示整个协议的长度。

先来看一下PacketHead这一结构。当收到一个完整的协议之后，读出协议头中的协议号，这个协议号明确了协议体中的数据结构。定义一个复杂一点的协议头，代码如下：

```
#pragma pack(push)
#pragma pack(4)
struct PacketHead {
    unsigned short MsgId;
    unsigned short SubId;
    unsigned short Size;
};
#pragma pack(pop)
```

给这个协议头设计了主协议号和子协议号，还包括后面协议体的长度。假如验证账号的协议MsgId为1，SubId也为1，它在内存中的数据如图2-10所示。

byte	byte	byte	byte	byte	byte
0x01	0x00	0x01	00	0x1010	0x00
MsgId		SubId		Size	

图2-10 协议头格式

协议头为固定的格式，上例中Size的大小为10字节。Size的含义可以分为两种：一种是Size大小本身（包括协议头的大小）；另一种Size的大小是从Size这个数据的下一个字节算起的。对于这两者，编程者习惯用哪种就用哪种。

收到网络数据时，首先将缓冲区中已收到的数据大小与一个协议头的大小相比较，如果小于一个协议头就不处理，等到从网络读取的数据大于等于一个协议头时，把协议头读出来，就可以取到协议体的Size，如果缓冲区中的数据小于协议体的长度，就说明数据还没有完全收到，继续等待，直到缓冲区中的数据大小足够后取出，这就是一个完整的协议。如果取完一个完整协议之后缓冲区中还有数据，那么它必定是下一个协议头。整个二进制数据的结构如图2-11所示，由一个又一个协议头与协议组成。

Head	Body	Head	Body	Body	
固定长度	可变长	固定长度	可变长		

图2-11 协议头可变长格式

按照这个思路，读取一个协议时要做两次长度判断，第一次查看是否可以取出一个头部数据，取得头部数据之后，再根据头部中的Size查看缓冲区中的数据是否足够。在这个过程中，还需要有一个PacketHead实例来辅助判断。为了提高效率，变通一下，本例中将整体数据长度放在开始处，这样就不需要创建一个PacketHead实例，只需要从缓冲区中取一个unsigned short数据，它就是一个完整包的长度，如果缓冲区中的数据小于它，就不处理，等到缓冲区中的数据足够后才创建一个PacketHead实例，将数据完整读出来。不要小看这个优化，如果服务端每秒运行100帧，那么每一帧都会查看读取缓冲区中是否有数据，每一帧为了读取长度都需要生成一个PacketHead实例，因此需要反复生成100个PacketHead实例。

关键点2：字节对齐

用pragma pack命令实现内存数据对齐。pack（push）和

pack（pop）成对出现，#pragma pack（4）指定了对齐的字节数，该命令为4字节，表示我们定义的变量都按4字节读取。我们指定的字节对齐数与定义的结构息息相关。下面用例子来说明。

```
#pragma pack(4)
struct PacketHead {
    unsigned short MsgId;
    unsigned int SubId;
    unsigned short Size;
};
```

unsigned short和unsigned int数据类型的长度分别是2和4，现在以4字节对齐，那么对于short数据类型，在内存中需要占用4位，不足4位以空数据占位，如图2-12所示。

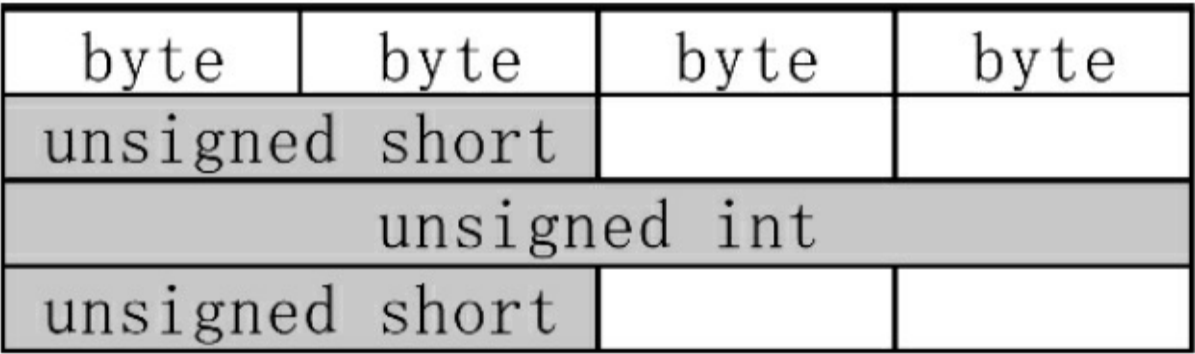


图2-12 协议头内存结构1

修改一下数据，将SubId改为unsigned short类型再来看看。如果按之前讲的规则，按4字节读取，那么定义的数据应该如图2-13左侧所示。但实际上，真正的内存分布如图2-13右侧所示。Pack有一个规则，如果指定数值小于结构中的最大值，就按结构中的最大值作为对齐值。3个unsigned short只有2字节，所以是按2字节来对齐的，而第一个例子中有一个int是4字节，所以是按4字节来对齐的。

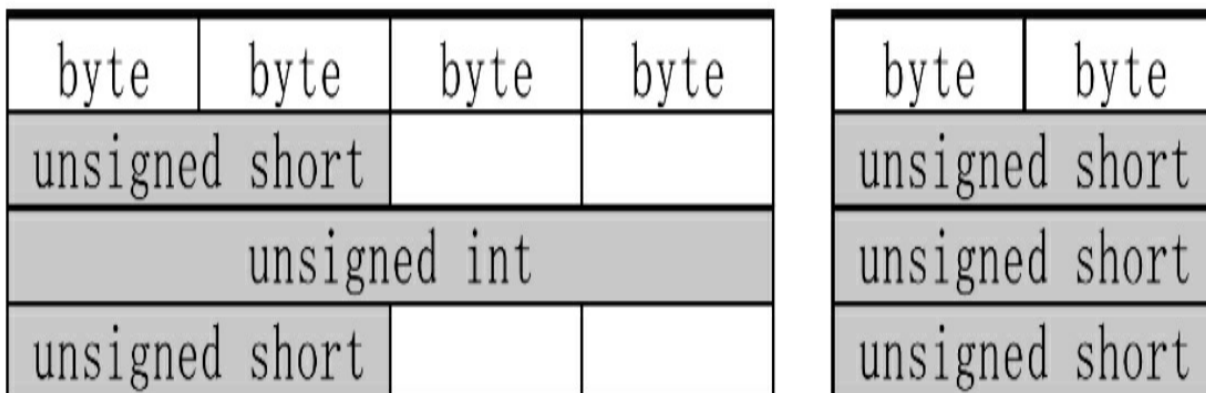


图2-13 协议头内存结构2

关键点3: 为什么需要字节对齐

字节对齐与CPU的读取方式有关，用字节对齐的方式读取速度更快。CPU读取数据是从偶数地址开始的，例如一个short类型的两个字节正好位于0x07、0x08这两个地址上，那么在读取数据时，CPU需要读取两次，分别是0x06和0x08。而字节对齐很好地解决了这个问题。

举例说明，如果一个协议头涉及char和unsigned short两种数据类型，在图2-14的上半部分数据中，第二个unsigned short就需要CPU读取两次，第一次从char的位置开始读取short的高位，因为short的高位不在偶数地址上，所以必须从前一个偶数地址开始第二次读取short的低位。而字节对齐之后，如图2-14的下半部分所示，在char类型之后，空了一个字节，使得unsigned short类型的值落在了偶数地址上，CPU只需要读取一次就可以读出unsigned short的值。

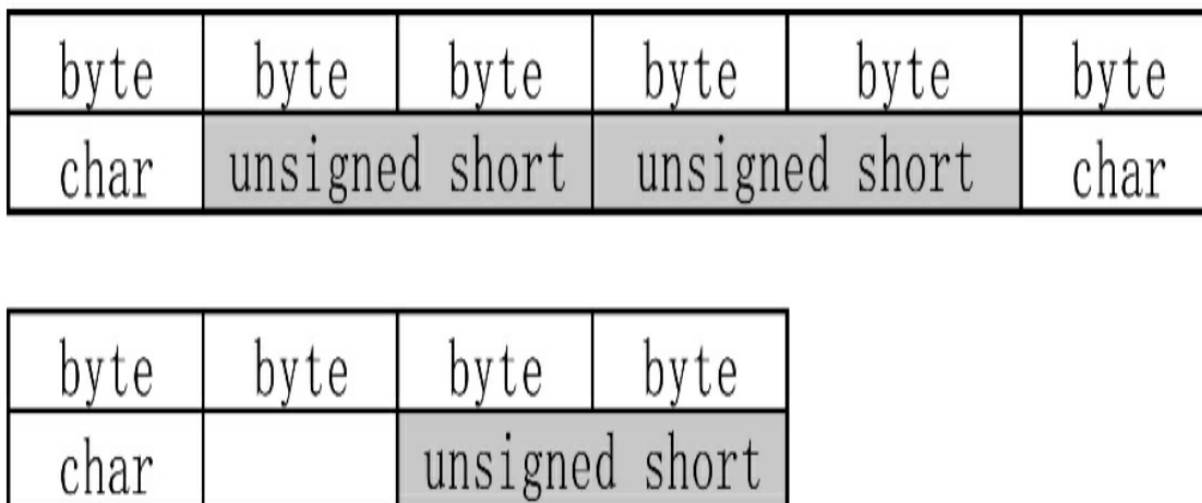


图2-14 字节对齐内存结构

鉴于这种特性，在对协议头进行字节对齐时，可以有意识地把占用内存大的数据类型的数据放在前面，而把多个占用内存小的数据类型的数据组合成字节对齐的单位，这样既节省了空间，又节省了时间。

关键点4：关于协议的特别说明

（1）在游戏协议中最好不要出现字符串，传递时间类型时采用long类型而不是string类型。字符串"123"和数字123在传输中的占位大小是不相同的，区别非常大。

那么，对于需要显示字符串的地方应该如何处理呢？当然是定义字符串码，一般用unsigned short类型来定义字符串码。例如，登录错误码定义为01，取到01错误码之后，再由客户端显示出中文意思。

（2）在游戏功能越来越丰富、玩家游戏时长越来越长的情况下，如果在编码的时候不注意，就会让存储的数据越来越庞大。特别是一些游戏运营活动，后期维护不当，有可能在第一次同步玩家数据的时候，让玩家数据过大，超过缓存预期的长度，这时必须拆包。

例如，玩家的数据已经达到了128KB，但一次发送协议的最大长度为64KB，那么必须分成两次发送。

可以从逻辑上规避这种情况，例如将帮派数据、基本数据、道具数据分开发送。但是作为网络底层，我们永远不知道上层逻辑调用时会出现什么情况。所以，在框架前期需要考虑到这些特殊的情况，并考虑好兼容的问题。

其中一个解决方案就是，当包的长度超过我们预期的最大值时，自动将它拆分成两个包，并在客户端进行拼接。要完成这个功能，需要在PacketHead中添加两个字节大小的数据：总包序号和当前包序号。默认情况下，两个数据为0，即只有一个包。当出现2和1时，表示后面还跟有一个相关的包。客户端只有将这两个包都取到之后，才能拼成一个完整的协议数据。

在本书的例子中，SendNetworkBuffer和RecvNetworkBuffer都采用了自增长缓存的方法，这是另一个解决方案，也可以有效地避免这个问题。

关键点5：协议体

在早期的游戏编程中，协议内容一般是由程序员自定义一个结构类型。以登录为例，它的结构类型的定义可能如下：

```
struct AccountCheck{
    unsigned short Version;
    char Account[128];
    char Password[128];
};
```

结构类型定义完成之后，在代码中实现序列化，并转化为二进制串。这个结构类型一旦定义，客户端和服务端就必须使用相同的格式。当数据从客户端到达服务端时，以同样的规则反序列化，生成一个结构体（Struct）。

自定义结构类型有一定的优势，执行效率相对来说比较高，序列化与反序列化都是清晰可见的。但自定义结构类型有一个致命的缺点，当客户端和服务端协议结构不一致时，容易引起异常或者宕机，必须解决这类兼容问题。特别是对于在线游戏，有人对协议进行分析试探的时候，传来的协议可能是错误的。我们必须有一个根本的认识，从网络传

来的协议任何时候都是不可靠的，它有可能是一个伪客户端。

另一方面，在上面自定义的结构类型的结构体中加了一个Version字段，随着游戏上线的时间增长，我们要修改原来的协议变得十分烦琐。因为既要考虑到旧的结构体，又要处理新的结构体。常用的办法就是增加Version字段，同一个协议的每一个不同的版本都需要处理。

现在不需要这么复杂的步骤了，有了一个可替代方案，就是Google提供的Protocol Buffer开源项目，简称Protobuf。Protobuf是跨平台的，并提供多种语言版本，也就是说，服务端和客户端的编程语言可以不一致，数据却可以通用。序列化和反序列化功能Protobuf都已经完成了，不需要我们过多关心，这样可以把编码的重心放在游戏逻辑上。本书中，协议体部分都采用Protobuf定义的结构，在后面的章节中，有专门的例子来介绍Protobuf，这里不再赘述。

回过头去看服务端和客户端对于Packet的使用。从ConnectObj取出数据时，取出来的是一个Packet实例，既有协议头，又有协议体。通过Packet可以读出这个协议的完整内容。同时，发送数据时，只需要向ConnectObj传入一个Packet对象，ConnectObj类中的函数将Packet传到Buffer中去，将它转为二进制数据并发送给目标。Packet类简化了对于二进制数据的操作。

2.1.13 小结

本节非常详细地介绍了：`select`函数，并为它创建了两个不同功能的类：一个用于监听；另一个用于连接。经过梳理，会发现其实Select模式并不复杂，其原理就是将关心的Socket放到读、写或异常集合中，再传递给：`select`函数，它会帮助我们分析、筛选并返回有事件的Socket集合。一旦有了返回值，再判断每个Socket处于哪一个集合中，针对不同的集合有不同的操作。如果Socket处于读集合，那么进行读操作；如果Socket处于写集合，那么进行写操作；如果发生异常，就可以断开网络。

值得注意的是，：`select`函数的连接数量是有限制的，在Linux系统中，将robots测试工程中的线程调为1100就会发现产生错误提示，这是因为Linux默认支持集合的最大值为1024，当然可以通过修改将其扩大。`_FD_SETSIZE`定义在`/usr/include/Linux`目录下的`posix_types.h`文件中：

```
#define __FD_SETSIZE 1024
```

在Windows下，我们从WinSock2.h中可以看到：

```
#ifndef FD_SETSIZE
#define FD_SETSIZE      64
#endif /* FD_SETSIZE */
```

因此，我们可以在工程的`network.h`文件中定义一个`FD_SETSIZE`值。

```
#define FD_SETSIZE      1024
```

Select模型是一种主动的查询模式，这种模式有上限的局限性，而且每一帧都需要把所有关心的Socket压入集合中，这也带来了效率问题。

下面我们介绍一种Linux特有的非阻塞模型——Epoll。

Epoll网络模型

Select模型兼容两种系统，但Epoll只能在Linux系统下运行。相对于Select，Epoll最大的好处在于它不会因为连接数量过多而降低效率。在前面的示例中，我们可以发现Select采用了轮询的方式，假如有3000个连接，在每一帧都要将3000个Socket放到集合中进行轮询。而Epoll正好相反，Epoll对需要监听的事件进行注册，对于读事件，只需要注册一次，而不是每一帧盲目地轮询。

Epoll和Select需要解决的问题是一样的：网络连接何时可以读写，何时出错。牢记这个关键就会发现Epoll其实并不复杂。

2.2.1 函数说明

3个关键函数说明如下：

(1) int epoll_create (int size)

要使用Epoll，就需要创建描述符。创建函数返回一个文件描述符。当我们不再需要使用时，需要调用：：close来关闭描述符。参数size表示初始数组的长度。

(2) int epoll_ctl (int epfd, int op, int fd, struct epoll_event*event)

事件注册函数，对Epoll监听的列表进行事件注册或者删除操作。该函数的参数说明见表2-6。

表2-6 epoll_ctl函数的参数

参 数 名	说 明
epfd	创建函数生成的 Epoll 文件描述符
op	是对 Epoll 的监听列表进行操作的命令。op 的常用值有 3 个： ● EPOLL_CTL_ADD: 注册新的 fd 到 epfd 中。 ● EPOLL_CTL_MOD: 修改已经注册的 fd 的监听事件。 ● EPOLL_CTL_DEL: 从 epfd 中删除一个 fd
fd	监听的文件符，这里也就是 Socket 值
event	epoll_event 结构

例如，要对一个Socket进行读监听，需要先创建一个`epoll_event`结构，再调用函数：`epoll_ctl`将其注册到由：`epoll_create`生成的Epoll描述符上去。其代码如下：

```
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLRDHUP;
ev.data.fd = fd;
epoll_ctl( epollfd, EPOLL_CTL_MOD, fd, &ev );
```

以上代码调用：`epoll_ctl`函数修改文件符的监听事件为`EPOLLIN`和`EPOLLRDHUP`。在Linux下一切皆为文件描述符，包括Socket，这里`fd`也就是Socket值。`event`监听事件的枚举常用值说明如下：

- `EPOLLIN`：文件描述符可读。
- `EPOLLOUT`：文件描述符可写。
- `EPOLLPRI`：文件描述符有紧急的数据可读。
- `EPOLLERR`：文件描述符发生错误。
- `EPOLLHUP`：文件描述符被挂断。
- `EPOLLRDHUP`：文件描述符在对端被断开。

(3) `int epoll_wait (int epfd, struct epoll_event*events, int maxevents, int timeout)`

在给定的时间内，对已注册的文件符监听的事件进行收集。该函数类似于：`select`函数，其参数说明见表2-7。

表2-7 `epoll_wait`函数的参数

参 数 名	说 明
epfd	生成的 Epoll 文件描述符
events	收集到的事件数组
maxevents	最大事件数
timeout	timeout 的单位是毫秒，0 值意味着立即返回

函数：`epoll_wait`的返回值是收集到的监听事件数量。参数 `events` 是一个输出参数，当监听事件有返回时，`events` 就是有事件发生的文件描述符集合。

2.2.2 源代码分析

下面以实际代码来说明Epoll的工作方式。本节例子的源代码在02_02_network_epoll目录下，框架结构与2.1节的工程完全一致，仅Select代码的部分被Epoll替代。为了区分代码使用了Epoll宏。在Network类中增加相关函数，代码如下：

```
class Network : public IDisposable {
...
    void InitEpoll();
    void Epoll();
    void AddEvent(int epollfd, int fd, int flag);
    void ModifyEvent(int epollfd, int fd, int flag);
    void DeleteEvent(int epollfd, int fd);
protected:
#define MAX_CLIENT 5120
#define MAX_EVENT 5120
    struct epoll_event _events[MAX_EVENT];
    int _epfd;
    int _mainSocketEventIndex{ -1 };
    ...
};
```

函数AddEvent、ModifyEvent和DeleteEvent的功能是对Epoll文件符集合进行增加、修改和删除，3个函数都使用到了底层的：`epoll_ctl`函数，传入的参数各有不同，具体如下：

```
void Network::AddEvent(int epollfd, int fd, int flag) {
    struct epoll_event ev;
    ev.events = flag;
    ev.data.ptr = nullptr;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &ev);
}
void Network::ModifyEvent(int epollfd, int fd, int flag) {
    struct epoll_event ev;
    ev.events = flag;
    ev.data.ptr = nullptr;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_MOD, fd, &ev);
}
```

```

}
void Network::DeleteEvent(int epollfd, int fd) {
    epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, nullptr);
}

```

在详细分析代码之前，先来了解一下整个流程。如图2-15所示的流程图说明了Epoll的工作原理。

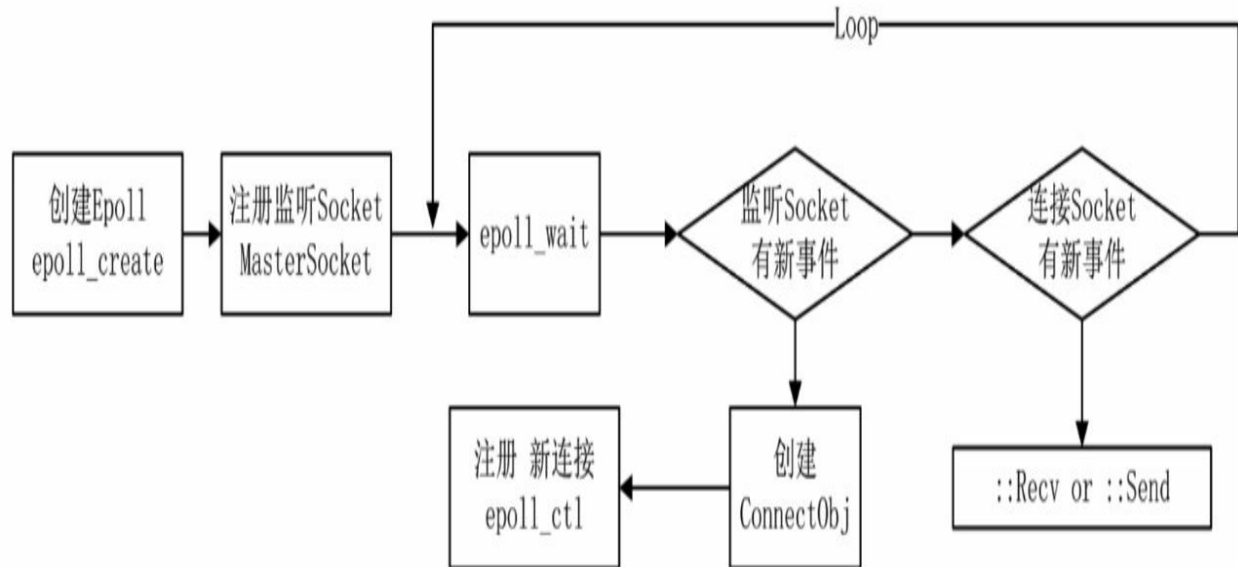


图2-15 Epoll流程

关键点1：初始化

初始化函数InitEpool，实现了Epoll的创建。

```

void Network::InitEpool() {
    _epfd = epoll_create(MAX_CLIENT);
    AddEvent(_epfd, _masterSocket, EPOLLIN | EPOLLOUT | EPOLLRDHUP
}

```

对于监听类，监听端的MasterSocket有数据会触发EPOLLIN读事件。对于连接类，连接成功会触发EPOLLOUT写事件，如果是直接成功的，就会触发EPOLLIN读事件。所以对于MasterSocket，在增加监听事

件时，监听了读、写和异常3个事件。

关键点2: Epoll函数

准备好之后，就可以调用：`epoll_wait`函数进行监听。每一帧判断一下MasterSocket上是否有事件。如果有新的连接，就生成ConnectObj并执行：`epoll_ctl`函数，使用EPOLL_CTL_ADD命令把新的Socket加入需要监听的集合中，所监听的事件为EPOLLIN和EPOLLRDHUP，即需要监听网络的读操作与异常操作。源代码实现如下：

```
void Network::Epoll() {
    _mainSocketEventIndex = -1;
    // 遍历所有连接是否有待发送的数据，如果有，就增加EPOLLOUT写事件
    for (auto iter = _connects.begin(); iter != _connects.end(); ++iter) {
        ConnectObj* pObj = iter->second;
        if (pObj->HasSendData()) {
            ModifyEvent(_epfd, iter->first, EPOLLIN | EPOLLOUT | E
        }
    }
    // 对所有文件符的监听事件进行收集，返回有效事件个数
    int nfds = epoll_wait(_epfd, _events, MAX_EVENT, 50);
    for (int index = 0; index < nfds; index++) {
        int fd = _events[index].data.fd;
        if (fd == _masterSocket) {
            _mainSocketEventIndex = index;
        }
        auto iter = _connects.find(fd);
        if (iter == _connects.end()) {
            continue;
        }
        // 如果有错，就关闭网络
        if (_events[index].events & EPOLLRDHUP || _events[index].e
            iter->second->Dispose();
            delete iter->second;
            iter = _connects.erase(iter);
            DeleteEvent(_epfd, fd);
            continue;
        }
        // 有读操作
        if (_events[index].events & EPOLLIN) {
            if (!iter->second->Recv()) {
                iter->second->Dispose(); // 有异常，断开网络
                delete iter->second;
            }
        }
    }
}
```

```

        iter = _connects.erase(iter);
        DeleteEvent(_epfd, fd);
        continue;
    }
}
if (_events[index].events & EPOLLOUT) {
    iter->second->Send();
    // 有写操作，写操作完成之后修改监听事件，去掉EPOLLOUT
    ModifyEvent(_epfd, iter->first, EPOLLIN | EPOLLRDHUP);
}
}
}

```

在这个函数中，可以发现与：`select`函数不同的一点，：`select`函数每一帧中都对所有Socket进行注册，但是Epoll不需要那么频繁操作，只需要修改一次，底层就会一直保持其监听事件，直到上层逻辑对其监听的事件做出修改。在上面的代码中，发送数据完成之后，把监听文件符的事件重新修改为“EPOLLIN|EPOLLRDHUP”，即去掉EPOLLOUT写事件。底层函数不再关心写事件，避免每次都去检查可写，等到有数据需要发送时，修改文件符的事件再加入EPOLLOUT事件，实现按需调用。

关键点3：水平触发与边缘触发

在本例中，发送数据完成之后，将EPOLLOUT写事件移除了。假如不移除，会有什么事件发生呢？可以来测试一下。现在修改一下代码，在network.cpp的第179行将“ModifyEvent(_epfd, iter->first, EPOLLIN|EPOLLRDHUP)”这句去掉，同时加上一句“std::cout<<"trigger EPOLLOUT.socket: "<<fd<<std::endl;”，重新编译，执行“./robots 1 1”启动一个线程，发送一条数据。此时，可以看到在服务端会有无数个trigger EPOLLOUT的打印。这说明，如果我们不去掉对于EPOLLOUT事件的监听，那么只要写缓冲为空，就会触发这个事件。在Epoll中，这种情况为水平触发。水平触发模型下，只要读缓冲不为空，就会一直发送读事件，只要写缓冲为空，就会不断发送写事件。

但我们的程序并不是时时刻刻都在发送数据，所以Epoll提供了另一种边缘触发模式。在边缘触发模式下，只有读缓冲从没有数据变为有数据才会发送一次读事件，写缓冲从不可写变为可写时发送一次写事

件。边缘触发只有在临界值发生改变时才会触发事件，这样就大大降低了事件的触发频率。

现在修改一下代码。在函数Network: : CreateConnectObj和Network: : InitEpoll中，调用AddEvent函数时增加EPOLLET，即代码变为AddEvent (_epfd, socket, EPOLLET|EPOLLIN|EPOLLOUT|EPOLLRDHUP)。函数Network: : Epoll ()，修改ModifyEvent函数，增加EPOLLET。修改ModifyEvent (_epfd, iter->first, EPOLLET|EPOLLIN|EPOLLOUT|EPOLLRDHUP)，重新编译并执行“./robots 2 1”。此时可以观察到，写事件只到来了有限的数次，不再是一直触发。

边缘触发比起水平触发事件通知的时机减少，就这带来了另一个需要注意的事项，那就是只有这个事件发生，这个事件上的数据需要一次读完。以读为例，当有数据到来时会触发一次读事件。读的时候需要一次性把所有数据从缓冲中读完。如果不读完，就会产生遗留数据，在下一帧，读事件是不会再触发的。

2.2.3 小结

在测试这个例子时，可以在robots进程上运行1000个线程，每个线程要求完成两个数据的发送，两个来回即4条数据，也就是说一共发起了1000个连接，发送了 $1000 \times 2 \times 2$ 条数据。

同时，可以对两种模式的代码效率进行对比，对server和robots工程下的CMakeLists.txt文件进行修改。找到这一行“set (CMAKE_CXX_FLAGS "-Wall -std=C++14 -pthread -DEPOLL")”，去掉-DEPOLL，然后重新执行make-all脚本，编译出来的版本为Select版本。

在测试效率时需要将所有常规打印信息注释掉。打印日志的消耗在这种毫秒级别的测试中是非常大的。先启动服务器，再启动robots进程。表2-8详细展示了两种模式下的对比结果。每一种模式都执行了5次。

整体来说，Epoll的性能高于Select，测试结果每个人跑出来的数据可能都不一样，因硬件环境不同而不同，如果机器配置较低，也可以只跑500个数据。到目前为止，我们介绍了网络底层函数，还介绍了处理网络协议的Select模型和Linux下特有的Epoll模型。

表2-8 Select与Epoll效率对比

Epoll	Select
./robotsd 1000 2 time:1.53097s	./robotsd 1000 2 time:2.65565s
./robotsd 1000 2 time:1.37288s	./robotsd 1000 2 time:3.34457s
./robotsd 1000 2 time:2.37195s	./robotsd 1000 2 time:3.35389s
./robotsd 1000 2 time:1.78472s	./robotsd 1000 2 time:2.62674s

为了更好地融合进本书的框架，我们写了一套网络数据处理方式，使用的底层函数和模式都是网络编程中基本的知识点。一些现成的库（例如libevent、boost）中也有相关的Socket操作类，在编写网络功能的程序代码时可以借鉴和参考。libevent是一个用C语言编写的开源事件通知库，它的性能优良，可以编译为静态库进行使用。本节中，Epoll模型的例子中使用了一个固定的events数组，在libevent库中对其实现了自增长，其代码有一定的参考价值。由于篇幅所限，这里不再对其进行示例演示，如果读者感兴趣，相关的延伸阅读可以在网络上搜索一下。

网络协议：protobuf

前面我们提到过protobuf，本节对其进行简要说明。首先，我们要弄清楚protobuf究竟是什么。

在网上搜索protobuf，有很多标准化的回答，例如“混合语言数据标准”“序列化工具”等。但它到底是什么呢？用一种更贴切编程的思路来回答，首先，它是一个跨平台的代码生成器；其次，它提供了一系列的序列化方法。

为什么是代码生成器？来看一下它的定义就明白了。使用protobuf首先需要定义一个.proto文件。我们在里面编写一个结构：

```
message TestMsg {  
    string msg = 1;  
}
```

这是一个文本定义，protobuf工具可以将这个结构转换成各种语言，如果需要C++文件，就转换为.cpp文件；如果需要C#文件，就转换为.cs文件。这种跨平台的操作极大地方便了编码，因为服务端语言和客户端语言极有可能不一致。以C++为例，protobuf会将我们定义的TestMsg转为一个类，msg作为它的一个变量，同时提供修改和取得变量的函数。

其次，什么是序列化。在前面的章节讲到网络传输时使用的都是二进制，protobuf的序列化功能将一个结构实例序列化成二进制数据，在目标端可以反序列化还原这个数据。在序列化和反序列化的过程中，protobuf做到了极好的兼容，它使用定义的数字来标识一个字段，而不

是字段名。

了解它的意义之后，在我们的框架中会将这部分功能放在哪里呢？答案就是Packet类的缓存中。要更好地了解使其用方式，还是从例子入手。我们先来看看如何编译protobuf。

2.3.1 在Windows下编译使用protobuf

在GitHub上可以下载protobuf的源代码：<https://github.com/google/protobuf/releases>，从中下载一个较新的版本。因为我们将要用到C++版本，所以下载protobuf-cpp的压缩文件。在Windows下编译protobuf需要用到CMake的Windows版本，所以另一个需要安装的工具是Windows版的CMake。

为了方便使用，在本书源代码库中为读者准备了一份下载好的protobuf和CMake压缩文件。同时，在源代码根目录的lib和include目录下准备了Visual Studio 2019编译的protobuf库文件和头文件，如果当前版本不是Visual Studio 2019，那么可以按下面的步骤进行编译。

打开cmake-gui.exe，出现可视化的配置界面，如图2-16所示。

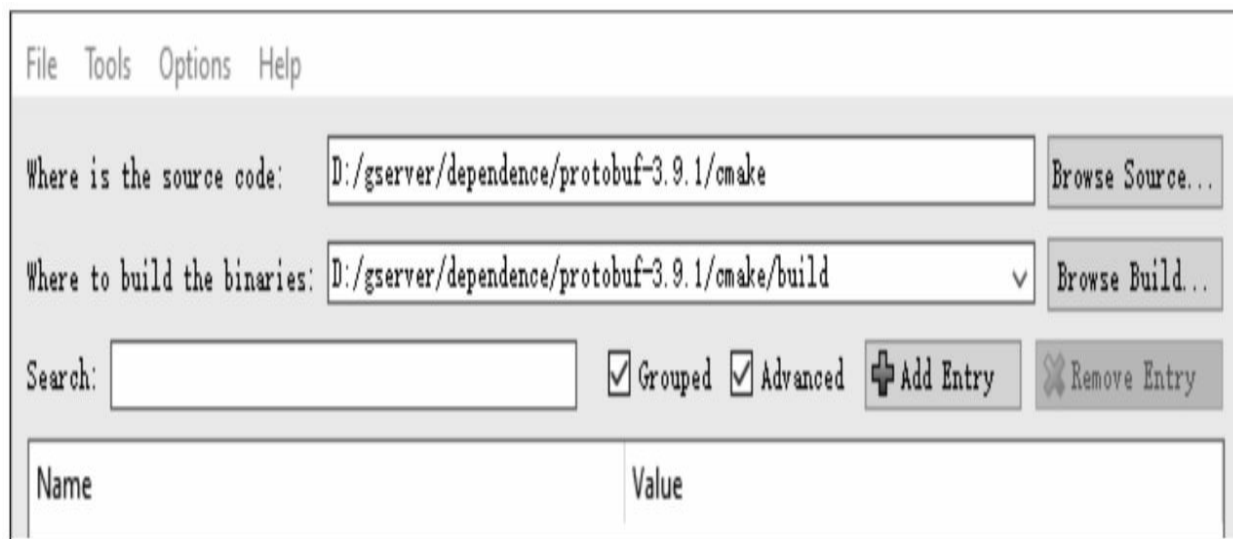


图2-16 CMake protobuf

解压protobuf-cpp的压缩文件，在CMake工具界面的source目录中选中protobuf-3.9.1\cmake目录，创建protobuf-3.9.1\cmake\build目录作为生成目录。默认情况下，生成静态运行库，为了配合我们的工程需要进行修改，在CMake界面中，勾选Grouped和Advanced复选框，让CMake的所有属性显示出来，取消勾选protovuf_msvc_static_runtime复选框，然后单击界面右下方的Configure按钮。

配置完成，单击Generate按钮生成工程。生成成功之后，这时在cmake\build目录中就有Visual Studio的工程文件Protobuf.sln。在Windows下编译时，我们需要检查工程文件在Visual Studio中的默认运行库设置。默认运行库的值一般有如下几个：

- /MD表示运行时库不集成，生成的文件较小。
- /MT表示运行时库集成，生成的文件较大，但MT移植性好，没有过多的DLL依赖。
- /MDD、/MTD表示两个版本的调试版本。

使用MD编译，运行时可能存在本地不存在运行库的情况。但生成的文件很小，考虑到都是在开发机上运行的，为了方便，所有第三方库都选用MDD，配置如图2-17所示。

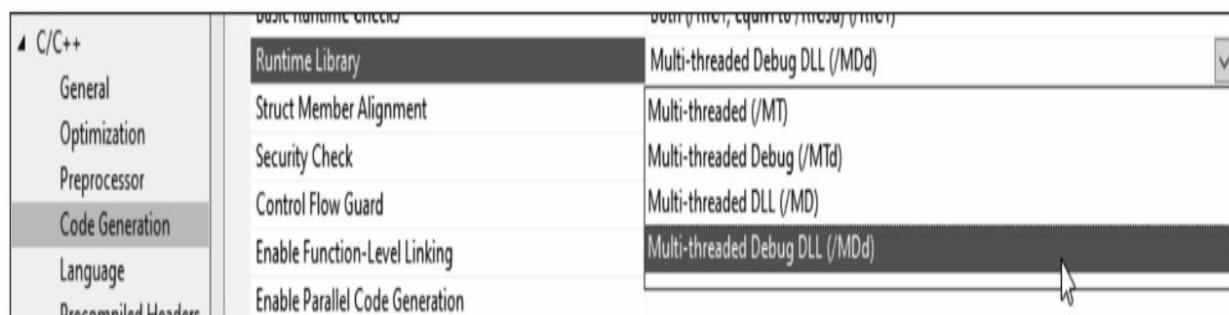


图2-17 C++运行库

打开protobuf.sln，完全编译工程。我们最终需要的是protoc.exe和libprotobufd.lib这两个文件，把它们存放在build\debug目录下。从libprotobufd.lib这个名字中最后的d可以看出，生成的是一个Debug版本。生成的protoc.exe工具不仅仅是为本例服务，以后所有的工程都会用到它，为了以后方便共用文件，我们在源代码库的根目录增加了3个目录：

(1) tools目录。将protoc.exe放进去。

(2) libs目录。该目录放置所有工程中需要的第三方库文件，其下有Linux和Windows两个子目录，分别对应Linux和Windows需要的库。将libprotobufd.lib文件放到libs\windows目录下，以后所有的工程

Windows版都可以直接使用该库文件，不用再重新编译。

(3) include目录。该目录放置所有工程中需要的第三方include目录，将protobuf-3.9.1\src\google整个目录复制到include\windows目录下。

编译完成之后，我们需要把protobuf加入工程中，参见源代码库下的02_03_protobuf工程，我们需要对Windows版的network工程做出一系列的调整。

关键点1：修改Visual Studio工程的include目录

为了便于配置，我们已将Windows所有用到的头文件都放在include\windows目录下，同时还有一个common目录存放Linux和Windows都需要的文件。修改network工程属性“VC++目录”→“包含目录”，加入：

```
$(SolutionDir)..\include\common  
$(SolutionDir)..\include\windows
```

其中，变量\$(SolutionDir)是Protobuf.sln文件所在的目录。同理，修改server工程和robots工程，增加include目录。

关键点2：修改Visual Studio工程引用库

在server和robots工程属性“VC++目录”→“库目录”增加“\$(SolutionDir)..\libs\windows”目录。现在我们的库目录有两个目录，虽然看上去路径描述不太一样，但是这两个目录都指向库目录。

(1) “\$(SolutionDir)..\libs\windows”指向了第三方库的目录下，也就是源代码的根目录libs目录。

(2) “../../../libs/\$(Configuration)”指向了本工程中的libs。

经过图2-18的对比，可以看到有两个libs目录，看上去有点绕。再

次强调，根目录下的include和libs是所有工程共用的，例如protobuf库，而工程目录下的libs是本工程专用的。使用以上目录的原因在于，大量的示例工程基于相同的第三方库，所以提出一个共用库目录，而每个工程有自己的库工程，例如本小节与2.2节的network库代码就不一样，生成的库文件也是不同的。这时，这些工程特有的库文件会生成在工程的libs子目录下。

```

└── 02_03_protobuf
    ├── bin                // 执行文件生成目录
    ├── libs              // 库生成位置
    └── src                // 工程源代码目录
        ├── apps
        │   ├── server
        │   ├── libs
        │   │   ├── network // network库
        │   └── tools
        └── robots
└── include                // 全局include
    └── windows
└── libs                  // 全局libs, 存放的第三方库
    └── windows
└── tools
    └── protoc.exe

```

图2-18 根目录与工程目录对比

关键点3: 修改Visual Studio引用库

增加库目录之后, 工程server和robots还需要指定库文件。根目录libs\windows下已有需要的libprotobufd.lib文件了。打开robots的属性页面, 在连接器中加入依赖项。server工程同理。

完成了以上几点, protobuf库已经被加入工程中, 可以在Windows下使用了。

2.3.2 在Linux下编译使用protobuf

比起Windows，在Linux下使用protobuf相对简单一些。在Linux默认的仓库中有一个protobuf 2的版本，因为我们使用的是protobuf 3，所以需要在Linux下编译。下面是它的编译步骤。

首先，在终端对protobuf-cpp-3.9.1.tar.gz进行解压，这个文件已经在源代码根目录的dependence目录下了。

```
[root@localhost dependence]# tar xf protobuf-cpp-3.9.1.tar.gz
```

解压完成之后，dependence目录下多了一个protobuf-3.9.1目录，进入其目录，依次执行以下3个命令，对protobuf进行编译与安装。

(1) [root@localhost protobuf-3.9.1]# ./configure

(2) [root@localhost protobuf-3.9.1]# make

(3) [root@localhost protobuf-3.9.1]# make install

命令make的执行需要花费一定的时间，最后make install将protobuf安装到/usr目录下。安装完成之后，可以发现目录/usr/local/include下有protobuf相关的头文件了，库文件在/usr/local/lib目录下。安装完成之后可以检查一下版本号。

```
[root@localhost protobuf-3.9.1]# protoc --version  
libprotoc 3.9.1
```

安装完成之后，要让工程02_03_protobuf在Linux下成功运行，我们需要修改一些参数。

关键点1：环境变量

在make install之后，可以看到安装提示，我们的库被安装

到/usr/local/lib目录之下，include目录位于/usr/local/include中。我们要在CMake中使用protobuf，就需要先配置两个环境变量。

```
[root@localhost ~]# vim ~/.bash_profile
```

修改如下：

```
LD_LIBRARY_PATH=/usr/local/gcc-9.1.0/lib64/:/usr/local/lib:$LD_LIB
export LD_LIBRARY_PATH
LIBRARY_PATH=/usr/local/lib:$LIBRARY_PATH
export LIBRARY_PATH
```

在附录中安装Gcc时，也修改过.bash_profile文件，它可以配置当前账户的个人环境变量。LIBRARY_PATH是静态库的环境变量，编译时需要使用静态库时，就会从LIBRARY_PATH这个变量指定的目录中查找。除了静态库外，还修改了LD_LIBRARY_PATH环境变量，这是动态库的环境变量。修改了环境变量之后，需要关闭终端并重新进入。

关键点2: CMakeLists文件配置

在server工程中加入protobuf，还需要修改CMakeLists.txt文件。

(1) 增加include目录

```
include_directories(/usr/local/include)
```

(2) 连接protobuf库

```
set(CMAKE_CXX_FLAGS "-Wall -std=C++14 -pthread -lprotobuf -DEPOLL"
```

在CMAKE_CXX_FLAGS变量中加入了“-lprotobuf”，在编译时会从环境变量的路径中找到protobuf的库文件。有关环境变量已经在上面的步骤中设置完成了。

2.3.3 使用protobuf定义协议

在源代码库下的02_03_protobuf目录准备了一个应用protobuf的示例。工程与前一个例子的流程完全一致，只是在协议体的处理上采用了protobuf定义协议，而不再是字符串。我们来看看使用protobuf协议的几个关键点。

关键点1：定义.proto文件

在使用protobuf定义协议时，需要编写一个后缀为.proto的文件，并对协议进行定义，本例中的定义文件在network工程的msg.proto文件中。protobuf的语法非常简单，我们逐一进行说明。

```
syntax = "proto3";           // 此位表示是以proto 3的语法来书写定义
package Proto;              // 给定一个命名空间，本例中为Proto
message TestMsg {           // 定义一个消息
    string msg = 1;
    int32 index = 2;
}
```

消息定义的关键字为message，为了便于序列化，每一个字段的格式为：类型名字=分配符。分配符不可以重复，但可以跳跃。一般来说分配符一旦确定，最好不要修改，实际上并不绝对。在网络通信中，分配符是可以修改的，修改的前提是服务端和客户端同时修改。因为通信的消息总是及时的，不存在存储的问题，如果把protobuf的序列化串作为DB的存储字段，那么是绝对不可以修改分配符的。

在序列化时，protobuf根本不在乎这个属性的名字是什么，也不会存储名字，它所关心的是属性的分配符，分配符和类型是一一对应的。在protobuf的语法中，message可以嵌套一个message。例如，我们可以写成如下形式：

```
message TestComplexMsg {
    bool isBool = 1;
    repeated TestMsg test_msg = 2;
}
```

在TestComplexMsg消息中包括TestMsg消息，关键字repeated表示该属性是一个TestMsg数组。

除了msg.proto文件之外，工程中还新建了一个proto_id.proto文件，用来定义协议号：

```
syntax = "proto3";
package Proto;
enum MsgId {
    None = 0; // proto3的枚举，第一个必为0
    SendData = 1;
    Over = 2;
}
```

这里定义了两个协议号，测试时发送数据使用MsgId.SendData，用到的结构类型为TestMsg。更多的protobuf语法可以参见其官方文档。

关键点2：生成.cpp、.h文件

定义好.proto文件之后，我们可以写一个批处理来生成.cpp、.h文件。批处理文件内容如下：

```
"../../../../tools/protoc.exe" --cpp_out=./ proto_id.proto msg.
```

即让protoc.exe解析proto_id.proto和msg.proto文件，参数--cpp_out指定C++的输出文件目录，“./”为当前目录。在生成文件时用到了protoc.exe，是在编译protobuf时protoc项目生成的，复制到了根目录tools目录下。

该批处理文件位于network/protobuf目录下。build.bat可以在Windows系统下执行，build.sh在Linux下执行。

执行批处理之后，会得到两组4个文件：msg.pb.h、msg.pb.cc、proto_id.pb.h和proto_id.pb.cc，加入network工程中进行编译。

关键点3: 在Packet类中使用protobuf结构

为了方便地使用protobuf, 修改一下Packet类, 提供两个模板函数:

```
class Packet : public Buffer {
public:
    ...
    template<class ProtoClass>
    ProtoClass ParseToProto() {
        ProtoClass proto;
        proto.ParsePartialFromArray(GetBuffer(), GetDataLength());
        return proto;
    }
    template<class ProtoClass>
    void SerializeToBuffer(ProtoClass& protoClase) {
        auto total = protoClase.ByteSizeLong();
        while (GetEmptySize() < total) {
            ReAllocBuffer();
        }
        protoClase.SerializePartialToArray(GetBuffer(), total);
        FillData(total);
    }
};
```

函数ParseToProto根据输入的类型将Packet缓冲中的二进制数据反序列化成一个协议结构, 即protobuf定义中生成的结构, 而函数SerializeToBuffer将一个结构序列化到Packet的缓冲中, 该函数使用了protobuf的底层函数SerializePartialToArray。打开robots工程的client.cpp文件, 可以看到发送和读取的代码:

```
void Client::DataHandler() {
    ...
    if (_lastMsg.empty()) {
        // 发送数据
        _lastMsg = GenRandom();
        Proto::TestMsg protoMsg; // 定义结构
        protoMsg.set_index(_index);
        protoMsg.set_msg(_lastMsg.c_str()); // 将随机字符串作
        Packet packet(Proto::MsgId::SendData); // packet实例, 组
        packet.SerializeToBuffer(protoMsg); // 将结构序列化
```

```

        SendPacket(&packet);
    } else {
        if (HasRecvData()) {
            Packet* pPacket = GetRecvPacket();
            if (pPacket != nullptr) {
                // 取出协议体结构
                Proto::TestMsg protoMsg = pPacket->ParseToProto<Proto:
                ...
            }
        }
        ...
    }
}

```

在2.2节的源代码中，在工程robots的client.cpp文件中，发送Packet时使用的是Packet packet(1)，传入了一个常量值，表示协议号，给了一个固定值1。现在可以用proto_id.proto中定义的枚举类型来标识它，使常量变得更有意义，也更便于阅读。Packet packet(Proto::MsgId::SendData)一看就知道是一个发送数据的协议。

在使用protobuf中的嵌套消息时，要特别注意对函数：
set_allocated_xxx和::mutable_xxx的使用：

(1) 使用::set_allocated_xxx，赋值的结构对象需要用new创建出来，不能用局部的，这里保存的是对象的指针。

(2) 使用::mutable_xxx赋值时可以使用局部变量，因为在调用时内部执行了创建操作。

关于protobuf更多的细节可以参考官方文档。除了protobuf之外，在组织协议内容上还可以使用message pack。message pack也提供了多种语言版本，其速度快于protobuf，message pack的兼容性没有protobuf那么好，我们这里所说的兼容性是指同一个协议发生变化时，其协议的兼容问题。用于网络协议的传输，message pack也是非常不错的选择，但用于DB数据存储时，message pack显然是不合适的，protobuf在数据存储上优势非常明显。在后面讲解数据库时，我们会选择存储二进制数据，并以protobuf的格式保存。

2.4

总结

网络编程部分就介绍到这里了。回顾一下，这两章从基本的网络函数开始，我们了解了网络数据的阻塞与非阻塞、收发流程以及两个网络模型Select和Epoll，阐述了协议头与协议体的作用，并学会了使用protobuf定义协议发送消息。同时，还提出了协议数据结构Packet、ConnectObj、NetworkListen和NetworkConnector类的概念。

网络编程可谓是网络游戏的基础，而我们后面即将搭建服务端框架，也是基于本章中的网络模型。也许读者还没有完全弄明白这些类的作用和使用方法，不用着急，在后面搭建服务端架构时，我们还会不断使用这些类，以加深对它们的了解。

第3章

线程、进程以及Actor模型

本章开始涉及游戏编码的一些基本元素，重点放在基础的架构上，不同的游戏类型采用不同的架构方式，本章将讨论这些架构的特点，设计出本书的游戏架构。本章包括以下内容：

- 了解各种游戏架构。
- 了解什么是游戏主循环。
- 理解进程和线程。
- 了解Actor模型，以及Actor模型是如何通信的。

3.1

游戏架构概述

有了网络编程的基础，现在我们来设计游戏架构。要设计好一个架构，需要先来了解常用的游戏架构。

3.1.1 无服务端游戏

单机游戏没有服务端，所有数据都存储在本地，所有判断也是在本地完成的，玩家不会与其他玩家进行交互。单机游戏不存在通信，不需要网络编程。除了单机游戏之外，还有一种无服务端游戏架构——P2P架构。

在P2P架构中，将所有客户端作为终端，是一种点对点的架构。P2P架构可以是一个完整连接拓扑架构，每个客户端与其他客户端之间都有连接，形成网状节点，信息可以直接在用户之间交换，而不需要通过服务端。所有终端拥有完全一样的全部数据，接收完全一样的协议。

P2P架构需要制作一个非常强大的客户端。如果每个客户端均要与所有终端保持连接（例如有1000个客户端），那么其效率显然会打折扣。所以，此架构适用于局域网或者少数人的对战游戏。

对于游戏来说，如果存在服务端，服务端可以充当仲裁者。但P2P架构没有一个仲裁者，在某些情况下，逻辑判断会存在争议。例如，两个玩家在PK时，其中一个玩家只剩下最后一滴血，在他使用回血技能的同时，另一个玩家对他释放了攻击技能，那么如何在两个终端都保证其判断的结果一致呢？为了解决该问题，P2P架构需要在以下几个方面做到非常精准，达到所有终端的协议必须一致，包括协议号与协议顺序，另一方面，每个终端上的随机种子需要完全一样等。

P2P架构并不是主流的手机游戏架构，有一个大致的了解即可。虽然P2P游戏不多，但值得一提的是，P2P架构衍生出了新对战游戏的架构，以及一种新的数据同步方式——帧同步。简单来说，帧同步方案是在极短的一段时间内（例如200毫秒之内），服务端收集所有客户端的操作，生成一个操作序列，并将这个操作序列发送到每一个客户端。帧同步解决了两个很重要问题：第一是网络响应延迟问题，第二是服务器压力问题。帧同步将计算放置到了客户端，释放了服务端运算压力，更进一步降低了网络延迟，对于竞技类游戏，帧同步方案是首选方案。

3.1.2 单进程CS架构

与P2P相反，CS（Client/Server）架构需要一个非常强大的服务端，图3-1展示了CS模型的架构。服务端对各个客户端发送来的数据进行验证，判断其合理性，并执行正确的操作，再把结果反馈给一个客户端或者全部客户端。

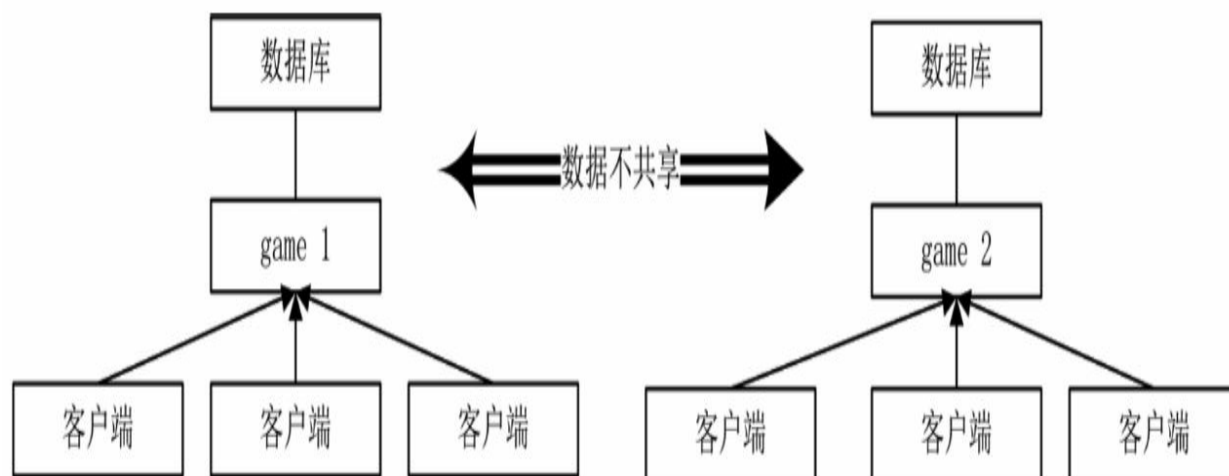


图3-1 CS模型的架构

大部分手机游戏的服务器使用CS架构，通信采用长连接或者HTTP。随着大型游戏的兴起，MMO类型的游戏服务器变得更复杂一些，它可能采用功能服务器的划分。

我们玩手机卡牌游戏时，通常可以看到每周开新服的情况，这类游戏多半是单服CS架构。单服CS架构的开发相对简单，它把所有功能都集中在一个进程中，大部分游戏为了开发效率不使用多线程，以单线程为主，降低了服务端宕机的风险。这类游戏客户端连接到指定的服务端，数据在服务端进程之间不共享。

3.1.3 多进程CS架构

图3-2是一个多进程架构，除了game逻辑服务端之外，还共用了login登录服务端、数据库、日志系统等。在这个架构中，可能还有副本进程，有时为了方便使用，也可以直接将副本进程合并到game逻辑服务器中。举一个简单的例子，棋牌游戏就可以采用这种方式，登录大厅之后，就可以创建不同的副本，不论你登录哪个服务器，操作流程都是一致的。

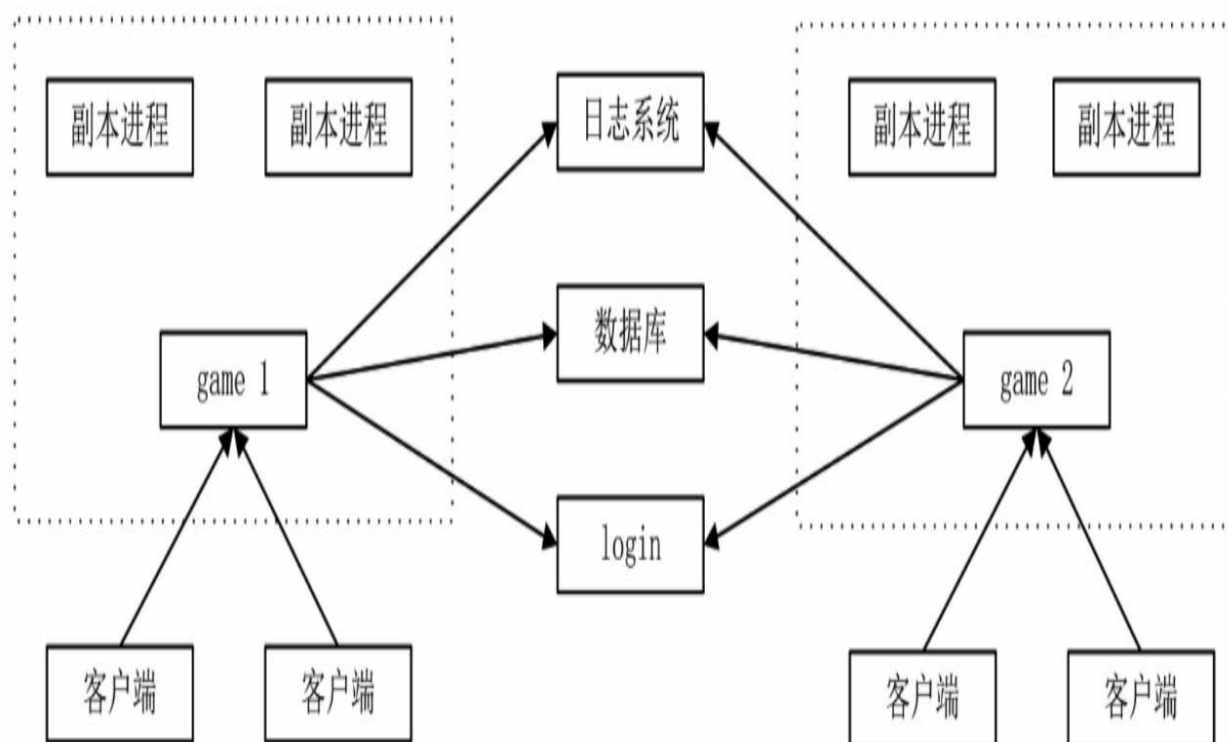


图3-2 CS多进程架构

框架瓶颈

虽然有许多种游戏架构，但是采用哪一种架构取决于要实现的游戏类型。将游戏按空间划分大致可以分为3种：

（1）滚服游戏。滚服游戏可以认为是一种平行空间的游戏方式，因为一个游戏世界容纳不了太多人，过一段时间就新开一个服务器，每个服务器的世界都一样。因为不断地增加新的服务器，用“滚服”来形容非常贴切。

（2）副本游戏。在副本游戏中，会为玩家专门开辟一个地图或空间实例。例如棋牌，为每一桌可以开一个副本空间。

（3）大图分割空间游戏。将一张大图分割成几个处理块，或者将不同地图放到不同进程进行处理。这类游戏又分为两种类型：一种类型如《剑侠情缘网络版叁》，不同的城镇在服务端可能在不同的进程中；另一种类型如《魔兽世界》，采用了Big World（大地图）的方式，所有的人看似都在同一个地图上，但其实这个地图像农田一样被划分成了多个小块，每个小块由不同的地图实例管理。

几乎大部分游戏都会落在这3种空间分类上，手机端的多人在线角色扮演类游戏一般是滚服游戏。竞技类游戏（如《王者荣耀》）本质上采用了副本方式。而《剑侠情缘网络版叁》类游戏在不选服的情况下容纳了大量的玩家，则是分割空间方式实现的。游戏类型的不同，其瓶颈各有不同。下面我们分别来讲述这3类游戏的瓶颈。

3.2.1 滚服游戏

滚服游戏也可以理解为空间复制。相同的游戏数据在不同的服务端实例上。以卡牌类游戏为例，给定一个服务端列表，玩家可以登录任何一个服务端，每个服务端的数据不相互冲突，每个服务端的游戏逻辑是完全相同的。

这类游戏技术的实现较为简单，只需要多开服务端，用简单的单进程CS架构就可以实现。但它有一个缺点，玩家承载是有限的。如果每个平行空间承载3000人，经过一段时间的游戏之后，玩家流失，某个平行空间可能只剩下不到100人。这时，我们需要采用的方式就是合服。随着时间的推移，服务器1的人会越来越少，这时需要对服务器1和服务器2的数据进行合并，即合服。

合服是滚服游戏一定要考虑的问题，一开始就要考虑的问题是合服时的数据库处理。因为每个服务端的数据不受彼此影响，所以在设计的最初就要想到，即使不是同一空间，也需要使用不同的角色名字，道具的ID生成方式也要唯一，以便于合服。

滚服游戏中的瓶颈在于每个服务端处理的玩家数量有限，所以需要不停地开服、合服，使服务端的硬件资源使用达到最佳状态。

客户端在服务器列表中选择一个游戏服务器game进程进行通信，该服务器支持以下几种功能：与登录服务器通信、与数据库通信以及实现必要的游戏逻辑。滚服游戏一般支持的玩家有限，一台服务器平均支持同时在线3000人左右。

在滚服游戏中，服务器两两之间是无须通信的。

3.2.2 副本游戏

副本游戏是对同一地图的数据进行复制，生成新的实例，但并不是所有生成多个实例的游戏都是副本游戏。

有两个需求产生了副本空间，一个是策划层面的需求。产生多个实例空间减少了地图中资源的争夺，这种情况往往会发生在刚进入游戏时，例如游戏有多个新手村，打怪升级时需要某种Boss，而Boss数量有限，引起玩家争夺资源。滚服游戏也可以有多个新手村，但它并不是副本游戏。

另一个是策略和技术方面的需求，例如斗地主时的多个房间。推图游戏中的每个可以探索的战斗关卡在服务端都会生成一个可以探索的实例空间，每个空间的数据独立，以保证玩家能独立游戏。这里说的“可以探索的实例空间”是指3D场景空间，如果你的推图游戏只是一场简单的战斗，那么可以不用额外建立副本空间。

这样说起来有点抽象，以《王者荣耀》为例，它就是采用了副本游戏的方式。众所周知，《王者荣耀》的玩家每日在线千万，将千万级的玩家放在同一个线程中是绝对不可能的，它必然采用了类似卡牌类游戏创建房间的架构，也就是说这些房间（游戏地图）分布在许多台物理服务器的许多进程上。和棋牌类游戏一样，这些房间的规则都是一样的，只需要在服务端编写一个地图类，就可以采用相同的规则创建出所有的地图。《英雄联盟》游戏也是如此。

3.2.3 大图分割空间游戏

分割空间是把一个世界的实例分成几部分，放在一个或几个进程中。采用空间分割方式的游戏一般来说是大型游戏，将一个巨大的游戏世界分割成几个实例。

在实际制作游戏时，我们很少看到真正只有一张Big World地图的游戏。分割一张地图有一个比较难处理的问题，玩家移动的时候，如果正好处于分割空间的分界线上，就会产生边界问题。可以用九宫格的方式来处理这个问题，就是向本地图为中心的九宫格发送一定边界范围内的同步信息，但处理起来依然很麻烦。为了避免处理这个问题，空间分割就演变成了将某些地图放在A服务器，将另一些地图放在B服务器，以此来减轻每个服务器受到的压力。

分割空间类的游戏架构与滚服类的游戏架构完全不同，在分割空间类的游戏中，不同的地图放在不同的进程，有时可能物理机都不相同，这时单进程的game服务器被拆分成了几个game服务器，而一组game服务器构成了整个架构。一旦出现瓶颈，可以进行拆分。例如，某个进程管理A、B、C、D四张地图，当物理机产生计算或内存瓶颈时，game又可以再被分割，A、B用旧的服务器，C、D再组成一个新的game服务器。

分割空间类的游戏架构不需要合服，因为它不存在玩家人数的限制，对于客户端而言，它好像只有一个服务端入口，不需要选服，实则其背后有庞大的设计。这也就是常说的全球服。

设计游戏框架

前面讨论了一些基本的游戏类型，又讨论了各种游戏可能存在的瓶颈，有这么多服务端架构，应该采用哪一种服务端架构呢？这个问题实际上是由游戏内容来决定的。本书从技术层面出发，目标是做一个难度相对较大的全球服。理解了全球服的思路，其他架构都是全球服的缩影。

下面来展望一下即将实现的游戏架构以及它的基本结构图，如图3-3所示。

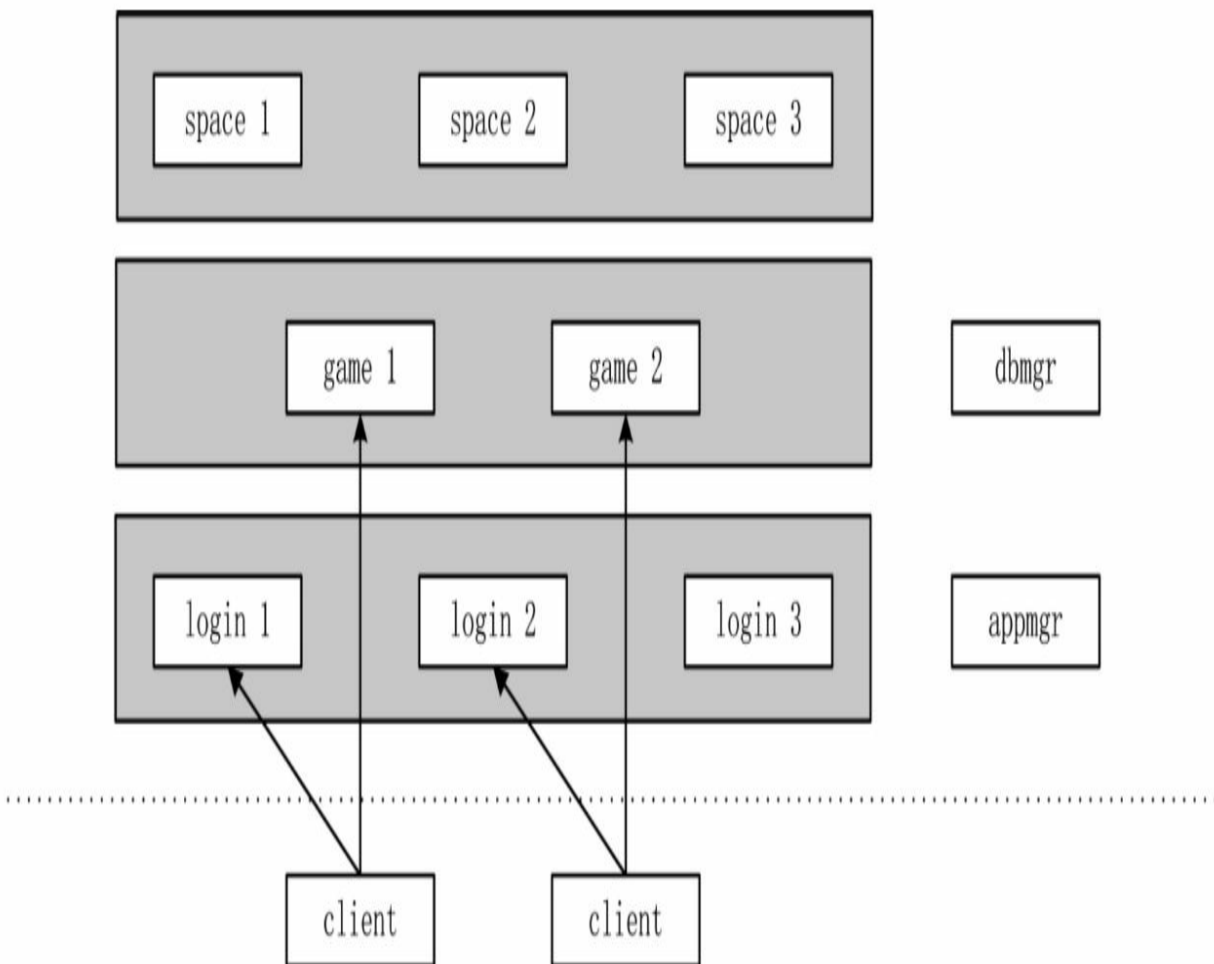


图3-3 游戏框架

本书要以分割空间的方式实现一个不分服的架构，在这个架构之下可以再深度扩展，不仅可以实现一个MMORPG，也可以实现类似《王者荣耀》的对战类游戏。当然，既然可以分割空间到不同的进程，我们也希望它可以合并空间到一个进程，也就是说这个框架也可以实现常规的滚服游戏，这是一个可伸可展的框架，可以满足大部分游戏的需求。

在开始编码之前，首先需要厘清这个架构之间的进程关系，以及每个进程需要做些什么，再开始编码。下面列个清单说明一下其服务器结构。

(1) 登录验证服务器login，需要N个。

(2) 用于代理中转的game服务器，需要N个，玩家与某个game进行网络通信，每次登录的game可能并不一样，这样就保证了game的均衡。

(3) 我们把世界分成若干个空间，所以需要有N个space（空间）进程。space进程放置着真正的地图实例，确保玩家在space服务进程上是均衡的。例如有10个玩家，开了10个副本，有5个space进程，那么相当于每个space实例上开了两个副本，以达到均衡的目的。

(4) appmgr进程，需要1个，用于进程的管理。

(5) dbmgr进程，需要1个，用于数据库的统一存储。dbmgr进程是否需要独立成为一个单独的进程是不一定的，根据游戏类型的不同而不同，这部分功能也可以归到每个game上。为了区分功能，让代码更清楚，本书中将这部分内容独立成了dbmgr。

要实现全球服架构，达到不分服的目的，尽可能让一组服务器承载更多的玩家，服务端需要解决的首要问题是“聚合”问题。这里说的“聚合”是什么意思呢？

聚合就是那些让玩家聚在一起，以完成一些需要同步数据与操作的功能。在处理这一问题上，space服务进程并不是单纯意义上的地图，它要实现的是一种聚合，而地图只是恰好是一个聚合，但聚合不仅仅限于地图。具体一点，帮派、聊天系统也是一种聚合，这种聚合并不是物理地图上的聚合，而是一种逻辑上的聚合，一旦帮派成员有操作发生，需要同步到帮派内每个在线成员的客户端。

前面讨论的副本空间的框架也有两种实现方式，一种是客户端维护一个网络连接，只与game进行通信；另一种是客户端需要维护两个网络连接，同时与game和副本进行通信。也可以把副本服务器称为战场服务器，典型的用法是在《英雄联盟》中，登录、匹配数据在一个进程上，而进入地图之后，它在另一台服务器上。这两种结构各有其特点，并没有好坏之分，只是需求不相同。

本书为了实现兼容MMORPG与MOBA类游戏的双重功能，将采用第一种方式，将game服务进程视为网关服务器。顾名思义，网关是用来连接两个网络的，网关服务器是一个中转服务器，它是连接客户端与地图（战场）服务器之间的桥梁。

为了循序渐进，初步目标是实现一个login服务器，完成第一个登录功能。后期会再次对这个架构进行扩展，让它变成一个更复杂的分布式服务器，并且以最新ECS概念的框架来重构它，以达到代码简洁，又能快速编码的目的。

在图3-3中，这组服务器同样适用于棋牌游戏，举例来介绍它的流程：玩家通过login服务器进行登录，在login服务器上建立一种选择game服务器的策略。玩家账号验证成功之后，会为其选择一个适合的game登录。当玩家需要进入棋牌房间时，向appmgr询问房间的space进程，space进程为其创建一个房间实例。而game这时相当于一个中间代理服务器，连接了客户端与这个特定的space服务进程的通信。

本节的内容有点多，意在展示本书示例的整体架构，随后会逐一分析和实现这些步骤。

根据图3-3，在这个架构中，简单的配置都需要5个进程来支持。也许读者之前只做过一个进程程序的框架，但是别被吓倒，先通过例子了解每个进程的作用，再用一些编码上的模型与技巧让它们合并到一个进程中，以方便我们进行测试，听上去是不是很有趣？随时可以拆分这些功能到5个进程中，也可以合并这些功能到一个进程中。

下面要真正开始编码了。

如果读者没有真正接触过游戏编码，首先要了解的概念是游戏主循环（Main Loop）。

游戏主循环

不论服务端框架有多少个进程，每个进程一定要有一个游戏主循环。单线程的游戏服务端结构中，一个主循环中需要实现两个主要功能：

(1) 数据收集。对于服务端来说，操作是由网络层发送过来的，所以服务端需要不断监听网络层是否有新的数据到来。

(2) 不断更新、计算世界的状态，根据收集来的数据更新游戏世界，并把这些更新数据广播给一个或者多个终端。

客户端比服务器要多做一些功能。除了这两个主要功能外，客户端还要监控设备的输入操作和实现绘制——世界绘制、NPC绘制、玩家模型绘制、UI绘制等。

对于服务端来说，多线程框架与单线程框架要做的事情是一样的，只是它把这些功能放在多个线程中，网络层一个线程，游戏世界一个线程，等等。回顾一下之前网络层的Select和Epoll的例子，在main函数里都写了一个死循环，这个死循环不断地进行网络层的数据收集。当收到退出的命令时，从这个死循环中退出。

一个简单的主游戏循环是这样的：

```
while (Game.IsRun) {  
    UpdateNetwork();  
    UpdateAllWorlds();  
}
```

可以认为服务端的函数UpdateNetwork是一个输入，毕竟在服务端是不可能用鼠标操作的，它是一个处理输入的IO模块。UpdateAllWorlds根据收到的协议来更新游戏世界的地图。在游戏中，一个根本的概念是地图，这里所指的地图就是你身处的3D场景，在棋牌游戏中就是所在的房间。更新地图一般是由底层到上层更新的，先更新地图中的所有Player的状态。如果有NPC，那么还需要更新NPC的状态、AI的状态等。

在游戏编程中，最外层一定是一个死循环。只要你还在游戏中，这个循环就要一遍一遍执行下去。这个主游戏循环有一个缺点，每次更新地图之前，需要等待网络层IO处理完成才可以进行下一步，毕竟函数需要一个一个地执行。如果有两个线程，一个处理UpdateNetwork，另一个只负责UpdateAllWorlds，是不是会更大幅度地提高效率呢？

因此，在开始时就要引入线程。

理解进程和线程

本章要为搭建服务端架构做准备了。这里所说的服务端是指软件层面的服务，而不是物理机。服务端并不是单纯地指一个进程，如果你之前没有接触过进程，那么可以理解为一个进程就是一个EXE。服务端可能是由多个进程组成的。对于客户端来说，服务端似乎看上去是单进程的，实则不然。服务端的进程可能运行在一台物理机上，也可能运行在多台物理机上。

在搭建架构之前需要了解一些基本概念，基础的概念就是进程和线程。也许你听说过游戏服务器的架构是多线程多进程的，或者单线程多进程的，这是什么意思呢？本节来帮助读者弄清楚这些问题。

3.5.1 进程是什么

简单来说，进程拥有一个Id，进程Id（PID）是进程唯一的标识码。我们所编写的程序代码，每一个包含main入口函数的程序都认为是一个进程，在Windows系统中，.exe后缀的文件就是一个进程。

一个游戏服务器所需要的基本功能：登录、副本、任务、道具、日志系统、数据库基本的读取存储功能等。对于一个单进程架构而言，也就是说在主线程中至少需要实现以上几种基本功能。这使得单进程单线程的方式承载压力巨大，能处理的客户端非常有限。因为我们需要在每一帧去更新世界和角色数据，而随着在线角色的增多，每一帧处理这些功能的时间会变长。单线程是串行的，如果有1000个人等着验证，那么只有先处理完这1000个人的请求，再处理世界逻辑的更新。

为了让服务器更为高效，我们可以将游戏服务器框架拆分为多进程架构，例如将登录、数据库操作、游戏世界分成3个进程，组成一个简单的多进程服务器框架，简称多服架构。

它的好处在于，当服务端收到登录协议时，是由登录进程去处理的，游戏逻辑在游戏世界进程中不会受到影响，即使有1000个人等着验证账户，游戏世界也不会因此而阻塞，因为它们在不同的进程中。

多进程游戏服务器的架构是非常灵活的，不同的公司、不同的人可能写出来的架构大不相同。但它的原则是不变的，简单而言，就是把本来合在一起的功能按一些其他的需求拆分为多个进程。进程与进程之间使用Socket进行通信。但是这种拆分对于客户端是不可知的，也不需要知道。

我们可以这样理解多进程——服务端有多个功能的服务器。每个功能服务器就是一个进程。多进程是游戏服务的首选架构，但在多进程架构之下有两种模式：多进程单线程和多进程多线程。

对于多进程单线程，虽然有多个功能服务器，但每个服务器都是单线程的。同理，对于多进程多线程，即某些或多个功能服务器是多线程的。这两种模式各有优缺点，多线程可以更好地利用多核，而单线程的编码比多线程更为简单。

3.5.2 线程是什么

一个进程至少拥有一个线程，即主线程。每个线程拥有自己的调用堆栈，是CPU调用的基本单位。我们从CPU调度的角度来看单线程和多线程有什么不同。图3-4左侧为一个单线程的服务端，右侧为一个多线程的服务端。

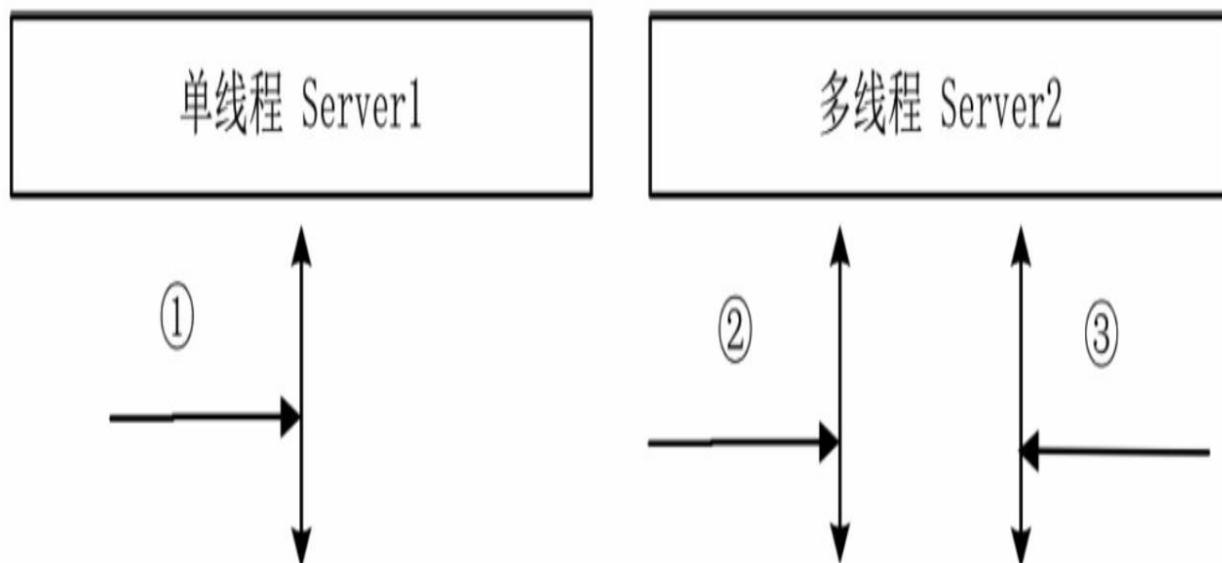


图3-4 单线程与多线程

对于Server1、Server2两个进程，Server1拥有一个线程，Server2拥有两个线程。当这两个程序同时运行时，CPU的调用顺序可能为，前一秒在线程1处，后一秒在线程2处，处理完线程2，它可能跳回线程1继续向下执行，再进行线程3的调度。每个线程互不对对方产生影响。

线程1进行IO请求等待时，CPU就会自动执行另一个线程，线程1得到IO请求的资源后会等待CPU继续执行线程1。CPU的这种切换叫上下文切换。

CPU均匀地调度所有线程，当其中一个线程被IO阻塞时，CPU不会等待，会马上转向另一个线程。这就是多线程比单线程更快的原因。多线程可以更好地均衡CPU的负载。

既然多线程比单线程效率更高，为什么还有人用单线程，都采用高效率的多线程不是更好吗？

原因很简单，是因为线程之间不共享内存，不共享内存会带来编码上的不方便。

我们一直强调每个线程有每个线程的调用堆栈，例如A线程的数据，如果同时有两个线程进行写操作，就会出现一些无法预估的宕机错误。在编程上为了避免这种错误，有两种常规的处理办法。

(1) 加锁。每种编程语言都有加锁语法。对于那些跨线程的数据进行加锁，这种加锁行为相当于让访问数据的线程进行排队，一个一个串行访问数据。虽然加锁可以解决线程访问的问题，但是实际上在处理这些数据时，它变成了“单线程”，而不能并发。

(2) 消息机制。永远都不正面访问线程中的数据，而是通过消息的方式来调用。请求操作到来时，我们把数据以消息的方式保存下来，处理完成之后，再用消息的方式传递出去。

在消息机制中，如图3-5所示，只有在进入或取出队列时数据会加锁，控制住了加锁的使用范围，而在处理任何一个协议时，因为是本线程内的操作，并不需要加锁，所以不会造成加锁泛滥。

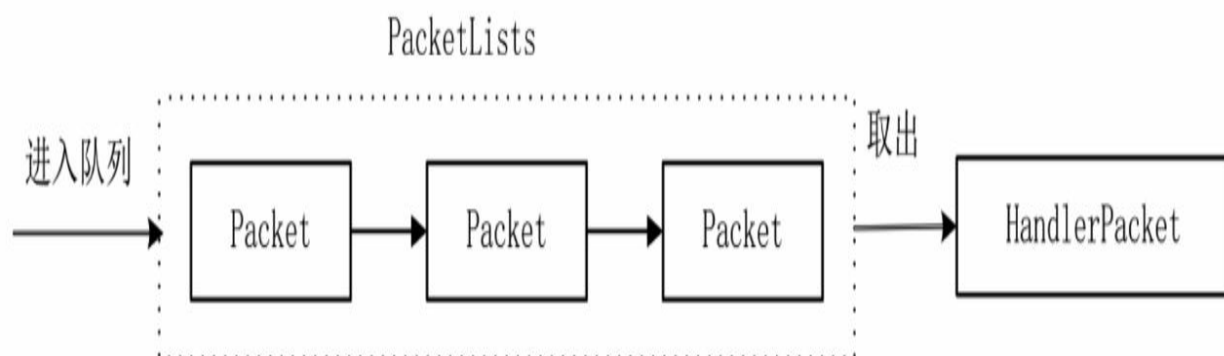


图3-5 消息机制

在游戏编程中，常见的一种线程使用方式为：例如有1000个DB数据需要存储，发起1000个线程，每个线程处理一个，处理完成之后销毁线程，回调到主线程。类似的需求很多，例如发送邮件。这种方式使用了

并发处理数据，的确提高了效率，但如果出现回调函数，就避免不了加锁。因为回调函数在主线程中，从一个线程调用主线程函数时必须加锁。在编码中，有了一个锁，就会有无数个锁，随着游戏的功能越来越多，锁会越来越多，遗漏一处都会产生宕机。这也是很多人不喜欢用线程的原因。

总而言之，多线程更好地利用了多核的优势，充分利用了IO等待的时间。在编程上，单线程可以随时取走你想取的数据，不需要加锁，多线程则不能，而且有诸多限制。

本书的框架采用多线程方式，为了避免频繁加锁，消息机制是本书的首选。

现在，理解了进程和线程，可能会有人非常疑惑，不知道什么时候应该用多线程，什么时候应该用多进程，看上去它们能实现的功能类似。

前面我们讲过了，线程与进程对于系统资源管理方式存在着本质的区别，进程内部资源共享，这里指的资源包括文件、内存、CPU等。每个进程执行时都有一块内存区域。进程是CPU的最小调度单位，在多CPU物理机上，多线程比单线程更具优势。

线程不能独立执行，没有自己的内存地址。它是一种轻量级的进程，依附于进程，不拥有自己的PID。我们在衡量线程与进程时，一定要考虑到这些因素。

从逻辑层上来讲，进程和线程可以做的事情是相似的，一个功能在两个进程上可以实现，在两个线程上也是可行的。它们的区别在于底层而不是上层。好好利用线程是高效率游戏编程的开始。

3.5.3 C++标准线程库

还是用代码来实际体验一下线程。要完成这个例子，需要先了解一下C++的线程库。以往使用线程，为了方便可能会用到boost库的thread，但在C++11标准中，std::thread库对线程进行了包装，使用起来非常方便。

本节的源代码位于03_01_thread目录中，本章所有的例子都是在C++14标准下完成的。工程中有两个main测试，其中一个简单的测试示例在main.cpp文件中：

```
#include <thread>
#include <iostream>
void ThreadTest1( int value ) {
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
    std::cout << "thread test. value:" << value << std::endl;
}
int main1( int argc, char *argv[] ) {
    int value = 1;
    std::thread t1( ThreadTest1, value );
    //t1.join( );
    t1.detach();
    std::cout << "exit main" << std::endl;
    return 0;
}
```

将main1修改为main函数并进行编译。在前面对网络库进行测试时，我们使用到了std::thread，它需要包含一个头文件<thread>。

关键点1：退出方式

线程有两种退出方式：一种是调用::join函数；另一种调用::detach函数。

以上面的代码为例，在调用::join函数时，程序在关闭线程时使用了阻塞方式，只有等到ThreadTest函数结束之后，主线程才会结束。编译程序执行测试，可以看到在主线程退出之前打印输出了线程退出的

相关信息。而调用：`std::detach`函数不会阻塞主线程，它使线程从主线程中剥离出来，脱离了主线程的控制。可以看到主线程退出时线程函数还没有被调用。

关键点2：为线程创建参数

在调用线程回调时，必不可少需要用到参数。`std::thread`传递参数也相当简单。

```
int value = 1;
std::thread t1( ThreadTest, value );
```

函数定义为`void ThreadTest (int value)`。

关键点3：线程互斥

以一个例子来说明为什么需要线程互斥，以及如何互斥。其文件定义在`main2.cpp`中。

```
#include <thread>
#include <iostream>
#include <mutex>
int global_value;
std::mutex mutex_obj;
void ThreadTest( ) {
    //std::lock_guard<std::mutex> guard( mutex_obj );
    global_value++;
    std::cout << "\nthread test. global_value:" << global_value <<
}
int main2( int argc, char *argv[] ) {
    std::thread t1( ThreadTest );
    std::thread t2( ThreadTest );
    t1.join( );
    t2.join( );
    std::cout << "exit main" << std::endl;
    return 0;
}
```

这段代码创建了两个线程，两个线程都对全局变量做了一个++运算。程序运行起来结果是多种多样的，其中有一种结果如下：

```
[root@C7-Pf bin]# ./thread-testd
thread test. global_value:2
thread test. global_value:2
```

为什么会有这种情况？因为两个线程都没来得及打印之前，CPU在t1线程和t2线程同时执行了global_value++。但这不是我们需要的结果，我们想要的是串行运行的结果，任何时候都是累加的。

为了避免这种情况发生，需要用到std::mutex对象。std::mutex是C++中基本的互斥量，其头文件为<mutex>。

一个简单的加锁、解锁，调用lock和unlock函数：

```
std::mutex mutex_obj;
void ThreadTest() {
    mutex_obj.lock();
    global_value++;
    mutex_obj.unlock();
}
```

注意

mutex还有一个try_lock()函数返回bool值，该函数判断互斥量是否被锁住，如果正好被锁住，就不会阻塞进程，但不会执行try_lock() {…}中的代码。

我们对std::mutex对象使用lock进行加锁之后，可能忘记对mutex进行解锁，C++14提供了一种Lock类来规避这个问题。另一种加锁方式如下：

```
std::mutex mutex_obj;
void ThreadTest() {
    std::lock_guard<std::mutex> guard( mutex_obj );
    global_value++;
}
```

```
}
```

当一个`std::lock_guard`对象被创建后，它就会尝试去获得其参数`std::mutex`的所有权。`std::lock_guard`被析构，`std::mutex`被自动释放，所以不需要显式地释放`mutex`。`std::lock_guard`是一个区域锁，范围为`std::lock_guard`的生命周期，上例中，加锁范围为函数的生命周期，退出函数将解锁。

关于C++的线程就介绍到这里，更多细节可以阅读C++并发相关的文档。

Actor模型

除了知道线程是如何运行的外，我们还需要了解一个模型——Actor模型。Actor模型是为了并发而设计出来的。一个纯粹的Actor只接收消息，受事件驱动，然后根据消息执行相应的计算。其最大的特点是，大量Actor之间是相互隔离的，它们不共享内存。Actor之间有且只有一种通信方式，就是消息传递。在Actor模型中，对象的任何数据不对外，所有数据只有它自己可以修改。更为直白一点的说法就是，两个Actor之间不能互调函数。

当我们在使用线程时，因为有资源的竞争，需要加锁。同时，有两个或者两个以上的线程需要访问某个对象时，为了避免出错必须加锁。如果一个系统日渐发展壮大，又使用了多线程，那么到了后期，这些锁几乎是一种毁灭性的灾难，因为其不可维护。为了解决加锁的问题，本书中所有的对象都是基于Actor模型进行设计的。采用消息通信的方式来修改数据，避免了加锁。

这种设计还有一个好处在于可以进行单元测试。很多情况下，游戏公司很少进行自动化测试。主要是因为代码耦合太强，没有办法进行单元测试，也不知道如何进行单元测试，测试的结果应该如何衡量，这些都是难以解决的问题，但Actor模型很好地解决了这些问题。

因为这种模型使得逻辑的耦合性降低，一个Actor不会与其他任何Actor有耦合，所以在测试时可以根据需要创建任何一个需要测试的Actor。

例如，有一个ChatSystem类负责聊天系统。一般的做法可能是这样的：定义一个聊天协议，包括3个基本值：PlayerId、ToPlayerId和Context，分别表示发起聊天的人、发送对象和内容。当聊天协议到来

时，通过玩家管理类PlayerManager找到发起对象，称为A对象，再找到目标对象，称为B对象，再给B对象的客户端发送一个聊天协议

(PlayerId、ToPlayerId和Context)，客户端收到这个协议时，界面上就会显示聊天的内容。当然，这个过程还需要这两个玩家在同一个进程中，要不然从哪里去找目标呢？

在Actor模型下，实现方式完全不同。因为Actor模型中，ChatSystem类根本得不到PlayerManager类。一旦ChatSystem类可以取得PlayerManager，就建立了耦合。为了实现聊天的功能，我们需要分几步：

(1) 当玩家上线时，向ChatSystem类注册一个消息，这个消息告诉ChatSystem类“我上线了”，这时玩家可以在ChatSystem类中注册一些需要的信息，例如所在的网关服务器。在不同模型下，注册信息可能不同，其目的是让ChatSystem类知道向哪个进程发送消息，该玩家可以收到消息。

(2) 在通知ChatSystem类“我上线了”的同时，ChatSystem类取得了该玩家的所有好友列表。

(3) 有了前两个步骤，现在客户端发送一个协议 (PlayerId、ToPlayerId和Context) 给ChatSystem，它不再需要向PlayerManager获取数据，因为所有的数据是由它本身维护的。这种Actor的方式可能会使内存数据增加，但它减少了耦合性。

当我们需要测试这个聊天系统时，可以在robots机器人工程中直接创建一个ChatSystem类，模拟玩家的登录数据向它发送登录消息，模拟玩家聊天的协议以便进行各种测试。如果以传统的方式实现聊天功能，测试ChatSystem类的功能时，需要先建立一个PlayerManager类，而PlayerManager类可能又依赖于其他的类，这样的耦合性是不可能简单地实现单元测试的。

Actor模型是基于消息的，单个Actor的原型分为3部分：消息队列、行为和数据。消息队列又被称为MailBox或邮箱，邮箱比较贴切，因为邮箱是协议的中转地，发起邮箱的双方没有耦合。为了便于编码，本书做了一些小小的修改，在程序中把邮箱设计为Message List，即存放消息协议的列表。简而言之，MailBox这块功能是用来收发协议的。同时，Actor还需要有一些行为定义，这些行为可以认为是对每个协议

不同的处理函数，Actor肯定还有一些自己维护的数据。一个Actor必定是由以上3部分组成的。

图3-6展示了多个Actor是如何配合工作的。

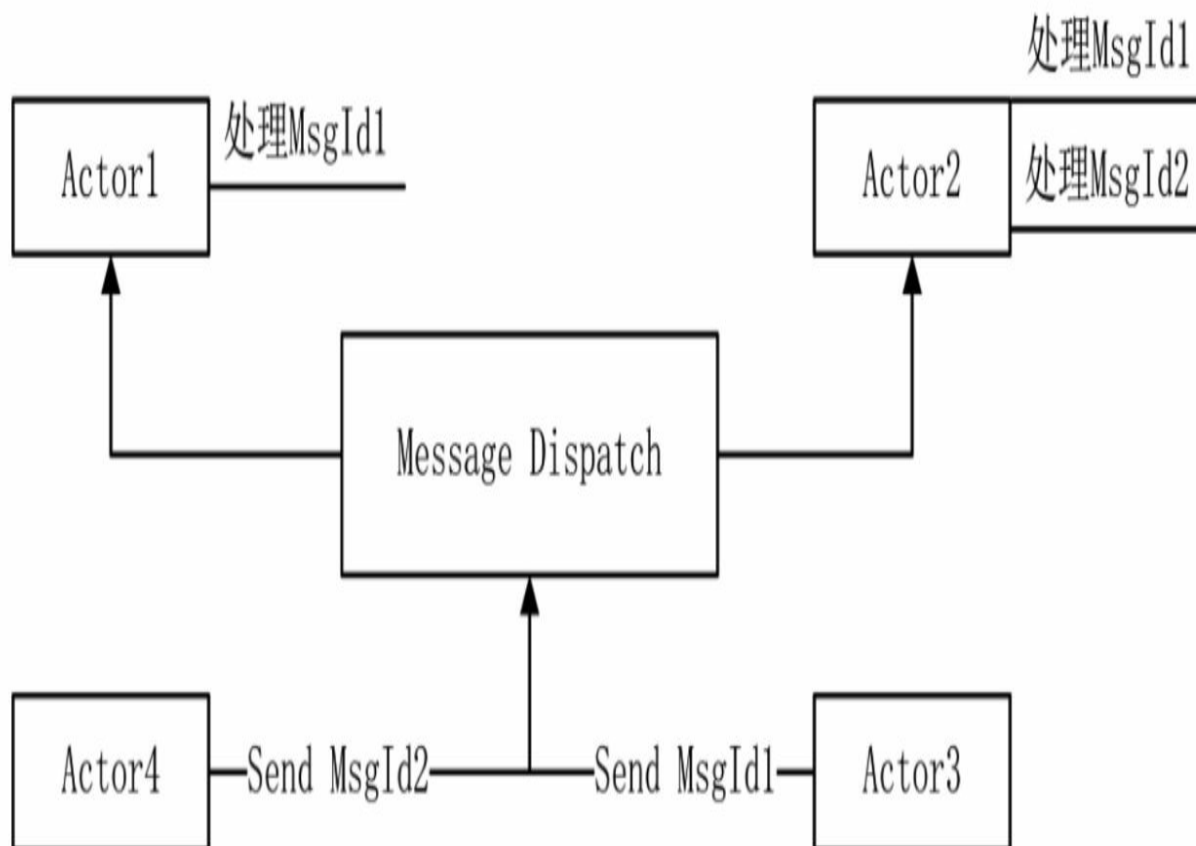


图3-6 Actor对象交互

多个Actor协作时需要加一个消息广播的机制。图3-6中的Message Dispatch用于消息分发。

图3-6中有4个Actor，Actor1关心MsgId1消息，Actor2关心MsgId1、MsgId2消息。当Actor3广播一个MsgId1消息时，Actor1和Actor2就会收到消息的通知，然后根据自身的特点决定当消息到来时自己需要完成什么样的操作。

还是以聊天系统为例，ChatSystem类生成时就决定了它需要关心至少3个协议（这里协议也可以称为事件）：第一个事件是玩家登录事

件，第二个事件是玩家离线事件，第三个事件是聊天事件。

聊天事件是如何到达ChatSystem类的呢？

当玩家对象Player通过底层网络得到一个聊天协议时，Player并不处理它，而是马上进行一个内部广播，向整个Actor系统的所有对象通知该协议到了。除此之外，Player什么也不需要做，Player并不关心最终这个协议是由谁去处理的，而关心这个协议的Actor得到这个协议之后会进行相应的处理。

Actor这种消息机制让各个功能模块完全独立，一个聊天系统随时可以加上，也随时可以抽取出来，类似一种插件机制。而且Actor机制可以让我们的服务器配置多样化，如果ChatSystem类可以在任何进程中创建出来，服务端的功能组合就会显得异常强大。

假设登录验证功能也是一个Actor模型，那么可以在需要时将登录验证功能独立到登录进程中，也可以将该功能放到网关进程中。如果整个游戏的所有功能都是由Actor对象组成的，那么可以将它们配置到一个进程中，也可以配置到多个进程中。这样既可以实现简单的滚服CS架构，又可以实现复杂的分布式服务器架构。

使用Actor模型还有一个很重要的原因，要利用Actor这个消息机制的模式充分发掘线程的效率。不要忘记了，Actor模型原始的功能就是为了提高并行效率。下面用一个例子来看看Actor是如何和线程一起高效工作的。

游戏框架中的线程

在本书的框架中，并不是直接简单地使用线程去执行一个函数、调用一个类，而是要对线程进行必要的扩展，以方便使用。首先明确目标，需要建一个线程管理类，实现如下功能：

- (1) 初始化游戏线程。
- (2) 关闭所有线程。
- (3) 更新所有线程。

既然有线程管理类，必然有一个线程类，实现如下功能：

- (1) 启动一个线程。
- (2) 关闭、销毁本线程。
- (3) 对线程中的所有对象进行更新，实现Update帧函数。
- (4) 管理“线程包裹”。

还需要建立一个线程包裹ThreadObject类，实现如下功能：

- (1) 更新自身数据，实现Update帧函数。
- (2) 销毁自己。

线程包裹类是线程上的最小单位。图3-7展示了这3者之间的关系。

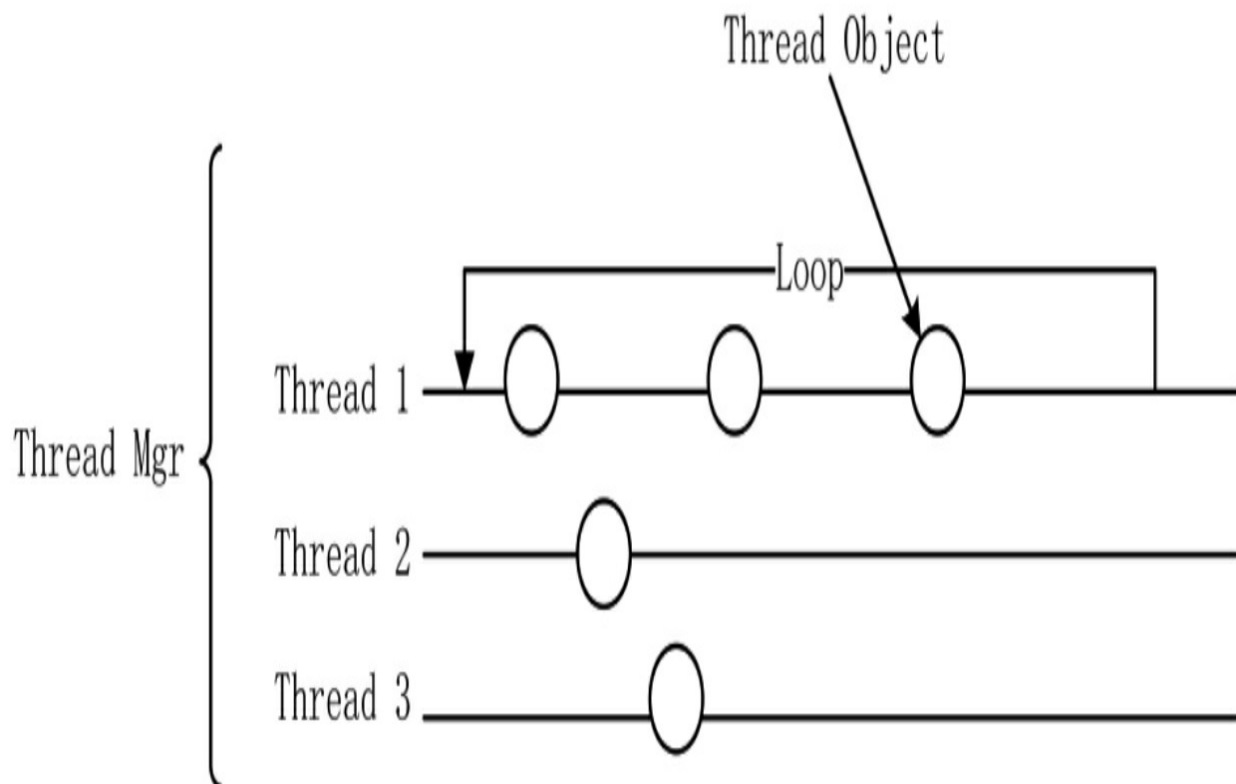


图3-7 线程与数据结构

图3-7中的Thread 1、Thread 2和Thread 3分别表示3个独立线程，其中Thread 1维护了3个线程包裹对象。线程就像一个流水线一样，每个ThreadObject实例就是上面一个个包裹，为了方便理解，就叫它线程包裹类。线程不关心这些“包裹”中的内容是什么，只是每帧调用Update函数时，对“包裹”对象进行更新。而线程管理则管理着Thread 1、Thread 2和Thread 3，线程管理器并不关心每个线程中有多少个线程包裹。

为了理解游戏线程，创建了一个新的工程，在本节源代码03_02_engine_thread目录中。在这个目录中有3个工程：login、robots以及基础libserver库。login进程打开了一个服务器监听端口；robots创建了10个Robot对象连接到login，这10个对象均匀地放在3个线程中；而libserver提供了网络与线程两个基本功能。

本节的例子目的在于明确基础类之间的关系，仅仅实现login进程的监听与robots进程的连接功能。编译目录，启动login和robots，在

login控制台会产生10个连接打印。

3.7.1 包裹类ThreadObject

每个线程包裹都是一个以Actor为原型的类，它们互不影响。基类ThreadObject定义在libserver工程的thread_obj.h文件中。从本节开始，例子中不再有Network库工程，替代它的是底层库libserver，在后面的章节中将一步一步完善、重构、优化这个底层库，最终成为一个高效运作的游戏底层库。先来看看在本例中线程包裹类的定义：

```
class ThreadObject : public IDisposable {
public:
    ...
    virtual bool Init= 0;
    virtual void RegisterMsgFunction( ) = 0;
    virtual void Update( ) = 0;
    ...
};
```

在这个基类中有3个virtual函数，也就是说，我们希望所有继承该类的子类都要实现这3个函数。这3个函数的具体说明见表3-1。

表3-1 线程对象的函数列表

函 数 名	说 明
Init	初始化函数
RegisterMsgFunction	注册自己感兴趣的协议
Update	帧函数，更新数据

每个Actor必定有自己感兴趣的协议类型，函数RegisterMsgFunction用来实现Actor的协议注册，帧函数中实现不同的Actor的不同更新操作。

3.7.2 线程类Thread

线程类定义在thread.h文件中，主要函数如下：

```
class Thread : public SnObject, IDisposable {
public:
    void AddThreadObj( ThreadObject* _obj );
    void Start( );
    void Stop( );
    void Update( );

private:
    bool _isRun;
    std::list<ThreadObject*> _objlist;
    std::thread _thread;
};
```

该类维护了一个std::thread，提供了启动线程与结束线程的函数。其中，AddThreadObj函数是将一个指定的线程包裹类ThreadObject实例放在本线程中，属性Objlist维护了这些线程包裹类实例。线程类并不关心这个流水线上的包裹有什么用，它只需要在更新时遍历Objlist中的所有对象，调用这些对象的Update帧函数即可。

```
void Thread::Update( ) {
    _thread_lock.lock( );
    std::copy( _objlist.begin( ), _objlist.end( ), std::back_inser
    _thread_lock.unlock( );
    for ( ThreadObject* pTObj : _tmpObjs ) {
        pTObj->Update( );
        ...
    }
    _tmpObjs.clear( );
    std::this_thread::sleep_for( std::chrono::milliseconds( 1 ) );
}
```

前3行代码将线程中管理的对象暂时取到一个临时的集合中。这里做了加锁操作，因为在进程运行中随时可能会向线程内部增加包裹，这个增加的需求可能来自本线程内部，也可能来自线程外部，所以必须加

锁。但代码中并没有把unlock放到函数结束，如果等到所有的数据都完成了Update操作才将锁放开，那么只会降低效率。因此，在使用时为当前对象做一个临时的拷贝。

在Thread：：Update函数中，对其线程上的每一个ThreadObject实例调用ThreadObject：：Update函数进行数据更新。

3.7.3 线程管理类ThreadMgr

线程管理类定义在thread_mgr.h中，定义如下：

```
class ThreadMgr :public Singleton<ThreadMgr> {
public:
    void StartAllThread( );                // 启动所有线程
    bool IsGameLoop( );                   // 所有线程是否还在运行中
    void NewThread( );
    void AddObjToThread( ThreadObject* obj );
    ...
private:
    std::mutex _thread_lock;
    std::list<Thread*> _threads;
};
```

线程管理类有两个主要任务：

(1) 每帧检查一下每个线程是否还在工作，若线程全部退出，则进程退出。

(2) 管理多个线程对象。当有新的线程包裹对象加入线程时，丢给管理类处理，会将新对象放置到一个合适的线程中。ThreadObject本身并不知道自已会加入哪个线程中。

线程管理类维护了一个线程对象列表，AddObjToThread函数将一个包裹类加入线程中，外部并不知道该对象加到哪个线程中，函数内有一个平衡算法。为了实现均衡，所有线程上的包裹类是由线程管理类来分配的。具体函数实现就不在这里分析了，相关代码并不复杂。值得一提的是，线程管理类用到了设计模式上的单例模式，因为它只需要有一个实例。

关键点：Singleton

在定义ThreadMgr类时使用到了Singleton模板类，它的定义为：
class ThreadMgr: public Singleton<ThreadMgr>。如果读者曾经接触过设计模式，就会发现这是一个单例模式。单例模式确保一个类只有一

个实例，提供了一个全局指针来访问唯一实例。

在libserver工程的singleton.h文件中编写了一个单例模板。其定义如下：

```
template <typename T>
class Singleton {
public:
    template<typename... Args>
    static T* Instance( Args&&... args ) {
        if ( m_pInstance == nullptr )
            m_pInstance = new T( std::forward<Args>( args )... );

        return m_pInstance;
    }
    static T* GetInstance( ) {
        if ( m_pInstance == nullptr )
            throw std::logic_error( "the instance is not init, please init first" );
        return m_pInstance;
    }
    static void DestroyInstance( ) {
        delete m_pInstance;
        m_pInstance = nullptr;
    }
private:
    static T* m_pInstance;
};
template <class T> T* Singleton<T>::m_pInstance = nullptr;
```

在该模板中定义了3个函数：创建、取得和销毁。创建类时使用了可变参模板，因为并不知道具体类是否需要提供参数，提供多少参数，所以使用了可变参，使用时调用std::forward来展开参数。单例模板使用起来非常简单，还是以ThreadMgr为例。

```
ThreadMgr::Instance(); // 创建
auto pThreadMgr = ThreadMgr::GetInstance(); // 取得实例
pThreadMgr->DestroyInstance(); // 销毁实例
```

在使用单例模式时务必要非常小心，因为框架是一个多线程工程，线程管理类在任何线程上只要调用ThreadMgr::GetInstance函数就可

以得到，所以对于线程管理类中的函数进行了加锁操作。

3.7.4 libserver库与游戏逻辑

前面阐述了服务端的架构，它是由登录服务器login、管理服务器appmgr、逻辑服务器game、数据库服务器dbmgr和空间服务器space共同组成的。这些服务器在逻辑层的功能千差万别，但其基本的架构是完全一致的，它们每一个进程都有无数个线程，有网络处理能力，有收发网络协议的能力，两两之间可以通信，也可能会监听一个网络端口或者去连接其他服务器等。这些是每个服务器的基本功能。

把这些基本功能提取出来，变成整个框架共用的代码，即为所有进程编写一个通用libserver库，以提供这些通用服务。在03_02_engine_thread目录的工程中，libserver库提供的服务包括：

(1) 网络层：打开监听或进行网络连接。

(2) 使用Actor模型来处理协议。

(3) 管理多线程，每个具体的线程需要做什么不是libserver底层库需要考虑的事情，库相当于只提拱了一个运行平台，而每个具体的线程包裹对象要做什么是由具体实现类来实现的。

(4) 提供一个基本的ServerApp类来创建不同的服务器。

类ServerApp定义在libserver工程的server_app.h文件中。它是进程中创建的第一个类，完整的定义如下：

```
class ServerApp : public IDisposable {
public:
    ServerApp(APP_TYPE appType);
    ~ServerApp();

    virtual void InitApp() = 0;
    void Dispose() override;

    void StartAllThread() const;
    void Run() const;
    void UpdateTime() const;

    bool AddListenerToThread(std::string ip, int port) const;
```

```
protected:
    ThreadMgr * _pThreadMgr;
    APP_TYPE _appType;
};
```

无论是登录服务进程还是游戏逻辑服务进程，都需要建立多个线程，启动所有线程，判断线程是否结束，结束所有线程，这些功能在基础的ServerApp类中实现，不用在每个服务进程中都实现一次。所以编写一个模板函数：

```
template<class APPClass>
inline int MainTemplate() {
    APPClass* pApp = new APPClass();
    pApp->InitApp();           // 初始化
    pApp->StartAllThread();    // 启动所有线程
    pApp->Run();               // 执行
    pApp->Dispose();           // 销毁
    delete pApp;
    return 0;
}
```

这个模板函数让整个流程看上去异常简洁，模板函数展开之后的功能如下：

- (1) 创建一个服务器对象。
- (2) 初始化App。
- (3) 启动所有线程。
- (4) 执行Run函数，程序会一直在这里循环，直到收到退出信号。
- (5) 退出后，执行Dispose函数处理关闭后事宜。

继承自ServerApp类的子类，除了InitApp初始化函数需要实现之外，其他功能（如启动线程、执行和销毁）都已经做完了，上层需要做的事情很少。以login进程为例，看看它是如何工作的。

关键点1: main函数

在login工程的main.cpp文件中实现了程序的main入口函数。

```
int main(int argc, char *argv[]) {
    return MainTemplate<LoginApp>();
}
```

只有一行代码，逻辑简单，调用了MainTemplate模板函数。在模板函数中生成了一个LoginApp类，然后初始化，启动它的所有线程，执行完成之后销毁。

关键点2: LoginApp类

在login_app.h文件中定义了一个新类LoginApp，该类继承自基类ServerApp，定义如下：

```
class LoginApp : public ServerApp {
public:
    explicit LoginApp() : ServerApp(APP_TYPE::APP_LOGIN) { }
    void InitApp() override;
};
```

毕竟需要在基本功能之上开发新的功能，对ServerApp进行继承是很有必要的，类LoginApp实现了虚函数InitApp。当LoginApp类被创建出来时，调用ServerApp的构造函数创建了3个线程。

```
ServerApp::ServerApp(APP_TYPE appType) {
    _appType = appType;
    Global::Instance(); // 数据通用类，单例，提供一些通
    _pThreadMgr = ThreadMgr::Instance();
    UpdateTime(); // 更新时间
    for (int i = 0; i < 3; i++) {
        _pThreadMgr->NewThread(); // 创建线程
    }
}
```

LoginApp实例创建完成之后，调用InitApp函数为自己进行初始化：

```
void LoginApp::InitApp() {
    AddListenerToThread("127.0.0.1", 2233);
}
```

因为本例的功能很简单，所以工程login在初始化时只是创建了一个监听。函数AddListenerToThread是由基类ServerApp提供的。创建函数在ServerApp类中，实现如下：

```
bool ServerApp::AddListenerToThread(const std::string ip, const int
    NetworkListen* pListener = new NetworkListen();
    if (!pListener->Listen(ip, port)) {
        delete pListener;
        return false;
    }
    _pThreadMgr->AddObjToThread(pListener);
    return true;
}
```

函数AddListenerToThread实现的功能就是创建一个NetworkListen监听类，调用AddObjToThread函数将该类放入线程中。在创建LoginApp类时，基类中创建了3个线程，NetworkListen会落入其中一个线程中。

关键点3：新的主循环

线程和数据都准备好之后，就进入了执行阶段。多线程主循环代码如下：

```
void ServerApp::Run() const{
    bool isRun = true;
    while (isRun) {
        UpdateTime();    // 更新当前帧的时间
        std::this_thread::sleep_for(std::chrono::milliseconds(10))
        isRun = _pThreadMgr->IsGameLoop();
    }
```

```
    }  
}
```

在这个主循环中，首先更新了时间。在游戏中，我们使用的时间单位是毫秒。每一帧会对当前时间进行更新。然后，让主线程沉睡10毫秒，以便让其他线程得到CPU的控制权。最后，通过线程管理类实例调用函数IsGameLoop，查看是否还有线程在工作，如果已经没有任何线程在工作，就退出进程。

进一步看看函数IsGameLoop都干了些什么：

```
bool ThreadMgr::IsGameLoop() {  
    for (auto iter = _threads.begin(); iter != _threads.end(); ++i)  
        if (iter->second->IsRun())  
            return true;  
}  
return false;  
}
```

只要有一个线程还在运行，就认为程序还没有结束。还记得我们之前讲的单线程游戏循环吗？

```
while(true) {  
    ProcessNetwork();  
    UpdateAllWorlds();  
}
```

读者可能觉得太奇怪了，为什么这里只返回一个bool值，游戏逻辑呢，更新操作去哪里了？ServerApp中的Run函数几乎什么也没有干。

在多线程中，主线程的作用已经发生了本质上的改变。主线程的主要任务是维护多线程，而真正更新的操作是在子线程中完成的。本代码中，启动一个线程时使用匿名函数的方式绑定了线程的执行函数，其代码如下：

```
void Thread::Start() {
```

```
        _isRun = true;
        _thread = std::thread([this]() {
            while (_isRun) {
                Update();
            }
        });
    }
}
```

我们需要从单线程的编码模式中跳出来，更新逻辑已经分散在各个线程中了。那么再来看下一个问题，在LoginApp初始化时，创建的NetworkListen类去了哪里？网络层又是如何更新的？

首先NetworkListen类被分配到了某个子线程上，NetworkListen实例也是线程上的一个包裹类。不论是Select还是Epoll模型，都是通过每帧调用Update来实现读写的。该实例进入了某个线程之后，其Update函数会在Thread线程类的Update中被调用，线程也不知道这个实例是NetworkListen类，它只知道这是一个基于ThreadObject类的实例，并不关心这个实例实现的是什么功能，只会一视同仁地执行Update。图3-8展示了整个框架的基本结构。

工程robots中的RobotApp与LoginApp的思路是一致的。在robots工程中，将10个Robot放到了3个线程中。每个Robot初始化时对login进程发起了连接请求。

库libserver中的ServerApp类实现了整个框架的基本流程，但它只是一个流程，不管逻辑。而上层的LoginApp或RobotApp为了实现自己的需求需要向底层增加自己需要的ThreadObject。LoginApp进程需要监听，增加了NetworkListen实例。而RobotApp为自己增加了Robot类，它并不需要监听网络，所以也就不需要添加NetworkListen类。

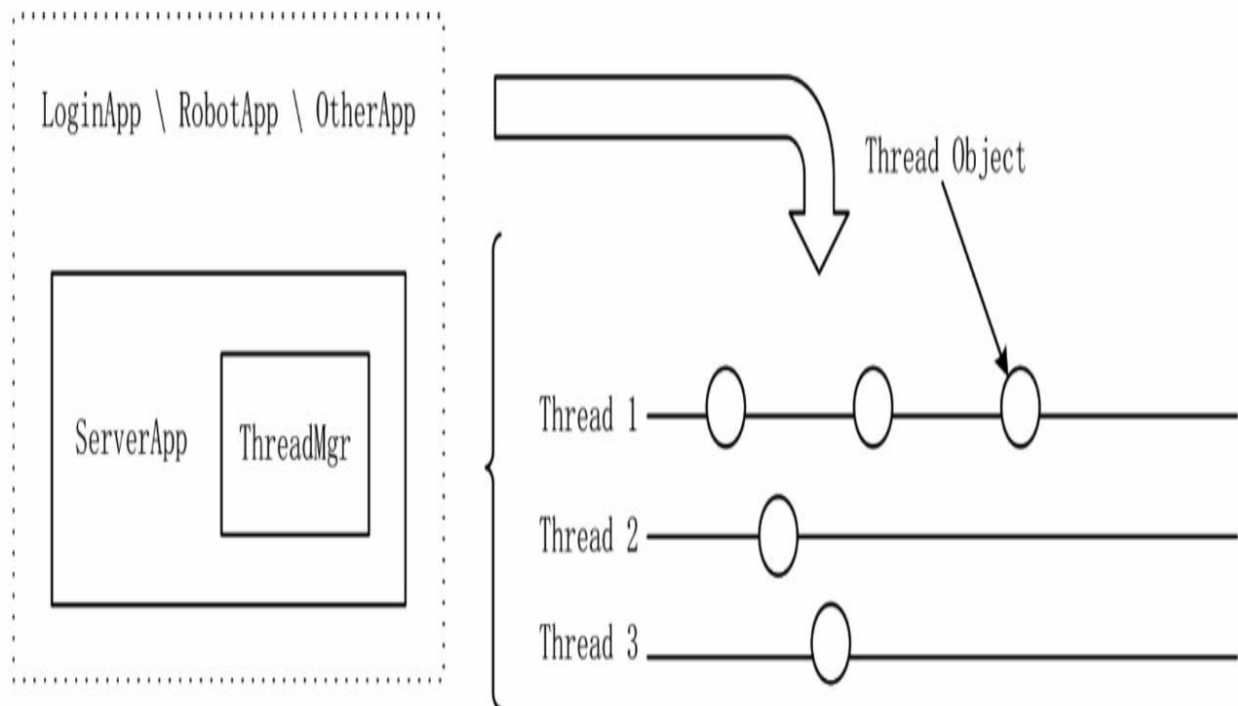


图3-8 线程与整体框架

理解这个流程之后，有一个大问题摆在我们面前，线程有了，线程上的Actor对象也有了，但是它们是如何互通消息的，下面继续来看看。

Actor对象之间的消息处理机制

本节的源代码位于03_03_engine_message目录下，其目的是向框架中加入基于Actor模型的消息队列。在第2章中，在协议体部分引入了protobuf协议。框架中所有消息由protobuf组成，所以有必要对消息定义做出规则说明。

3.8.1 消息定义原则

在使用protobuf时，定义一个消息有两个步骤：

- (1) 在proto文件中对协议内容进行定义。
- (2) 使用protoc.exe将定义生成C++或C#需要的文件。

在本书此后的所有例子中都将使用protobuf定义协议，所以有必要对它做出规范，以方便阅读与编码。

原则1：使用proto文件定义的enum为协议号

在使用协议号时，C++、C#代码中不需要分开定义协议号枚举，这个枚举在libserver库工程的proto_id.proto文件中，由protobuf同时生成两种语言的定义文件。

使用这一原则是为了统一客户端与服务端的代码。在后面的章节会引入一个客户端的架构，它基于Unity，采用C#脚本。如果我们在C++服务端，用C++语法定义协议枚举，在客户端再定义一个C#语法的协议枚举，一不小心就会出现协议编码不匹配的问题，所以将这个问题丢给protobuf，由它统一生成协议编号。

原则2：同一协议号不做两种用途

在编码中经常有这样一种现象，如果我们定义了协议号MI_AccountCheck用来验证账号，服务器收到了MI_AccountCheck协议，表示这是一个验证协议，处理完成之后返回MI_AccountCheck的协议给客户端，客户端默认收到MI_AccountCheck，就表示是MI_AccountCheck的返回。本书中不采用这种方法，如果遇到这种情况，那么返回协议编号可以定为MI_AccountCheckRs。二义性会给编码带来麻烦。无论是协议，还是在数据层中的类属性和函数都尽量不要有二义性。如果是一个用于判断的函数，就尽量让它只用于检查，不用于其他功能，将功能单一化。

原则3：协议编号与协议体同一命名

如果在msg.proto文件中定义了一个名为TestMsg的协议体，那么该协议的协议号枚举一定为MI_TestMsg。在msg.proto文件中，结构体定义如下：

```
message TestMsg {  
    string msg = 1;  
    int32 index = 2;  
}
```

在proto_id.proto文件中，协议枚举定义如下：

```
enum MsgId {  
    MI_None = 0; // proto3的枚举，第一个必为0  
    MI_TestMsg = 1;  
}
```

当框架功能越来越多时，如果不按这种一对一的命名原则，那么可能搞不清楚MI_TestMsg协议号到底对应哪个结构体，现在我们强制要求它们的名字一样，只要看到了协议号就知道协议体的名字，不需要记忆。

原则4：相同逻辑的字段，采用同一命名

在协议体的某些具体字段上采用同一命名。例如返回错误编码，统一命名为ReturnCode，以账号验证协议为例：

```
message AccountCheckRs {  
    enum ReturnCode {  
        ARC_OK = 0;  
        ARC_UNKONWN = 1;  
        ARC_NOT_FOUND_ACCOUNT = 2;  
        ...  
    }  
    string account = 1;
```

```
        ReturnCode return_code = 2;  
    }
```

该原则方便取读数据，统一整体风格。

3.8.2 消息队列机制

在讲解网络时演示过使用协议内容格式的代码，当时还只有单线程，代码简单。现在，线程之间需要互通消息，我们以消息队列的方式来实现消息的存储，即给ThreadObject加一个MessageList类，该类管理着当前ThreadObject需要处理的所有消息。

ThreadObject是一个Actor模型，MessageList类对应Actor的MailBox部分。我们来分析一下这部分代码，源代码位于03_03_engine_message目录。

关键点1: MessageList类

工程libserver下的message_list.h文件中有MessageList类的定义。

```
typedef std::function<void(Packet*)> HandleFunction;
class MessageList {
public:
    void RegisterFunction(int msgId, HandleFunction function); //
    bool IsFollowMsgId(int msgId);
    void ProcessPacket();
    void AddPacket(Packet* pPacket);
protected:
    std::mutex _msgMutex;
    std::list<Packet*> _msgList;
    std::map<int, HandleFunction> _callbackHandle; // 协
};
```

在这个类中，对4个函数的调用流程如下：

(1) 在MessageList类初始化时需要调用RegisterFunction函数，主动注册协议号，并为该协议号指定一个处理函数。

(2) 当有协议到达时，调用IsFollowMsgId函数判断一个协议号是否是自己关心的。如果是，就调用AddPacket函数将Packet类缓存，以便处理。

(3) 在帧函数中调用ProcessPacket来处理缓存的Packet协议。

每个线程包裹类ThreadObject继承自MessageList，也就是每个Actor模型类都有一个消息处理基类。

```
class ThreadObject : public IDisposable, public MessageList {  
    ...  
};
```

当ThreadObject收到消息之后会缓存到这个MessageList中，在适合的时机进行处理。

关键点2：消息分发

ThreadObject是如何收到消息的呢？这就需要实现消息分发。

再次回顾网络底层的流程，当某个网络Socket描述符上有读取事件发生的时候，Network基类调用了Socket对应的ConnectObj对象，让它执行了函数Recv进行读取操作。为了实现消息分发，我们在ConnectObj::Recv()函数中做了一点小小的修改。

```
bool ConnectObj::Recv() const {  
    bool isRs = false;  
    while (true) {  
        ... // 取出消息部分，有消息时isRs被赋值为true  
    }  
    if (isRs) {  
        while (true) {  
            const auto pPacket = _recvBuffer->GetPacket();  
            if (pPacket == nullptr)  
                break;  
            ThreadMgr::GetInstance()->AddPacket(pPacket);  
        }  
    }  
    return isRs;  
}
```

上面的代码中有两个while循环，一个是循环从系统网络缓存中取

出数据，另一个是循环从本地缓存中取出数据，一次到达的数据可能是多个，也可能一个也没有。成功收到消息之后，马上从缓冲区取出Packet，如果Packet不为空，就调用函数ThreadMgr::AddPacket进行广播。ThreadMgr的广播方式则是向自己管理下的所有线程发送Packet数据。

```
void ThreadMgr::AddPacket(Packet* pPacket) {
    std::lock_guard<std::mutex> guard(_thread_lock);
    for (auto iter = _threads.begin(); iter != _threads.end(); ++i)
        Thread* pThread = iter->second;
        pThread->AddPacket(pPacket);
    }
}
```

线程管理类ThreadMgr将Packet发送到了每一个线程Thread实例中。在线程中，遍历每一个包裹类ThreadObject，ThreadObject调用IsFollowMsgId来判断该Packet是否是是自己想关心、想要处理的协议。

```
void Thread::AddPacket(Packet* pPacket) {
    std::lock_guard<std::mutex> guard(_thread_lock);
    for (auto iter = _objlist.begin(); iter != _objlist.end(); ++i)
        ThreadObject* pObj = *iter;
        if (pObj->IsFollowMsgId(pPacket->GetMsgId())) {
            pObj->AddPacket(pPacket);
        }
    }
}
```

如果ThreadObject关心该协议，就将该协议传递到MessageList，MessageList收到协议之后，将协议加入待处理的队列中。

```
void MessageList::AddPacket(Packet* pPacket) {
    std::lock_guard<std::mutex> guard(_msgMutex);
    _msgList.push_back(pPacket);
}
```

总结一下消息分发的流程，参考图3-9。

(1) 当网络层收到网络数据时，将数据组织成Packet，此时二进制数据依然还是二进制，存放在Packet的缓冲中，通知ThreadMgr进行广播。

(2) ThreadMgr收到一个Packet，分发给所有线程。

(3) 线程分发给所有该线程之上的包裹类对象，如果包裹类对这个Packet有兴趣，就将这个Packet放到它的MessageList中，等待下一帧进行处理。

(4) 在下一帧中，MessageList发现有待处理的Packet，将其取出，调用该协议号注册的处理函数。在处理函数中，Packet中的二进制数据转化为protobuf的定义结构，变为有意义的数据。

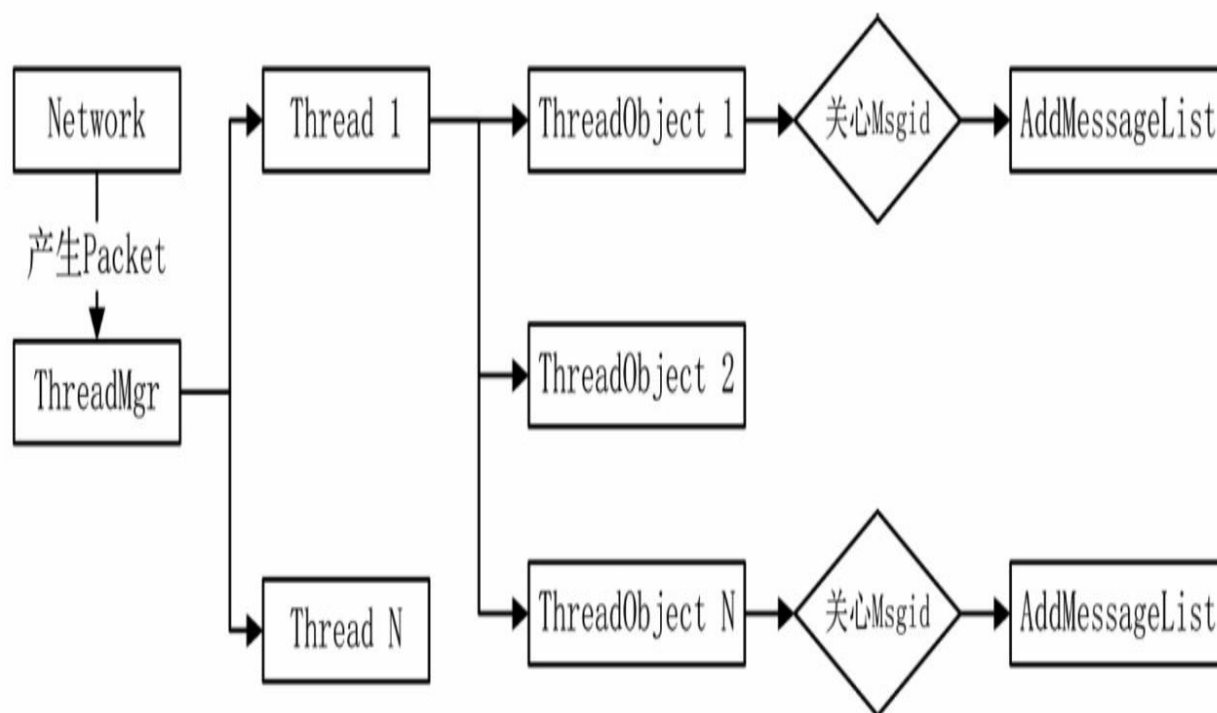


图3-9 消息队列

假设在这个过程中有多个实例对同一个协议号表示感兴趣，那么Packet会被增加到多个MessageList中。逻辑看上去并不难理解，关键

问题来了，如何判断当前这个Packet是ThreadObject所需要的呢？这就需要谈到另一个机制：消息过滤。

关键点3：消息过滤

在消息分发过程中，Packet类中的属性MsgId（也就是协议号）起作用了，回到类MessageList中看看，这个类除了维护消息队列之外，还有一个消息注册与处理机制。消息注册函数实现如下：

```
void MessageList::RegisterFuntion(int msgId, HandleFunction functi
    std::lock_guard<std::mutex> guard(_msgMutex);
    _callbackHandle[msgId] = function;
}
```

一个给定的协议号，对于每种类型的包裹类ThreadObject来说，其处理函数都不同，为了适应这种不同，包裹类ThreadObject为每个协议号指定一个处理函数。一旦这些处理函数注册成功，就决定了该类想要处理这个协议号，也就实现了消息过滤。

```
bool MessageList::IsFollowMsgId(int msgId) {
    std::lock_guard<std::mutex> guard(_msgMutex);
    return _callbackHandle.find(msgId) != _callbackHandle.end();
}
```

函数RegisterFuntion在ThreadObject生成时调用。当有协议通过底层传来之后，调用IsFollowMsgId预先判断有没有对应的处理函数，如果有，就表示“我非常关心这个数据”，需要留下来处理。

关键点4：消息处理

通过过滤消息，还留在MessageList中的Packet一定是ThreadObject类想要处理的消息。下面来看看存在队列中的消息是如何被处理掉的：

```
void MessageList::ProcessPacket() {
```

```

std::list<Packet*> tmpList;
_msgMutex.lock();
std::copy(_msgList.begin(), _msgList.end(), std::back_inserter
_msgList.clear());
_msgMutex.unlock();

for (auto packet : tmpList) {
    const auto handleIter = _callbackHandle.find(packet->GetMs
    if (handleIter == _callbackHandle.end()) {
        // 出错
    } else {
        handleIter->second(packet);
    }
}
tmpList.clear();
}

```

注意前几行代码，因为MessageList类是暴露在多线程下的，所以需要加锁。消息是由Network类发送过来的，这个类也是一个运行在线程上的包裹类，可能与当前对象在同一线程，也可能不同，所以这里必须要加锁。

上面的代码中使用了一个临时的List。如果函数在处理时加锁到函数的结束，那么整个消息回调函数都处于加锁状态，没有必要，加锁的目标只针对MessageList类中的消息队列。为了加速，消息队列复制到临时的List中，以提高效率。

之后的操作就很简单了，将协议号对应的回调函数取出来执行一下。

关键点5：验证数据

下面用Robot类来验证消息队列中消息发送与接收的功能是否能正常工作。在03_03_engine_message目录下的robots工程中，如果Robot类连接成功，就向服务器发送一条测试协议数据。其代码实现如下：

```

void Robot::Update() {
    NetworkConnector::Update();
    if (IsConnected() && !_isSendMsg) {
        Proto::TestMsg msg;
    }
}

```

```

        msg.set_msg("robot msg");
        auto pPacket= new Packet((int)Proto::MsgId::MI_TestMsg,_ma
        pPacket ->SerializeToBuffer(msg);
        SendPacket(pPacket);
        _isSendMsg = true;
    }
}

```

工程login收到这个数据会进行广播，发送到所有线程中的所有ThreadObject实例，但MI_TestMsg是我们新写的协议，并没有任何一个实例会去处理它。现在新建一个类TestMsgHandler专门来处理该协议，类定义如下：

```

class TestMsgHandler :public ThreadObject {
public:
    bool Init() override;
    void RegisterMsgFunction() override;
    void Update() override;
private:
    void HandleMsg(Packet* pPacket);
};

```

该类实现在ThreadObject类的虚函数。在初始化的时候调用RegisterMsgFunction实现了消息注册，实现如下：

```

void TestMsgHandler::RegisterMsgFunction() {
    RegisterFunction(Proto::MsgId::MI_TestMsg, BindFunPl(this, &Te
}
void TestMsgHandler::HandleMsg(Packet* pPacket) {
    auto protoObj = pPacket->ParseToProto<Proto::TestMsg>();
    std::cout << protoObj.msg().c_str() << std::endl;
}

```

处理这个协议的函数TestMsgHandler::HandleMsg，收到协议之后会将发送数据打印出来。编译运行本节的例子，可看到打印结果。

到目前为止，我们为游戏服务端的框架撰写了基本的线程和消息队列，组织了一个基本的游戏框架，在框架中，实现一个功能只需要非常简单的步骤。在处理消息时，和TestMsg消息的处理一样，在某个

ThreadObject类中注册一个处理函数就可以。

3.9

总结

本章我们正式进入了游戏编码，创建了游戏框架使用的线程，以Actor原则创建了线程中的对象ThreadObject类，有了这些类，已经有了一个简单框架的基础。

为了验证现有框架的开发能力，下一章将在这个简单的框架之上实现第一个功能——账号登录与验证功能。

第4章

账号登录与验证

登录是游戏开发中非常重要的一个功能，如果设计得不好，这里就可能成为进入游戏的第一个瓶颈。在整个登录过程中重要的一环是账号验证，但不要以为登录只是简单的账号验证，其中包括的细节与异步非常多。例如，如果同一个账号同时在两个Socket上发起登录协议该如何处理，如果有大于1000个玩家同时登录该怎么办，服务端是否能承受得住，是否需要排队登录，等等。本章先介绍一个简单的登录方案，等到我们足够熟悉登录流程之后，再来讨论复杂的情况。本章包括以下内容：

- 制作一个简单的账号验证接口。
- 账号登录验证。
- 创建机器人工程进行登录测试。

登录流程图

验证用户的协议顺序，如图4-1所示。

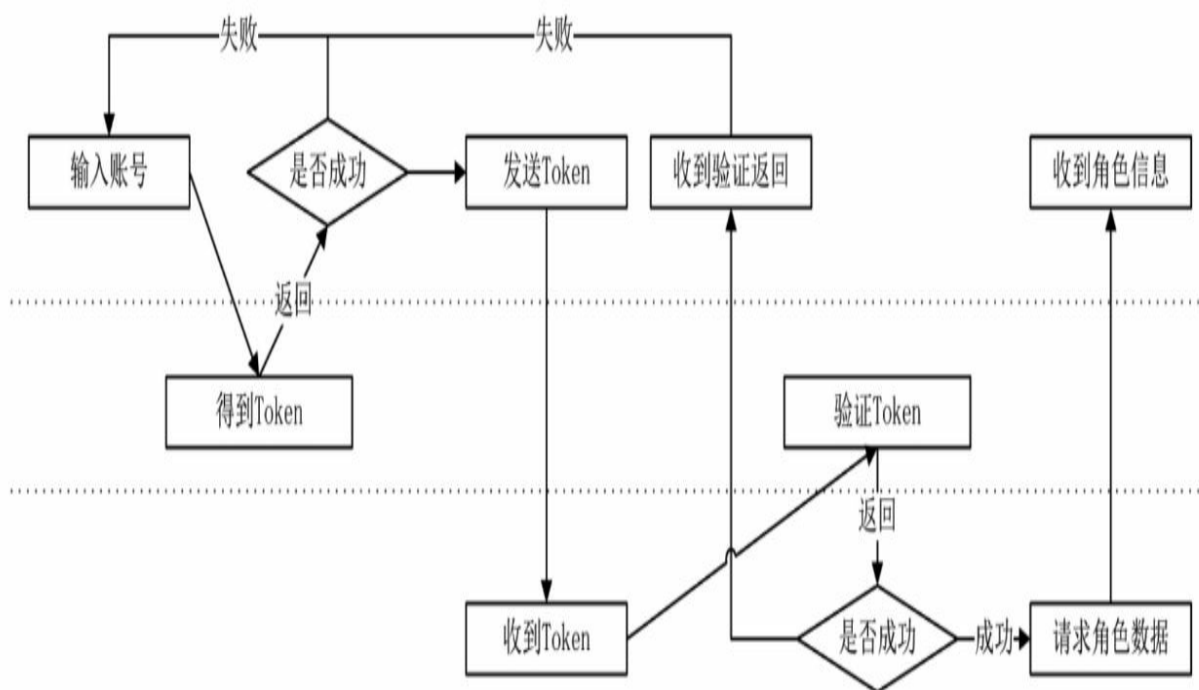


图4-1 登录流程

一般来说，游戏本身没有验证用户的接口，因为游戏最终是发布到各大平台的，验证用户账号是否合法需要调用平台的用户验证接口，也就是图4-1虚线标注的中间部分，这一部分的功能由平台提供。图4-1上半部分是由客户端发起的，中间是第三方平台，最下面是服务端的处理流程。总的来说，整个登录分成如下几步：

(1) 客户端通过平台给定的SDK取到该账号的token和账号ID，生成的token是有时效性的。

(2) 客户端将token发送到服务端，服务端通过平台提供的HTTP将token和账号ID传给平台进行验证。

(3) 服务端收到平台的反馈结果，如果失败就通知客户端，如果成功就加载玩家数据。

每个平台接入的API各有不同，但大部分采用了HTTP。为了搭建一个真正的环境，我们将实现一个HTTP账号验证系统，以模仿真实的登录环节。本节的工程源代码在04_01_engine_login目录中。

制作一个简单的验证接口

我们准备了两个PHP，分别实现注册和验证功能。为了能使用PHP这两个文件，真实模仿平台的用户验证，需要在Linux上安装一些环境。

如果读者是初次接触游戏编程，可能比较疑惑，如果我们拥有DB，为什么不能直接用数据库进行账号验证？

在实际的游戏发布中，很少会用到自己的账号进行登录，一般都是平台的账号。平台不允许在游戏中直接输入账号和密码进行验证，这样有密码泄露的风险。平台采用的验证解决方案是提供SDK包验证接口，返回token数据。一般来说，token是有时效性的，几秒几分不等。

在本节中，我们用PHP搭建一个接口环境，在服务端使用HTTP请求方式验证账号、密码与在平台上采用token的登录方式在原理、流程上是完全一致的。

在内部测试时可以用账号和密码进行登录，等到了外网，这个框架同样适合平台的token验证方式，不需要再写一套逻辑流程。我们不需要了解这两个PHP的语法，它们只是一个工具，用于简单验证与注册。为了使用PHP，在Linux上装Nginx服务和PHP环境。下面提供一个简易的安装流程。

(1) 执行`yum-y install epel-release`命令。

安装epel软件仓库，以方便从非CentOS的官方软件仓库安装一些软件。该指令会在`/etc/yum.repos.d/`目录下增加一个源。如果`yum.repos.d`目录下已经有了epel软件仓库，此步骤就可以省去。

(2) 执行`yum-y install nginx--enablerepo=epel`命令。

(3) 执行yum-y install php php-fpm命令。

上面两条命令安装Nginx和PHP。Nginx是一个轻量级、高性能的代理服务器，其特点是小巧，内存占用少。在安装Nginx的时候使用了参数“enablerepo=epel”，即需要使用epel软件仓库来进行安装。安装完成之后可以查看一下版本号：

```
[root@localhost ~]# php -v
[root@localhost ~]# nginx -v
```

(4) yum install-y php-mysql。

该指令安装PHP的MySQL扩展，如果没有这些库文件，PHP访问MySQL或者使用函数时就会出现错误。

4.2.1 Nginx参考配置

安装完Nginx之后需要对Nginx进行PHP相关的配置，其步骤如下：

步骤01 在/etc/nginx目录下新建一个php-fpm54.conf文件，指令如下：

```
[root@localhost nginx]# vim /etc/nginx/php-fpm54.conf
```

输入内容如下：

```
location ~ .*\.php$ {
    fastcgi_pass 127.0.0.1:9004;
    fastcgi_param  PHP_SELF      $uri;
    fastcgi_param  SERVER_NAME    $host;
    fastcgi_param  SCRIPT_FILENAME $document_root$fastcgi_script_n
    fastcgi_index  index.php;
    include fastcgi_params;
    include php_cgi.conf;
}
```

使用“rpm-qa php-fpm”来查看当前安装的php-fpm的版本号。也许你的虚拟机上安装的不是PHP 5.4版本，php-fpm54.conf可以取成你想要的任何名字。

步骤02 新建php_cgi.conf文件，指令如下：

```
[root@localhost nginx]# vim /etc/nginx/php_cgi.conf
```

在上一个php-fpm54.conf文件的最后一行引用了一个文件：php_cgi.conf。在php_cgi.conf中输入如下属性：

```
fastcgi_connect_timeout 300s;
```

```
fastcgi_send_timeout 300s;
fastcgi_read_timeout 300s;
fastcgi_buffer_size 128k;
fastcgi_buffers 8 128k;
fastcgi_busy_buffers_size 256k;
fastcgi_temp_file_write_size 256k;
fastcgi_intercept_errors on;
```

步骤03 配置Nginx文件，指令如下：

```
[root@localhost nginx]# vim /etc/nginx/conf.d/www.conf
```

输入如下内容：

```
server {
    server_name 192.168.0.120;
    index index.php;
    root /opt/www;
    include php-fpm54.conf;
}
```

在这个文件中，`server_name`就是主机名，需要改成当前机器的IP地址。`/opt/www`为放置PHP文件的目录，在这个配置中还引用了之前新建的`php-fpm54.conf`文件。

步骤04 进入/opt目录，新建www目录：

```
[root@localhost ~]# cd /opt/
[root@localhost opt]# mkdir www
```

4.2.2 php-fpm参考配置

下面接着进行php-fpm的配置。首先需要对旧配置文件进行备份，其目录在/etc/php-fpm.d。

步骤01 进入/etc/php-fpm.d目录，执行mv备份旧文件。

```
[root@localhost php-fpm.d]# ls // 查看目录下的文件
www.conf
[root@localhost php-fpm.d]# mv /etc/php-fpm.d/www.conf{,-bak}
[root@localhost php-fpm.d]# ls // 再次查看，已备份
www.conf-bak
```

步骤02 新建www.conf文件，指令如下：

```
[root@localhost php-fpm.d]# vim /etc/php-fpm.d/www.conf
```

输入如下内容：

```
[global]
pid = run/php-fpm.pid
error_log = log/error.log
[www]
listen = 127.0.0.1:9004
user = nobody
group = nobody
pm = static
pm.max_children = 5
rlimit_files = 20000
request_terminate_timeout = 10
pm.max_requests = 20000
```

注意，在PHP配置中设置了9004端口，需要对应Nginx中配置的fastcgi_pass端口。然后重新启动Nginx和PHP服务，指令

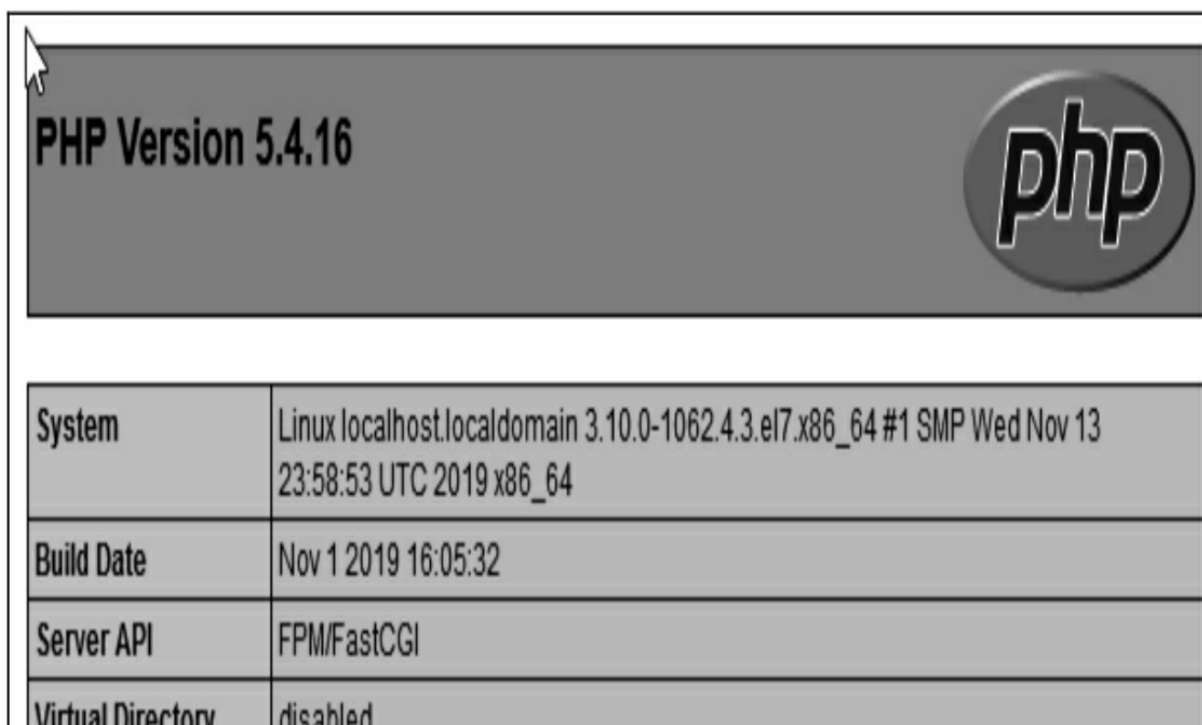
为“systemctl restart nginx php-fpm”。如果之前没有启动服务，可以使用“systemctl start nginx php-fpm”启动服务。最后使用下面的指令将这两个服务加入开机服务，以后一直会用到。

```
[root@localhost www]# systemctl enable nginx php-fpm
```

步骤03 新建测试文件。回到opt/www目录，编写一个info.php文件，其内容如下：

```
[root@C7-Pf www]# vim info.php
<?php
phpinfo();
?>
```

在浏览器中访问该PHP文件，配置正确会出现PHP状态，如图4-2所示。至此，部署完成。



System	Linux localhost.localdomain 3.10.0-1062.4.3.el7.x86_64 #1 SMP Wed Nov 13 23:58:53 UTC 2019 x86_64
Build Date	Nov 1 2019 16:05:32
Server API	FPM/FastCGI
Virtual Directory	disabled

图4-2 PHP信息界面

导入PHP登录接口

根据验证用户的流程图，在login工程中还需要一个PHP的API接口来验证我们的输入是否正确。PHP文件在本书源代码\tools\account目录下，进入该目录，使用cp指令将PHP文件放到我们配置的Nginx的默认目录/opt/www下。

```
[root@localhost account]# cp * /opt/www/
```

现在，在浏览器中访问“http://192.168.0.120/member_login_t.php”，可以看到一个纯JSON数据。使用curl指令，显示如下：

```
[root@localhost www]# curl http://192.168.0.120/member_login_t.php  
{ "returncode":0 }
```

4.3.1 修改PHP中的数据库配置

打开/opt/www目录下的mysql_config.php配置，需要修改DB信息：

```
$dbsource['account']['address'] = '127.0.0.1';  
$dbsource['account']['username'] = 'root';  
$dbsource['account']['password'] = '123456';
```

我们的例子中，因为Nginx和MySQL在同一台Linux上，所以MySQL的IP为127.0.0.1。如果你的密码与文件中有所不同，那么可以在这里修改。

4.3.2 导入测试账号

源代码目录\tools\account下有一个名为account.sql的文件，将account.sql导入数据库中，创建一个名为account的数据库，创建数据库时选择UTF8字符集和utf8_general_ci，表示数据库不区分字母大小写。导入account.sql之后，该数据库中创建了一个新的account表，并插入了一个现成的账号test，密码为123456。

配置完成之后，在浏览器上测试member_login.php接口，也可以使用curl指令：

```
[root@localhost www]# curl http://192.168.0.120/member_login.php  
{"returncode":2}
```

这个接口调用时连接了MySQL数据库，返回一个JSON数据，返回值有如下几个含义：

0=正确1=数据库操作错误
2=用户名错误
3=密码错误

上面用curl的访问中，因为没有输入账号，返回2，即用户名错误。至此，这个简单的PHP账号验证接口就做好了。

4.3.3 批量生成账号

数据库account下面还有一个存储过程，找到这个函数可以直接执行，会批量地生成1000个账号和密码，账号为test1-test1000，密码均为123456。完整过程代码如下：

```
BEGIN
DECLARE i INT DEFAULT 1;
while i <= 1000 do
insert into account (account, password) values (CONCAT('test',i),
set i = i +1;
end while;
END
```

执行过程即可生成测试账号。这1000账号是为了测试批量登录使用的。

编码中用到的第三方库

准备工作完成之后就可以开始编码了。在游戏的所有功能中，登录功能相对复杂，它需要一些第三方库来配合完成。为了完成验证账号的功能，本节需要用到两个第三方库：curl和jsoncpp。

- curl用于处理HTTP。

- jsoncpp处理返回的JSON数据。在某些平台，接口返回的数据也可能是XML格式的数据。这里仅给出JSON的例子，处理过程大同小异，如果是XML格式，就导入一个XML的分析库，其流程和JSON一致。我们将HTTP的分析部分交给了curl库来处理。现在，我们的框架基础还不够完善，在后面的章节中会用一个新的类来替代curl，不过，在本节的例子中，为了加快编码效率，暂时使用curl库。

4.4.1 库libcurl

curl的库名为libcurl，是一个开源的处理多种协议的包。下面分别来说明libcurl在两个系统下的编译和使用。在Linux下不需要编译，安装与配置步骤如下：

步骤01 执行安装命令“yum install-y libcurl-devel”。安装完成之后，在/usr/include目录下会多出一个curl目录。库文件位于/usr/lib64/libcurl.so。

步骤02 本节的源代码在04_01_engine_login目录中，修改login工程的CMakeLists.txt文件，修改“set(CMAKE_CXX_FLAGS“-Wall-std=c++14-pthread-lpthread-lcurl-DEPOLL”)”，将libcurl加入我们的工程中。

在Linux下安装的libcurl为7.29.0版本，安装时就能看到版本号。Yum仓库自带的版本选用的是稳定版本，一般不是新版本。在Windows端使用相同的版本号。如果读者想要自行编译，那么编译步骤如下：

步骤01 解压curl-7.29.0文件。在“开始”菜单的Visual Studio 2019菜单下，找到子菜单项x86 Native Tools Command Prompt for VS，以管理员身份运行。在命令行模式下进入刚才解压文件的目录，进入winbuild目录。编译一个可执行的最小版本，指令如下：nmake/f Makefile.vc mode=static ENABLE_IPV6=no ENABLE_SSPI=no ENABLE_IDN=no ENABLE_WINSSL=no MACHINE=x86。更多编译可参考winbuild目录下的BUILD.WINDOWS.txt文件。编译完成之后，builds目录下可以看到编译完成的静态库和include目录。

步骤02 将include目录复制到源代码目录的根目录include\windows之下，同时将lib目录中的库文件复制到本书源代码根目录libs\windows目录下。

步骤03 因为在login工程中会使用curl的静态库，所以修

改“login工程属性” → “C/C++” → “预处理器”选项，加入CURL_STATICLIB。

步骤04 最后，还需要将库文件libcurl_a.lib引入工程中。在“属性页面” → “连接器” → “输入”的“附加依赖项”中增加“libcurl_a.lib”。

4.4.2 库libjsoncpp

使用libcurl执行HTTP请求返回的数据是JSON格式的，所以需要有一个JSON的处理库。使用的第三方库是jsoncpp。下载地址：<https://github.com/open-source-parsers/jsoncpp>，下载jsoncpp-1.9.1.tar.gz版本。同样，该文件已放在了dependence目录下。

在Windows下已配置好所有工程，如果读者想要自行编译，那么编译步骤为：解压之后，使用CMake工具生成工程文件jsoncpp.sln，直接使用Visual Studio 2019工具编译生成jsoncpp.lib库文件。编译文件时需要注意以下几点：

- (1) 在生成之前，选择生成x86平台，与所有的库保持一致。
- (2) 打开jsoncpp_lib工程的属性页面，确认运行库为MDD，与之前的库保持一致。
- (3) 将jsoncpp-1.9.1\include目录复制到源代码根目录include\common\json下，方便以后所有工程使用。
- (4) 生成库文件jsoncpp.lib，复制到源代码根目录libs\windows下。
- (5) 为login工程增加依赖项jsoncpp.lib。

在Linux下的环境需要读者自行创建。在Linux下编译jsoncpp很简单，进入dependence目录，对jsoncpp进行解压，命令如下：

```
[root@localhost dependence]# tar xf jsoncpp-1.9.1
[root@localhost dependence]# cd jsoncpp-1.9.1/
```

原CMakeLists.txt文件为了兼容，过于复杂，在gcc1.9.0下编译时会出错。本书提供了一个简单的jsoncpp-CMakeLists.txt文件，该文件位于源代码库的dependence目录下，替换原CMakeLists.txt文件即可编译。

```
[root@localhost jsoncpp-1.9.1]# cp ../jsoncpp-CMakeLists.txt ./CMakeLists.txt
cp: overwrite './CMakeLists.txt'? y
[root@localhost jsoncpp-1.9.1]# cmake3 ./
[root@localhost jsoncpp-1.9.1]# make
```

执行make之后，会在当前目录生成libjsoncpp.so文件，将文件复制到根目录libs/linux下。这里没有使用make install将它安装在系统目录中，因为系统中可能已经有一个旧版本，为了不引起冲突，选择在本地保存一个新版本。Yum仓库中也有一个版本的jsoncpp，只是版本比较低，没有选用这个低版本。安装完成后需要配置使用，修改login工程中的CMakeLists.txt文件：

(1) 修改CMAKE_CXX_FLAGS，增加-ljsoncpp描述，代码如下：

```
set(CMAKE_CXX_FLAGS "-Wall -std=c++14 -pthread -lprotobuf -lcurl -ljsoncpp")
```

(2) 增加一个连接目录，指向libjsoncpp.so，统一放到根目录的libs目录下。

```
link_directories(${CMAKE_BINARY_DIR}/../libs/linux)
```

(3) 修改环境变量。在~/.bash_profile文件中增加库的自定义目录。修改属性LD_LIBRARY_PATH，加入so文件的目录位置/root/game/libs/linux，注意Linux下的分隔符为冒号。

```
[root@localhost ~]# vim ~/.bash_profile
...
LD_LIBRARY_PATH=/root/game/libs/linux:/usr/local/gcc-9.1.0/lib64/:
export LD_LIBRARY_PATH
```

上面的配置中使用的是动态库，执行时会寻找相关的so文件。除了动态库外，还可以使用静态库。静态库与动态库不同，它是直接将静态文件.a编译到生成文件中。若要生成静态连接，则可以修改login工程下的CMakeLists.txt文件。

(1) 在最后增加一句 “target_link_libraries (\${MyProjectName} libjsoncpp_static.a”

(2) 在属性CMAKE_CXX_FLAGS中去除-ljsoncpp, 设为 “set (CMAKE_CXX_FLAGS“-Wall-std=c++14-pthread-lprotobuf-lcurl-DEPOLL”)”。

在动态库中，login进程生成的debug文件大小为5MB左右，在静态库中大于这个值。动态库和静态库各有优劣。库jsoncpp的大小比较小，只有300KB左右。但对于protobuf这样的库来说，大小约为30MB，这种情况下，如果还使用静态库的话，生成的login文件就会大于30MB。

账号验证代码分析

现在准备工作已经完成，可以开始验证账号了。本节完整的工程代码在04_01_engine_login目录中，按照之前的框架设计，验证账号这个功能，只需要做一个“线程包裹”类即可。在这个类中，处理关于账号验证相关的协议。下面来分析login进程是如何实现账号验证请求的，先从协议号开始讲起。

4.5.1 定义登录协议号

既然是新的功能，必然有新的协议号产生。在工程libserver下的proto_id.proto文件中定义了验证账号需要的协议号，有3个，分别为：

```
C2L_AccountCheck          = 1001;  // 验证账号
C2L_AccountCheckRs        = 1002;  // 验证返回结果
MI_AccountCheckToHttpRs   = 1005;  // HTTP请求的返回协议
```

一个完整游戏的协议是非常多的，为了便于识别协议，在协议号的前面加了C2L这样的标识，表示从客户端到login进程（client to login的缩写）。C2L_AccountCheck表示客户端向login进程请求验证一个账号，而C2L_AccountCheckRs表示这个协议的返回协议。返回协议是从login发送到客户端的，但是为了统一，这个返回协议前缀还是使用C2L，但多了一个Rs=Result表示结果。

协议MI_AccountCheckToHttpRs有一个Rs，是一个结果协议，这是login向Nginx请求HTTP验证的结果。当服务端收到一个账号验证时，服务端没有能力对这个请求做出判断，需要向第三方请求，协议MI_AccountCheckToHttpRs就是这个请求结果的返回。

当然，这种协议号规定每一个人每个框架都不一样，这只是本书采用的约定规则。具体看看这3个协议的定义：

（1）协议号C2L_AccountCheck对应Proto：：AccountCheck类，客户端发起请求时，输入一个账号与密码。这个协议的协议体如下：

```
message AccountCheck {
    string account = 1;
    string password = 2;
}
```

当然，如果在其他的平台，协议中就可能带上token字段。

(2) 协议号MI_AccountCheckToHttpRs对应Proto: : AccountCheckToHttpRs类。当服务端收到客户端请求验证时会发起一个HTTP请求，为了处理HTTP请求创建了一个HttpRequest类专门来处理，该类也是基于线程包裹类的，它和Account类是独立的，不互通数据，也不会造成线程阻塞。当HTTP请求有了结果之后，要发起一个新的协议，将结束传回Account类中。其协议结构如下：

```
message AccountCheckToHttpRs {
    int32 return_code = 1;    // AccountCheckRs::ReturnCode
    string account = 2;
}
```

(3) 协议号C2L_AccountCheckRs对应Proto: : AccountCheckRs类。在服务端，只有收到MI_AccountCheckToHttpRs之后才认为一个验证结束了，这时会发送C2L_AccountCheckRs返回给客户端，其协议体如下：

```
message AccountCheckRs {
    enum ReturnCode {
        ARC_OK = 0;
        ARC_UNKONWN = 1;
        ARC_NOT_FOUND_ACCOUNT = 2;
        ARC_PASSWORD_WRONG = 3;
        ...
    }
    int32 return_code = 1;
}
```

在返回协议结构中定义了一个枚举，用来标识返回值的类型，以方便客户端界面显示文字。

4.5.2 处理协议的Account类

定义完协议之后，服务端需要一个新类来处理这些协议。打开login工程，其中有一个新类Account来处理登录数据。这个类是一个线程包裹类。其定义如下：

```
class Account :public ThreadObject {
public:
    bool Init() override;
    void RegisterMsgFunction() override;
    void Update() override;
private:
    void HandleAccountCheck(Packet* pPacket);
    void HandleAccountCheckToHttpRs(Packet* pPacket);
private:
    PlayerMgr _playerMgr;
};
```

Account类是login进程特有的类。Account类的主要目的是处理客户端发送来的账号验证请求。在Account类中维护了PlayerMgr实例，该实例的作用是维护所有Account类中正在登录的账号对象。

再次强调，基类ThreadObject是运行在线程之上的对象，如果把线程比喻成一条流水线，那么ThreadObject类就像运行在其上的包裹。ThreadObject类必须实现3个虚函数：Init、RegisterMsgFunction和Update，分别用于初始化、注册自己感兴趣的协议以及执行更新操作。在Account类中也实现了这3个虚函数。

4.5.3 Account类如何放置到线程中

Account类整理完成之后，就可以将它丢到线程中运行了。这段功能的代码在login工程的login_app.cpp文件中，初始化进程时，代码如下：

```
void LoginApp::InitApp() {
    AddListenerToThread("127.0.0.1", 2233);
    ...
    Account* pAccount = new Account();
    _pThreadMgr->AddObjToThread(pAccount);
}
```

在login工程中，除了创建网络监听类外，还创建了一个Account实例，调用线程管理类的AddObjToThread函数，将它丢到了某个线程中。在进入线程之前，会首先调用Account类的Init和RegisterMsgFunction两个函数。而在函数RegisterMsgFunction中，为类Account相关的两个协议绑定了处理函数：

```
void Account::RegisterMsgFunction() {
    RegisterFuntion(Proto::MsgId::C2L_AccountCheck, BindFunPl(this
    RegisterFuntion(Proto::MsgId::MI_AccountCheckToHttpRs, BindFun
}
```

上面的代码表示，Account关心C2L_AccountCheck和MI_AccountCheckToHttpRs两个消息，如果收到C2L_AccountCheck消息，就调用Account::HandleAccountCheck函数来处理；如果收到MI_AccountCheckToHttpRs协议，就调用Account::HandleAccountCheckToHttpRs来处理。详细来看HandleAccountCheck函数的实现，该函数处理登录协议：

```
void Account::HandleAccountCheck(Packet* pPacket) {
    // 从网络层传来的Packet，解析出我们想要的结构
    auto protoCheck = pPacket->ParseToProto<Proto::AccountCheck>()
    const auto socket = pPacket->GetSocket();
    auto pPlayer = _playerMgr.GetPlayer(protoCheck.account());
```

```

if (pPlayer != nullptr) {
    // 如果有相同账号正在登录，就返回客户端消息，同时关闭网络
    Proto::AccountCheckRs protoResult;
    protoResult.set_return_code(Proto::AccountCheckRs::ARC_LOG
    auto pRsPacket = new Packet(Proto::MsgId::C2L_AccountCheck
    pRsPacket->SerializeToBuffer(protoResult);
    SendPacket(pRsPacket);
    // 关闭网络
    const auto pPacketDis = new Packet(Proto::MsgId:: MI_Netwo
    DispatchPacket(pPacketDis);
    return;
}

// 更新信息
_playerMgr.AddPlayer(socket, protoCheck.account(), protoCheck.
// 验证账号(http)
HttpRequestAccount* pHttp = new HttpRequestAccount(protoCheck.
ThreadMgr::GetInstance()->AddObjToThread(pHttp);
}

```

如果同一个账号发起两次验证信息，就会关闭网络。关闭网络时调用了DispatchPacket函数，向整个框架所有线程广播MI_NetworkDisconnectToNet协议。其原理与之前的ThreadMgr::AddPacket一致，这里不再详细讨论了。

关注Account::HandleAccountCheck函数最后两行代码。新建了一个HttpRequestAccount类，将验证的功能转交给它，并将它加入了线程中。该类的主要作用是发起HTTP请求。根据返回结果才能判断这个账号与密码是正确或者错误的。

4.5.4 处理验证的HttpRequestAccount类

为了完成异步的账号验证，新建了一个处理HTTP的线程包裹类HttpRequest。图4-3展示了Account类与HttpRequest类的关系。

整个验证流程有不确定性，账号A的验证请求可能先到Account类中，但其返回值可能晚于账号B。这与它的处理类HttpRequest的返回时机有关系，每个账号都有一个HttpRequest实体来处理它的HTTP请求。Account类和这些HttpRequest类可能在一个线程，也可能不在。这个过程不是一个串行处理，所以结果的顺序是不能保证的。正因为它不是串行处理，不存在阻塞问题，所以效率得到了保证。

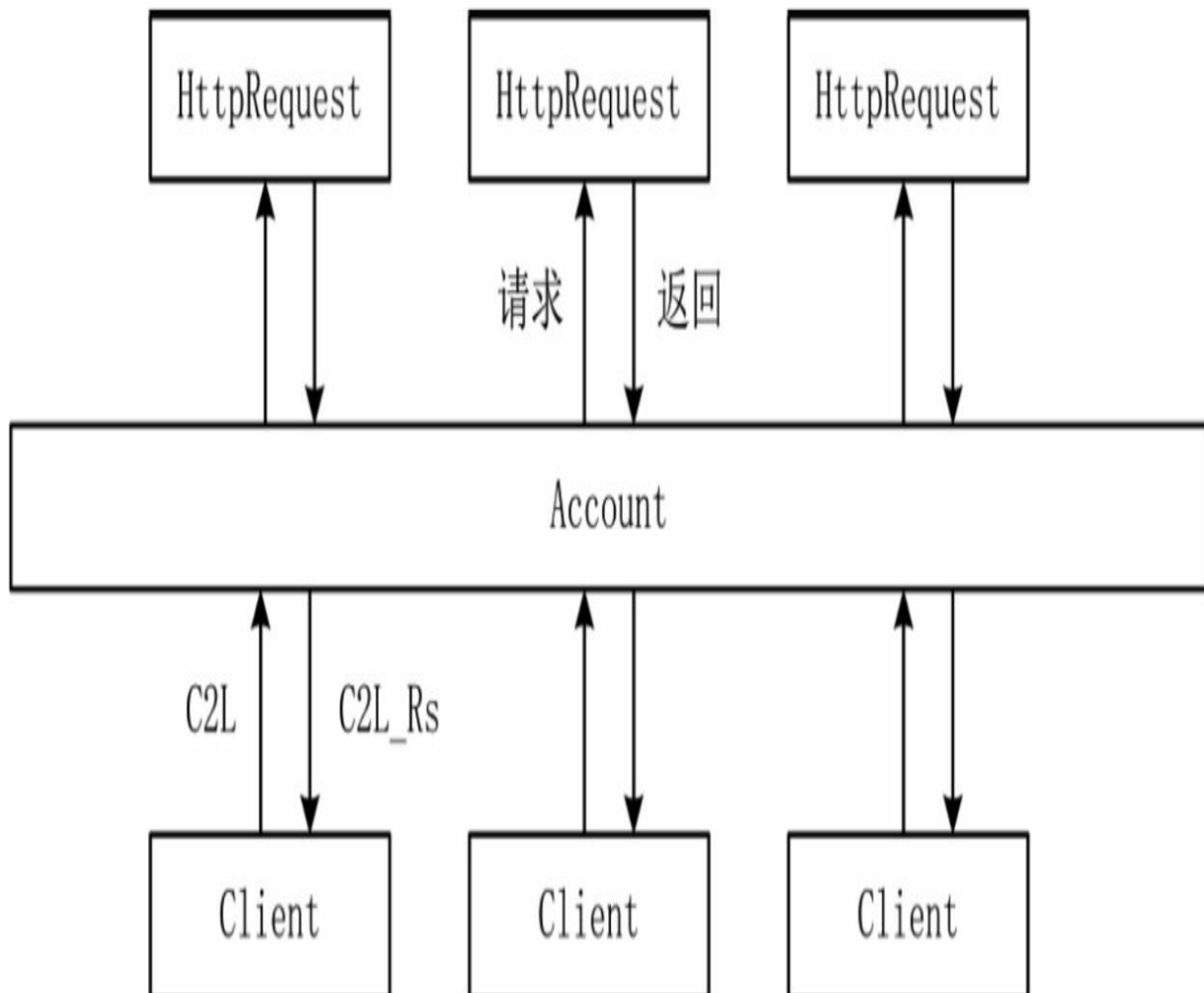


图4-3 Account类与HttpRequest类的关系图

那么，HttpRequest是如何无阻塞处理HTTP请求的呢？当我们请求一个HTTP数据时，从请求到返回数据分成了几个阶段：

```
enum HttpRequestState {
    HRS_Send,           // 发送数据
    HRS_Process,        // 等待数据
    HRS_Over,           // 完成
    HRS_NoActive,       // 完成后的非激活状态，等待线程删除
    HRS_Timeout,        // 请求超时
};
```

每个状态阶段都是异步的，即不堵塞程序的执行。在HRS_Process状态时会收到HTTP请求的返回数据，这时会向Account线程包裹类发送结果，然后HttpRequest这个实例将进入HRS_Over状态，随后销毁。HttpRequest类同样继承了ThreadObject基类，所以它需要实现基类的虚函数。实际上，HttpRequest类不会关心任何协议，因为它是一个功能处理函数。它真正的重要功能在Update函数中。在这个函数中的每一帧都在检查自己的状态，一旦收到数据，就要及时更新自己的状态，图4-4展示了这几种状态的前后关系，代码如下：

```
void HttpRequest::Update() {
    switch (State) {
        case HRS_Send: {
            if (ProcessSend())
                State = HRS_Process;
        }
        break;
        case HRS_Process: {
            if (Process())
                State = HRS_Over;
        }
        break;
        case HRS_Over: {
            ProcessOver();
            State = HRS_NoActive;
            _active = false;
        }
        break;
    }
```

}
...

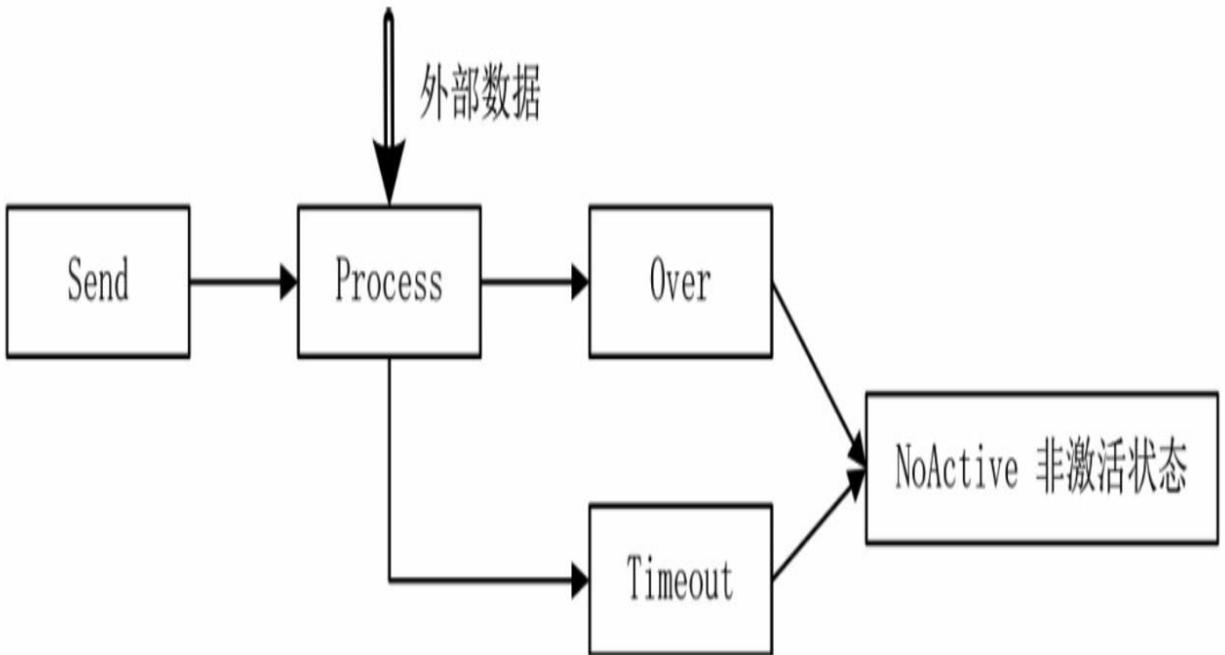


图4-4 HttpRequest类状态

返回数据的处理在函数Process中，采用的是libcurl库，这里为了节省篇幅，将不再深入libcurl库调用的细节。如果读者感兴趣，那么可以研究一下HttpRequest::Process的处理。有一条需要特别注意，在调用libcurl库时要保证每一帧都不会阻塞。

我们将HTTP请求以及返回的数据处理都放到了HttpRequest类中，让它作为基类。在此基础之上继承了一个子类HttpRequestAccount来实现登录测试的PHP文件返回结果的分析。类定义如下：

```
class HttpRequestAccount : public HttpRequest {
public:
    HttpRequestAccount(std::string account, std::string password);
protected:
    void ProcessMsg(Json::Value value) override;
};
```

该类相当简单，其处理细节隐藏到了基类中，只需要处理两个方面：

(1) 初始化，保证发送的链接指向我们测试的Nginx地址：

```
HttpRequestAccount::HttpRequestAccount(std::string account, std::s
    _password = password;
    _curlRs = CRS_None;
    _method = HttpRequestMethod::HRM_Post;
    _url = "192.168.0.120/member_login_t.php";
    _params.append("account=").append(_account);
    _params.append("&password=").append(_password);
}
```

在类的构造函数中进行了初始化的属性赋值，重要的3个属性是method、url和params。之前在PHP中的接口现在派上用场了，将地址赋值给变量url，参数赋值给变量params，参数的组织方式和在浏览器上使用的方式一致，method指明了想要用Post方式或Get方式提交这些数据。在测试时，url中的IP地址需要指向之前安装的Nginx服务器的网络地址。

(2) 返回的JSON数据。基类从返回值中成功得到数据之后会调用ProcessMsg虚函数，该虚函数的参数是一个Json::Value类型，我们从中取出想要的数，组装一个新的协议MI_AccountCheckToHttpRequest广播到框架中，最后Account类会处理该协议。ProcessMsg虚函数实现如下：

```
void HttpRequestAccount::ProcessMsg(Json::Value value) {
    auto code = Proto::AccountCheckRs::ARC_UNKONWN;
    int httpcode = value["returncode"].asInt();
    if (httpcode == 0)
        code = Proto::AccountCheckRs::ARC_OK;
    else if (httpcode == 2)
        code = Proto::AccountCheckRs::ARC_NOT_FOUND_ACCOUNT;
    else if (httpcode == 3)
        code = Proto::AccountCheckRs::ARC_PASSWORD_WRONG;

    Proto::AccountCheckToHttpRequest checkProto;
    checkProto.set_account(_account);
}
```

```
checkProto.set_return_code(code);

auto pCheckPacket = new Packet(Proto::MsgId::MI_AccountCheckTo
pCheckPacket->SerializeToBuffer(checkProto);
DispatchPacket(pCheckPacket);
}
```

类HttpRequestAccount只是内网的一个方便登录测试的接口，在实际操作中，例如要登录一些大平台，平台会提供一系列的HTTP接口，可以重新再编写一个处理类，在新类的构造函数中指定好其url和参数，一般来说参数都带token，向平台请求验证token是否正确，不论是正确还是错误都会返回一个特定格式的JSON数据。

再从整体上来看看login进程，现在Account和HttpRequest都放到了线程中。它们的耦合性很低，假设有一天不再需要login进程，需要直接在game进程上进行账号验证，只需要把Account和HttpRequest这两个包裹类文件加入game工程中，不需要修改任何代码就可以完成整个迁移，听上去似乎很简单，但事实就是如此。在后面的章节中会讲解这一部分内容。

在这个例子中，线程包裹类的低耦合性充分体现出来了，组织和拆分功能都非常方便，只需要把相关的线程包裹类加入新进程的编译中就可以了。

一个独立的包裹类的信息是闭塞的，它只处理从消息队列收到的协议，其结果再通过协议发送出去。发送协议的类并不知道接收协议的类如何处理，也不关心谁去处理。这是低耦合产生的根本原因。Account类并不知道是谁在处理HTTP请求，也不知道谁会将返回结果给它。HttpRequest类处理完数据之后不知道这些数据谁会需要，只要发送出去就好了。

结果测试

现在来看看工程的执行结果。在测试之前需要根据当前环境修改代码，因为验证账户的HTTP请求地址写死在代码中了，需要在HttpRequestAccount：：HttpRequestAccount函数中根据当前环境进行修改。

打开两个终端，进入目录04_01_engine_login，编译之后启动login进程，再启动robots进程。在robots控制台输入“login-help”。命令login有两个子命令：一个是“-a”，表示单人登录；另一个“-ex”，表示批量登录，格式如下：

```
login -a test      // 用test账号登录
login -ex test 2   // 用test为前缀的账号登录，登录数量为2
```

现在，我们输入指令“login-ex test 2”，批量登录，得到的打印信息如下：

```
[root@localhost bin]# ./robotsd
login -ex test 2
account check result account:test1 code:0
account check result account:test2 code:0
account check result account:test2 code:0
account check result account:test1 code:0
```

我们登录了两个账号，却得到了4个打印结果，是怎么回事呢？查看Robot类中C2L_AccountCheckRs协议的处理函数HandleAccountCheckRs，其实现代码如下：

```
void Robot::HandleAccountCheckRs(Robot* pRobot, Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::AccountCheckRs>();
    std::cout << "account check result account:" << _account << "
    if (proto.return_code() == Proto::AccountCheckRs::ARC_OK)
        ChangeState(RobotStateType::RobotState_Login_LoggedIn);
}
```

收到登录返回协议时，就会打印“account check result account: test code: ...”。

认真观察，发现每个账号都处理了两次，这是因为在线程中有两个Robot实例存在，而每个Robot都关心C2L_AccountCheckRs协议。当C2L_AccountCheckRs协议到来时，被无差别地发送到了每个线程，导致每个线程中每个关心该协议的Robot都处理了协议。

这就好像有两张地图，每张地图上都有无数个玩家。地图是线程包裹类，玩家是地图类中的数据。如果玩家发出一个move指令，两个地图都对move指令感兴趣，但是发出指令的玩家只可能在其中一个地图上，要如何处理呢？下面我们来解决这个问题。某个特定的Socket发出的协议只能由某个特定的类来处理。

消息过滤机制

本节着手来解决4.6节遗留下的问题：一个Packet协议被发送到了多个Robot实例上。要解决这个问题，首先需要扩展消息机制。本节的参考代码在目录04_02_engine_robots中。前面讲过，消息处理机制在MessageList类中，每个对象都继承自MessageList类，在该类中注册了自己关心的协议。现在需要对它进行修改，新的定义如下：

```
class MessageList :public IDisposable {
public:
void MessageList::AttachCallBackHandler(MessageCallBackFunctionInf
    _pCallBackFuns = pCallback;
}
protected:
    MessageCallBackFunctionInfo* _pCallBackFuns{ nullptr };
};
```

为MessageList增加一个结构，新的结构用来容纳之前MessageList中的函数与数据，类定义如下：

```
class MessageCallBackFunctionInfo {
public:
    virtual bool IsFollowMsgId(Packet* packet) = 0;
    virtual void ProcessPacket() = 0;
    void AddPacket(Packet* pPacket);
protected:
    std::mutex _msgMutex;
    std::list<Packet*> _msgList;
};
```

从上面的代码可以看出，MessageCallbackFunctionInfo是一个基类，继承自它的子类需要实现两个虚函数IsFollowMsgId和ProcessPacket。在本例中有两个继承类，分别是MessageCallbackFunction和MessageCallbackFunctionFilterObj。两个类定义如下：

```
class MessageCallbackFunction :public MessageCallbackFunctionInfo
public:
    using HandleFunction = std::function<void(Packet*)>;
    ...
protected:
    std::map<int, HandleFunction> _callbackHandle;
};
template<class T>
class MessageCallbackFunctionFilterObj :public MessageCallbackFunc
public:
    using HandleFunctionWithObj = std::function<void(T*, Packet*)>
    using HandleGetObject = std::function<T*(SOCKET)>;
    ...
    bool IsFollowMsgId(Packet* packet) override;
    HandleGetObject GetPacketObject{ nullptr };
private:
    std::map<int, HandleFunctionWithObj> _callbackHandleWithObj;
};
```

MessageCallbackFunction类是4.6节的例子中无差别接收协议的机制，而MessageCallbackFunctionFilterObj类是有条件地进行筛选。那么它是如何进行筛选的呢？主要改变的是新定义的函数HandleGetObject和函数IsFollowMsgId。以Robot类为例来看新类是如何实现消息过滤的。

```
void Robot::RegisterMsgFunction() {
    auto pMsgCallBack = new MessageCallbackFunctionFilterObj<Robot>
    pMsgCallBack->GetPacketObject = [this](SOCKET socket)->Robot*
        if (this->GetSocket() == socket)
            return this;
        return nullptr;
};
AttachCallBackHandler(pMsgCallBack);
pMsgCallBack->RegisterFunctionWithObj(Proto::MsgId::C2L_Account
}
```

在旧代码中对协议进行了无差别的注册。而新代码中，调用函数 `AttachCallbackHandler` 绑定一个自己需要的消息回调类的实例，`MessageList` 维护了一个 `MessageCallbackFunctionInfo` 指针，这个指针可能是一个无差别处理的 `MessageCallbackFunction` 类，也可以如 `Robot` 类一样是 `MessageCallbackFunctionFilterObj` 类。在 `MessageCallbackFunctionFilterObj<Robot>` 类中用匿名函数实现了 `GetPacketObject`，如果 `Packet` 中的 `Socket` 值和 `Robot` 中的 `Socket` 值一致，就认为是 `Packet` 的目的对象。当一个 `Packet` 到来时，会调用 `IsFollowMsgId` 函数来检查。其实现如下：

```
template <class T>
bool MessageCallbackFunctionFilterObj<T>::IsFollowMsgId(Packet* pa
    if (MessageCallbackFunction::IsFollowMsgId(packet))
        return true;
    if (_callbackHandleWithObj.find(packet->GetMsgId()) != _callba
        if (GetPacketObject != nullptr) {
            T* pObj = GetPacketObject(packet->GetSocket());
            if (pObj != nullptr)
                return true;
        }
    }
    return false;
}
```

如果是一个需要筛选的协议，就调用筛选函数。若能取到对象，则说明这个 `Packet` 通过了检查，正是当前对象需要的协议。1000个 `Robot` 登录，同时到来1000条 `C2L_AccountCheckRs` 协议，这时经过 `GetPacketObject` 函数之后，只会有一个对象与之对应，这就达到了筛选的目的。因为 `Socket` 永远是唯一的。现在来测试一下本例的结果。启动 `login`，然后启动 `robots`：

```
[root@localhost bin]# ./robotd
login -ex test 2 // 输入
account check result account:test2 code:0
account check result account:test1 code:0
[root@localhost bin]# ./logind
epoll model
test begin
Login-Connectting over. time:7.2e-05s
```

```
Login-Connectted over. time:0.000371s  
Login-Logined over. time:0.208033s
```

以两个账号同时登录测试，从robots进程的打印信息可以看出，Bug已经被我们修复了。

测试机器人

在04_02_engine_robots工程中写了一个robots工程。一般来说，将功能正式交付客户使用之前，需要使用robots进程对其进行测试。为什么我们需要先行测试？

大部分情况下，公司服务端和客户端是由两个人来完成的。作为后台开发，如果不进行自测，就交付给客户端的同事，这样会极大地增加双方的交互成本。robots的自测功能可以很好地避免这些烦人的情况出现，在4.7节的例子中，我们就从Robot类中发现了数据接收的异常。robots进程除了自测功能之外，还有一个很重要的使命，就是进行压力测试，找到影响效率的瓶颈所在。前面我们创建过测试进程，其作用是创建多个线程对网络的传输做出测试。

在后面的章节测试中会多次反复使用到robots，为了便于理解，在本节中对它的结构进行讲解。有以下几点需要注意：

（1）进程中可以有无数个Robot类实例。每个Robot类实例都是一个网络连接类，数据相互独立。这里体现了包裹类并发的好处，如果没有并发处理，串行操作就需要执行无数次更新之后才轮到自己，效率不会太理想。

（2）RobotMgr类对Robot类进行管理。虽然称为管理类，但是该类对Robot类只有一个统计功能。如果RobotMgr类要与Robot类进行通信，就必须通过协议。因为RobotMgr类同样也是基于ThreadObject的类。

4.8.1 状态机

在robots工程中，需要介绍一个在游戏制作中很重要的数据结构——状态机。在前面的代码中也处理了状态数据，HttpRequest类中定义了几个状态枚举，而在使用这些状态枚举时采用了switch的处理方式，其代码如下：

```
void HttpRequest::Update() {
    switch (State) {
        case HRS_Send: ... break;
        case HRS_Process: ... break;
    }
}
```

在游戏中，操作一个Boss，它拥有一些简单的AI，这些简单的AI可能是由状态机来实现的。举例说明，在《魔兽世界》中，Boss的AI就比较复杂，AI有层次之分，在某些条件下，一个层次可以打开另一个层次的AI，每一个层次的AI调用技能都不相同，Boss可以做出不同的攻击和回血操作，或者在需要时可以改变自己的行走速度。

层次越多，考虑的外部影响越多，Boss就越灵活，这使得Boss看上去极具智能。这些层次关系也称为状态机，即在不同的状态下，角色需要做不同的反应。它的调用不是简单的switch或者if else可以完成的。如果要按照if else来写，那么需要维护一个巨大的if else嵌套，一个if之下可能有许多其他的if else。这会让整个代码的逻辑变得异常混乱。

当状态较少时，还可以厘清逻辑关系。当状态较多时，状态与状态之间的逻辑交互变为网状，这会使代码变得异常复杂，没有办法很好地维护。它们两两交互，在满足条件的情况下，从任何一个状态都可以直接跳到其他的状态。而每一个状态都有可能进入其他状态，如图4-5所示。为了让处理变得一目了然，需要做一个状态机的模板，以便在使用时可以方便调用。

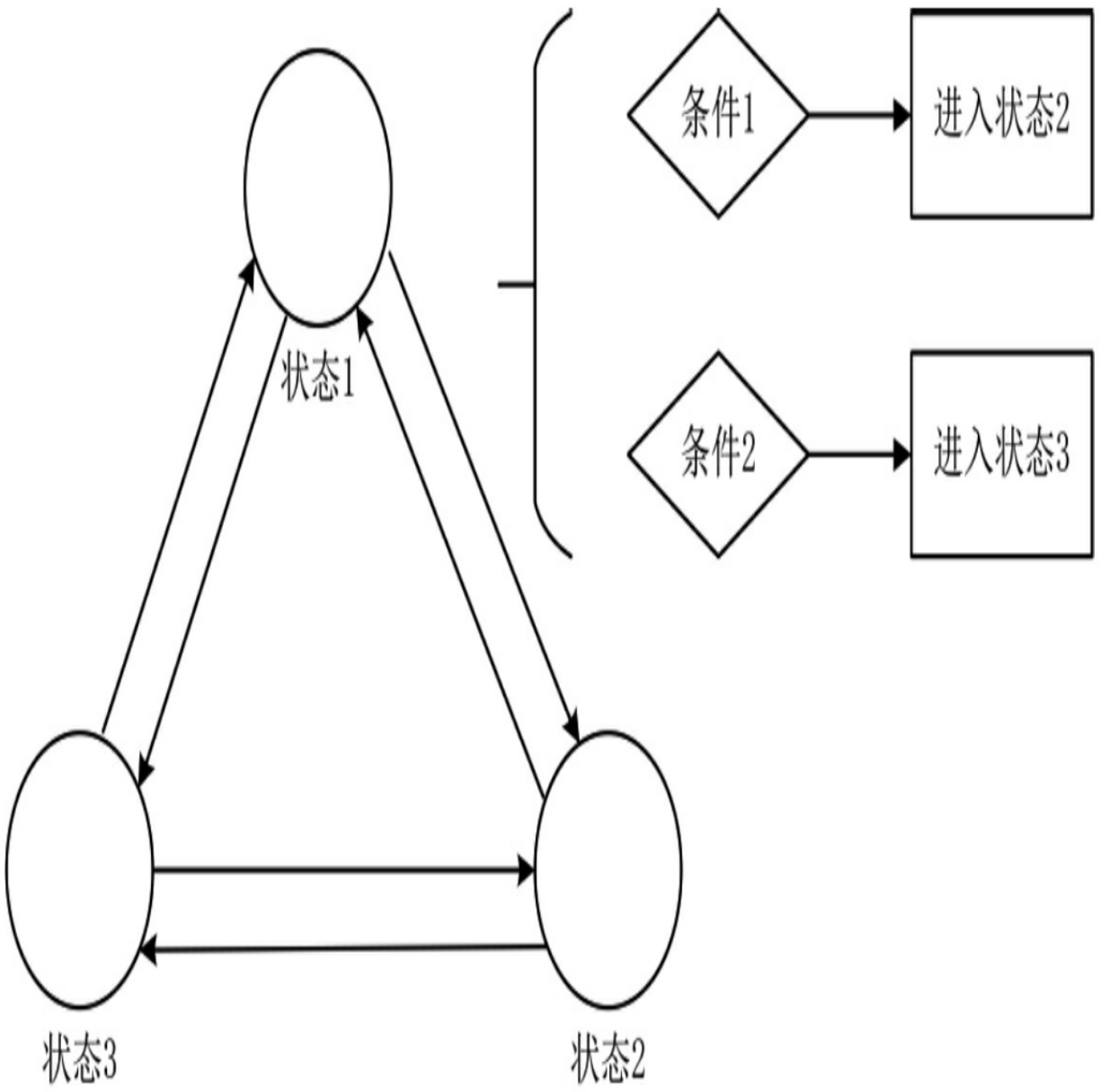


图4-5 状态机

4.8.2 状态机基类

首先，为每个状态编写一个基类，这个基类包括一些基本的操作，其定义如下：

```
template<typename enumType, class T>
class StateTemplate {
public:
    virtual enumType GetState() = 0;    // 得到当前的状态类型
    virtual enumType Update() = 0;      // 状态机内部的更新
    virtual void EnterState() = 0;       // 进入本状态
    virtual void LeaveState() = 0;       // 离开本状态
};
```

该模板类的定义在libserver工程的state_template.h文件中。T是其父类的类型，enumType是状态的枚举。StateTemplate类是每个状态对象的基类。每个状态只需要管理好自己的Update及进入和离开两个事件即可。打个比方，我们拥有一个Boss，当它血量减少到30%时，进入一种叫“狂躁”的状态。在这种状态之下，它会加速行走，并且在此状态下，它受到重击时需要使用一个回血的技能。

为Boss创建两个状态类，继承自StateTemplate，一个叫NormalState，另一个叫CrazyState。在NormalState的Update中，一旦检查到了血量低于30%，就进入一个新的状态CrazyState。NormalState的存活周期就结束了。在CrazyState生成时会调用EnterState事件，这时为角色设置加速，而在CrazyState的Update中，每一帧都会检查一下角色是否受到了重击，是否需要使用回血技能。当满足某个条件时，Boss回到NormalState状态，CrazyState状态退出时，重置速度为正常速度。

上面的例子充分说明了进入事件和离开事件的重要性。

综上所述，一个状态只管自己状态内发生的事情，至于是如何进入状态的，状态本身不关心，它更关心哪一种变化会让它退出当前状态，以及它在进入或者退出时，是否需要做一些特别处理。这种进入、退出函数的调用使得整个逻辑更清晰。

4.8.3 状态机管理类

每个需要维护状态的类，其代码都有相同的处理，框架中提供一个模板类来管理这些状态类。

关键点1：状态管理类定义

模板定义在state_template.h文件中，定义如下：

```
template<typename enumType, class StateClass, class T>
class StateTemplateMgr {
    ...
    typedef StateClass* (*CreateInstancePt)();
protected:
    std::map<enumType, CreateInstancePt> _dynCreateMap; // 每种状态的
    StateClass* _pState{ nullptr };                  // 当前状
    enumType _defaultState;                           // 最初
}
```

首先，在这个模板中维护了一个字典，该字典定义了一个状态枚举与创建函数的指针关系。其次，还维护了一个当前状态的指针。

关键点2：状态更换

先来看StateTemplateMgr模板每帧的实现：

```
void UpdateState() {
    if (_pState == nullptr) {
        ChangeState(_defaultState);
    }
    enumType curState = _pState->Update();
    if (curState != _pState->GetState()) {
        ChangeState(curState);
    }
}
```

如果当前状态为空，就进入默认状态。在UpdateState函数中，对当前状态进行Update操作，如果Update的返回值不再是当前状态，就说明有新状态产生，调用ChangeState进入新的状态。

```
void ChangeState(enumType stateType) {
    StateClass* pNewState = CreateStateObj(stateType);
    if (pNewState == nullptr) {
        return;
    }
    if (pNewState != nullptr) {
        if (_pState != nullptr) {
            _pState->LeaveState();
            delete _pState;
        }
        _pState = pNewState;
        _pState->EnterState();
    }
}
```

在ChangeState函数中，调用CreateStateObj来得到一个新状态的指针。接下来，旧状态调用LeaveState函数离开，新状态调用EnterState函数进入。

关键点3：状态是如何创建出来的

作为一个模板类，StateTemplateMgr要创建新的状态，采用的是函数注册的方式。函数CreateStateObj创建了一个指定类型的状态实例，其实现如下：

```
typedef StateClass* (*CreateInstancePt)();
StateClass* CreateStateObj(enumType enumValue) {
    auto iter = _dynCreateMap.find(enumValue);
    if (iter == _dynCreateMap.end())
        return nullptr;
    CreateInstancePt np = iter->second;
    StateClass* pState = np();
    pState->SetParentObj(static_cast<T*>(this));
    return pState;
}
void RegisterStateClass(enumType enumValue, CreateInstancePt np) {
```

```
        _dynCreateMap[enumValue] = np;  
    }
```

在StateTemplateMgr类中维护的字典dynCreateMap保存了状态类型与创建该状态的函数的指针关系，该字典的定义为std::map<enumType, CreateInstancePt>。这个字典的second值中保存了该状态的产生函数指针。这个函数指针是由RegisterStateClass函数注册生成的。

4.8.4 Robot类中的状态机

以Robot类来说明状态机的使用方式。对于一个新的状态，应该做哪几件事呢？

(1) 定义一个状态枚举。

(2) 为枚举写状态类，每个状态类都基于StateTemplate模板类。

(3) 选定管理类，并让管理类继承StateTemplateMgr，在适当的时机调用ChangeState函数进行状态切换。

对于管理状态的类来说，它有几个必要的处理：

(1) 初始化时，调用InitStateTemplateMgr初始化状态。

(2) 实现RegisterState虚函数，注册状态与状态类。

(3) 每帧调用UpdateState函数，更新状态类中的数据。

Robot类的状态枚举放在libserver项目中。在robot_state_type.h文件中定义了它目前可能有的状态类型：

```
enum RobotStateType {
    RobotState_Login_Connecting,    // 正在连接 login
    RobotState_Login_Connected,     // 连接成功
    RobotState_Login_LoggedIn,      // 登录成功，即账号验证成功
    ...
};
```

目前阶段，需要关心的枚举只有以上3个，下面为每个枚举写一个状态类。在这些状态中有一些相似的需求，把这些相似的功能写在RobotState基类中，该基类位于robots工程的robot_state.h文件中。其定义如下：

```
class RobotState : public StateTemplate<RobotStateType, Robot> {
public:
```

```

RobotStateType Update() override;
virtual RobotStateType OnUpdate() {
    return GetState();
}
void EnterState() override;
virtual void OnEnterState() { }
void LeaveState() override;
virtual void OnLeaveState() { }
};

```

基类RobotState继承了状态机模板类StateTemplate。第一个参数为枚举，值为RobotStateType。第二个参数是管理类StateTemplateMgr，此处为Robot类，所以Robot类必然继承了StateTemplateMgr。

在基类中有一个非常重要的操作，即检测Robot是否已断线，如果断线，就重新回到正在连接状态。其帧函数如下：

```

RobotStateType RobotState::Update() {
    const auto state = GetState();
    if (state > RobotState_Login_Connecting && state != RobotState
        if (!_pParentObj->IsConnected()) {
            return RobotState_Login_Connecting;
        }
    }
    return OnUpdate();
}

```

除了在连接状态下，任何状态发现Robot实例的网络状态发生了改变都需要做出反应。一旦断开，状态必然回到RobotState_Login_Connecting，告诉上层网络底层断开了。

当前Robot类的3个状态类的处理类定义在robots工程的robot_state_login.h文件中，限于篇幅，这里不再列举代码，在定义时使用了一个宏定义：

```

#define DynamicStateCreate(classname, enumType) \
    static void* CreateState() { return new classname; } \
    RobotStateType GetState( ) override { return enumType; }

```

在状态类中，每个状态类使用了DynamicStateCreate宏。该宏定义了一个静态函数，创建出当前状态的实例。宏中还定义了一个函数，用于返回自己的状态枚举。这两个函数对于所有状态类来说都一样，所以这里用宏来代替，以减少冗余代码。

梳理一下Robot类的3个状态，首先进入的是“连接中”这个状态，处理类为RobotStateLoginConnecting。在RobotStateLoginConnecting连接状态中，需要不断地询问底层是否已经连接上了。

```
RobotStateType RobotStateLoginConnecting::OnUpdate() {
    if (_pParentObj->IsConnected()) {
        return RobotState_Login_Connected;
    }
    return GetState();
}
```

如果已经连接上了，就会触发RobotState_Login_Connected状态，进入新状态时调用OnEnterState函数，该函数向服务器发送了验证协议。

```
void RobotStateLoginConnected::OnEnterState() {
    _pParentObj->SendMsgAccountCheck();
}
```

以类的方式来管理状态有一个好处是，一个状态类中只会关心自己的情况，不用考虑其他状态，这使得逻辑相对简单。流程图4-6展示了Robot类登录过程中的状态与事件的关系。

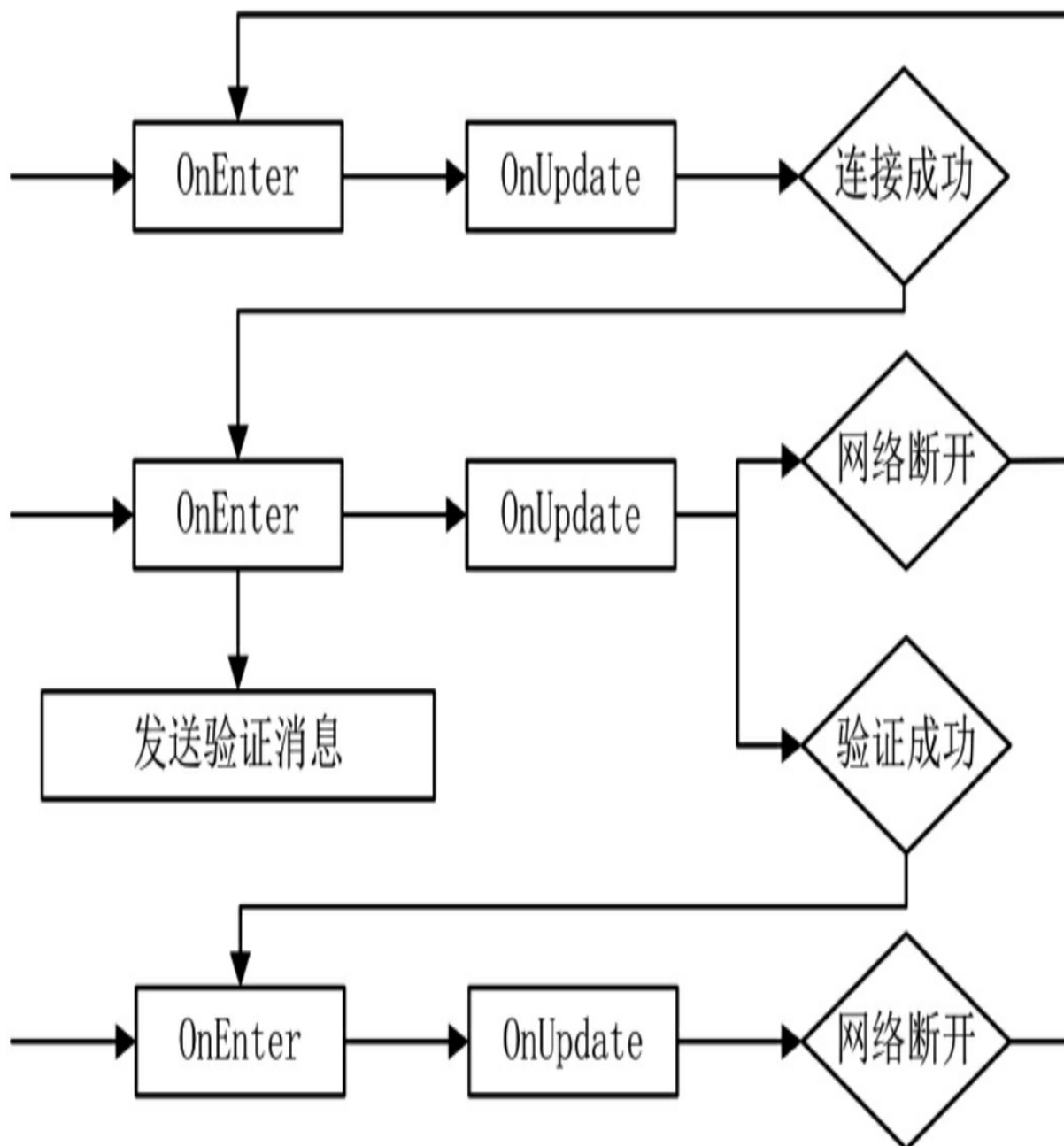


图4-6 Robot状态说明

现在状态类的逻辑弄清楚了，回到Robot类中来看一下模板管理类的使用，Robot类继承了状态管理模板类。其定义如下：

```
class Robot : public NetworkConnector, public StateTemplateMgr<Rob
```

Robot需要实现虚函数RegisterState，其功能是对状态类和其生成函数进行注册。

```
void Robot::RegisterState() {  
    RegisterStateClass(RobotStateType::RobotState_Login_Connecting  
    RegisterStateClass(RobotStateType::RobotState_Login_Connected,  
    RegisterStateClass(RobotStateType::RobotState_Login_Logined, D  
}
```

使用宏定义将处理类绑定到枚举上，宏定义如下：

```
#define DynamicStateBind(classname) reinterpret_cast<CreateInstance
```

在调用RegisterState时使用DynamicStateBind宏绑定了DynamicStateCreate宏中定义的函数。

以上代码实现了Robot的状态切换。有了状态机的这两个模板类，就能够以最少的代码量来完成一个新的状态管理。假如需要为Robot增加一种新的状态，只需要短短三步就可以实现：第一，增加一个状态枚举值；第二，为枚举值写一个实现类；第三，注册到Robot类中。

批量登录测试

到目前为止，我们已经建立了一个可以登录的基础框架，在这个基础框架中有网络类和消息过滤机制。作为后台服务端，有一个重要的功能需要我们去测试，那就是效率问题。

还是以04_02_engine_robots目录下的工程为例，关闭关于account check result部分的打印消息，robots进程和login进程中的打印都关闭。在这种秒级、毫秒级的效率测试中，日志和屏幕打印开销非常大，所以先关闭打印，来看一下批量登录的效果。在robots进程控制台中输入“login-ex test 200”，以瞬间登录200个账号来测试一下。

表4-1展示了两种网络模式下的性能，这个数据在每台机器上都不一样，但大同小异，读者一定会发现异常，Epoll的效率不是比Select高吗，为什么200个账号登录花了7秒，而Select却只花了0.15秒？太不可思议了，这是一个50倍的效率问题。代码都是一样的，只是在网络模型上的代码不同，这是为什么呢？这里需要着重说明很多人忽视的一个网络问题，就是Epoll的超时。重新来看一下这两个函数的声明。

表4-1 两种模式性能对比1

Select 模式下	Epoll 模式下
[root@localhost bin]# ./logind	[root@localhost bin]# ./logind
select model	epoll model
test begin	test begin
Login-Connectting over. time:0.013786s	Login-Connectting over. time:0.212692s
Login-Connectted over. time:0.014216s	Login-Connectted over. time:0.213287s
Login-Logined over. time:0.155839s	Login-Logined over. time:7.98523s

```
int select(int nfd, fd_set *readfd, fd_set *writefd, fd_set *ex
int epoll_wait(int epfd, struct epoll_event * events, int maxevent
```

两个函数最后一个参数都是timeout（超时），Select采用了时间结构，Epoll的参数是毫秒，在代码中，它们都是50毫秒，也就是0.05秒。可以做一个尝试，将这个值提高到6秒，以测试它的结果，你会发现Epoll变得更慢，而Select几乎没有什么改变。要明白出现这个结果的原因就要理解这两个函数的不同，Select对超时的定义是，如果在0.05秒之内有一个事件被触发，就立刻返回，而Epoll一定要等待0.05秒。当把超时修改为6秒时，Epoll会每次等待6秒，这是它变慢的原因。

现在将两个函数的超时都改为0，即立刻返回，再看看它们的表现。同样还是登录200个账号，结果展示在表4-2中。

表4-2 两种模式性能对比2

Select 模式下	Epoll 模式下
[root@localhost bin]# ./logind	[root@localhost bin]# ./logind
select model	epoll model
test begin	test begin
Login-Connecting over. time:0.03369s	Login-Connecting over. time:0.04865s
Login-Connected over. time:0.034204s	Login-Connected over. time:0.048793s
Login-Logined over. time:0.13833s	Login-Logined over. time:0.143942s

结果相差不大，这些结果根据机器的不同略有不同，不同机器之间的比较没有任何意义，而且内存环境也会影响这些数据，测试时可以多次执行并取平均值。

总结

本章中引入了账号验证功能。第5章先抛开框架，讨论另一个比较重要的话题——“性能优化与分析”。如果读者认真查看过代码，就会发现在之前的示例中，大部分情况下创建出了对象，但是并没有释放它们。

我们以现在这个工程为基础，在第5章详细分析如何优化它，如何利用工具查看内存泄漏问题，以及如何更好地创建内存对象等，为搭建效率更高、结构更好的框架打下坚实的基础。

第5章

性能优化与对象池

本章放慢脚步，对现有的工程进行一次优化。在本章中，将学习使用工具检查程序性能，同时介绍两个加快性能的数据结构以及对象池。

本章包括以下内容：

- 如何使用Windows和Linux系统下的性能优化工具。
- 两个提升性能的内存结构。
- 如何设计对象池。

Visual Studio性能工具

首先要使用的是Windows下的Visual Studio工具，相对Linux下的工具，它比较直观。还是以第4章中04_02_engine_robots目录下的robots工程为例，我们可以使用Visual Studio自带的性能查看器，打开“调试”菜单，单击“性能探查器”选项，选中“性能向导”选项，在选择性能分析页面中选择“检测，测量函数的计数和用时”选项，单击“开始”按钮。在下一个界面中，选中“一个或是多个可用项目”选项，单击“下一步”按钮，选择需要测试的工程robots，单击完成之后，Visual Studio会按正常流程启动robots。

在调试性能模式下启动的时长比较长，根据机器情况各不相同。robots启动完成之后，正常启动login进程，在robots命令行中输入“login-ex test 300”，批量登录300个账号来进行测试。完成之后，服务端打印信息如下：

```
λ .\login.exe
Login-Connecting over. time:14.0393s
Login-Connected over. time:14.0406s
Login-Logined over. time:31.0722s
```

在login进程中的数据显示，差不多花了30秒，因为有检测函数的介入，所以这个时间会比单纯执行程序要长。所有账号都登录完成之后，关闭robots，Visual Studio就会自动分析VSP文件，300个账号的vsp文件大约有2GB左右。因为vsp文件非常大，所以需要一定的时间，几秒、几分钟不等。打开后的界面如图5-1所示。



图 5-1 Visual Studio性能分析主界面

在上面的“当前视图”中选择“函数”视图，可以给出一个函数调用时长的列表，单击“已用非独占时间百分比”，排序如图5-2所示。

函数名	调用数	已用非独占时间百分比▼	已用独占时间百分比	平均已用非独占时间
ThreadObjectList::Update	21,712	87.81%	1.16%	15.00
std::_Invoker_functor::_Ca...	3	67.20%	0.67%	83,050.77
std::invoke<<lambda_c8...	3	67.20%	0.00%	83,050.77
std::thread::_Invoke<std::...	3	67.20%	0.00%	83,050.78
<lambda_c8d0e4f301d3c...	3	66.52%	0.00%	82,220.12
NetworkConnector::Update	19,074	26.77%	0.04%	5.20
Robot::Update	13,326	23.49%	0.00%	6.54
_scrt_common_main	1	22.41%	0.00%	83,089.42
mainCRTStartup	1	22.41%	0.00%	83,089.42

图5-2 Visual Studio性能分析函数列表

这里有两个概念需要我们厘清：

（1）非独占样本数是指包括子函数执行时间的总执行时间。

（2）独占样本数是不包括子函数执行时间的函数体执行时间，可理解为纯函数占用时间。

Visual Studio分析robots代码

现在详细查看robots工程的性能数据。打开“函数”视图，我们以“非独占时间百分比”来排序，即包含子函数的时长，然后看看是哪个函数消耗的时长最长。

位于榜首的就是函数void ThreadObjectList::Update()，双击该函数，看看它的详细情况。如图5-3所示，前3个函数调用消耗了80%的时长，这3个函数分别是std::this_thread::sleep_for、ThreadObject::Update和MessageList::ProcessPacket。

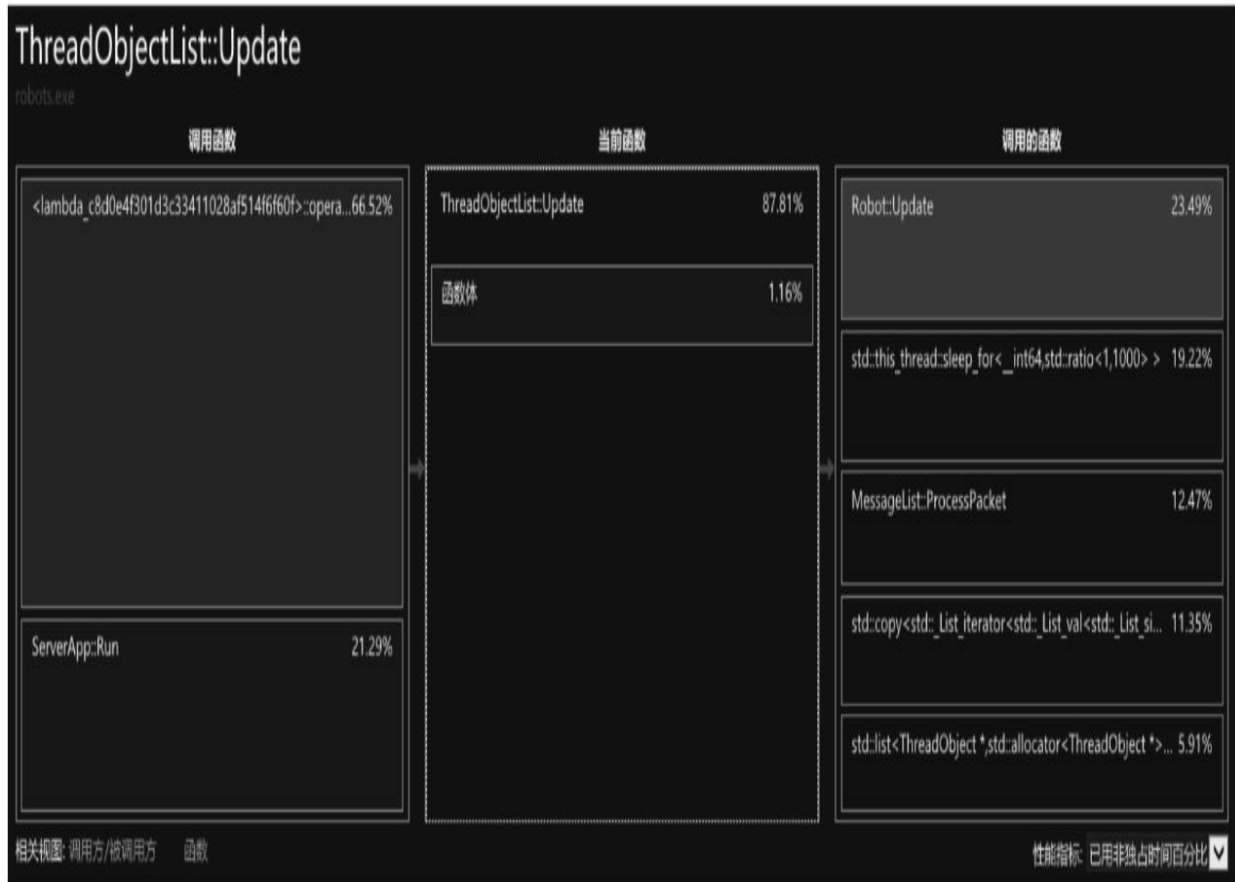


图5-3 Robot工程性能分析图——ThreadObjectList：：Update

关键点1: sleep_for

在线程中，我们调用了线程sleep_for函数，暂停了1毫秒，时间非常短。该函数的调用是让更多线程得到CPU的处理机会。sleep_for是没有办法优化的，我们跳过。

关键点2: ThreadObject：：Update

在ThreadObject：：Update的函数调用中，消耗时长最长的是Robot：：Update，如图5-4所示，这个函数调用了网络层的Network：：Update函数。



图5-4 Robot工程性能分析图——ThreadObject：：Update

继续向下查看，最终会发现调用瓶颈走到ThreadObjectList：：AddPacketToList这个函数上，如图5-5所示。



图5-5 Robot工程性能分析图——ThreadObjectList：：AddPacketToList

ThreadObjectList类是线程类Thread和线程管理类ThreadMgr的基类，是管理ThreadObject列表的一个基类。分析瓶颈出现的原因，先看一看调用堆栈：

(1) 当我们收到网络层的消息时，会调用ConnectObj：：Recv函

数。

(2) 函数ConnectObj::Recv将数据封装为Packet类，这时需要做的是向整个框架所有线程进行广播，调用为：ThreadMgr::GetInstance()->DispatchPacket(pPacket)。

(3) 在ThreadMgr::DispatchPacket函数中，调用了瓶颈的AddPacketToList函数。

详细看一下这两个函数的实现：

```
void ThreadMgr::DispatchPacket(Packet* pPacket) {
    // 主线程
    AddPacketToList(pPacket);
    // 子线程
    std::lock_guard<std::mutex> guard(_thread_lock);
    for (auto iter = _threads.begin(); iter != _threads.end(); ++i
        Thread* pThread = iter->second;
        pThread->AddPacketToList(pPacket);
    }
}

void ThreadObjectList::AddPacketToList(Packet* pPacket) {
    std::lock_guard<std::mutex> guard(_obj_lock);
    for (auto iter = _objlist.begin(); iter != _objlist.end(); ++i
        ThreadObject* pObj = *iter;
        if (pObj->IsFollowMsgId(pPacket)) {
            pObj->AddPacket(pPacket);
        }
    }
}
```

函数DispatchPacket遍历线程，函数AddPacketToList的主要作用是将网络层的包分发到各线程中的对象上去。这个函数是一个非常典型的处理不当的案例，是多线程中我们会经常犯的错。看上去这个逻辑没有任何问题，为什么说它处理不当呢？

整个逻辑的瓶颈在于Network实例读取Packet数据的地方，在分发消息时，在Network实例中一次遍历了所有对象，这显然是一个串行操作，不是我们想要的并行效果。作为并发来说，在分发时只需要将Packet发到线程中即可。至于Packet在线程中如何处理，它要分发到哪

个对象上，分发时是不需要关心的。这是多线程编码特别需要注意的一个点，一不小心并行就会变成串行。

关键点3: MessageList: : ProcessPacket

在性能分析中，可以看到std: : copy花费了大量的时间，下面分析一下原因。

300个账号就有300个robots对象，如果每个对象收到服务端发来的一条数据，就有300个Packet，这些Packet在处理之前都会缓存在MessageList中，最后在Update函数中std: : copy出来使用，std: : copy本身就很消耗时间，在Update函数中使用时会引起瓶颈。

另外，考虑到一种情况，我们设计ThreadObject的时候是按Actor模型设计的，Actor需要有一个MailBox来收发协议，所以有了MessageList的存在，但Actor与线程相结合时可以优化。我们并不需要将Packet放到每个MessageList中，而是可以缓存在线程中，对于一个线程而言，这些Packet是共用的。

以上3个关键点就占用了整个程序执行时间的80%，当然其中部分原因是这个程序本身的功能是比较少的。根据性能分析文件还可以看出，有大量std库的操作。对std库的使用需要注意一点，就是尽量减少循环的使用，采用事件或者观察者模式的方式来重构数据。下面就以以上内容为整个框架做一次数据优化。

内存中的数据结构

数据结构是计算机组织数据的方式，精心选择的数据结构可以带来更高的运行或者存储效率。下面将介绍两种适合我们框架的数据结构：一种是交换型数据结构；另一种是刷新型数据结构。

5.2.1 交换型数据结构

做优化分析时，有一个比较消耗时间的函数`std::copy`。`std::copy`用在了3个地方：一个是Thread对线程中管理的对象进行操作时，一个是在消息队列中，还有一个是在网络层发送数据时。以发送数据为例：

```
void Network::SendPacket(Packet* pPacket) {
    std::lock_guard<std::mutex> guard(_sendMsgMutex);
    _sendMsgList.push_back(pPacket);
}
```

发送数据时，将Packet放到一个List中，调用发送数据的对象可能与网络类不在同一个线程中，所以必须加锁。这些需要发送的数据在更新函数中被取出，真正地被发送到ConnectObj对象的系统发送缓存中去。在这个过程中使用了`std::copy`，代码如下：

```
void Network::Update() {
    std::list<Packet*> _tmpSendMsgList;
    _sendMsgMutex.lock();
    std::copy(_sendMsgList.begin(), _sendMsgList.end(), std::back_inserter(_tmpSendMsgList));
    _sendMsgList.clear();
    _sendMsgMutex.unlock();

    for (auto pPacket : _tmpSendMsgList) {
        ... // SendPacket
    }
    _tmpSendMsgList.clear();
}
```

从上面的操作可以看出，在每帧循环的时候将Packet列表复制到一个临时的数据中，释放锁之后，再对这个临时数据进行处理。有没有一种数据结构既高效又不用`std::copy`呢？当然有，姑且称之为交换数据。

先来介绍它的原理。如图5-6所示，准备了两块数据，一块用于读，另一块用于写，开始时它们都是空的。在某个时间段，发现写数据

块中有数据需要处理，这时将两个指针对调。这时写指针指向了一个空的数据块，而读指针的数据块上充满了数据，将需要读取的数据都处理完。等下一个时刻，发现写数据块有数据，再进行数据块的对调。因为读、写数据分开了，就不再需要加锁了。

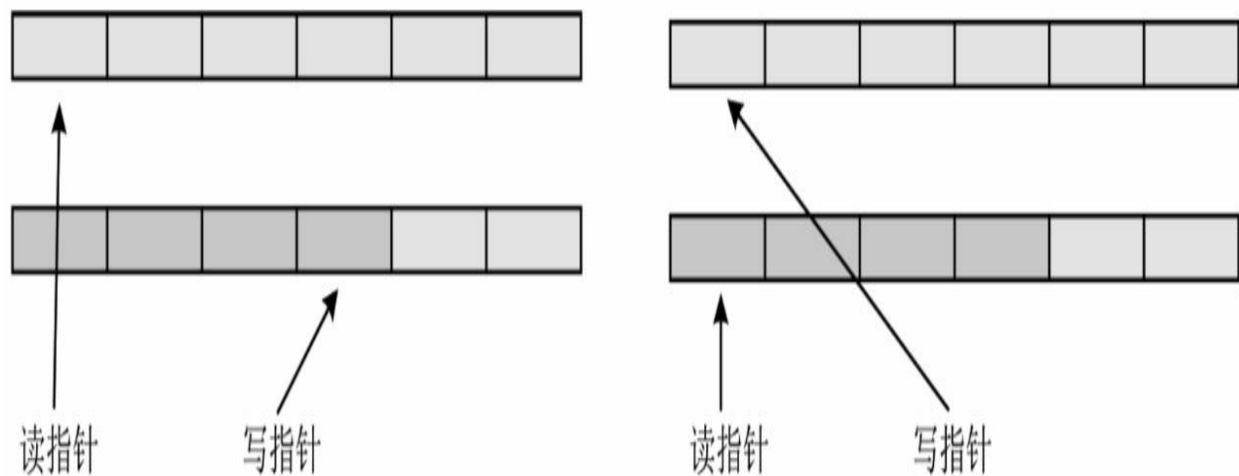


图5-6 交换内存指针

本节的源代码在05_01_pf目录的libserver工程的cache_swap.h文件中。下面来看看具体实现。

```
template<class T>
class CacheSwap {
public:
    std::list<T*>* GetWriterCache();
    std::list<T*>* GetReaderCache();
    void Swap();
    bool CanSwap();
    ...
private:
    std::list<T*> _caches1;
    std::list<T*> _caches2;
    std::list<T*>* _readerCache;
    std::list<T*>* _writerCache;
};
```

CacheSwap类维护了两个队列caches1和caches2，而readerCache和writerCache分别指向两个缓存中的其中一个。CacheSwap类有一个基本

的对调操作——Swap函数，实现如下：

```
template <class T>
void CacheSwap<T>::Swap() {
    auto tmp = _readerCache;
    _readerCache = _writerCache;
    _writerCache = tmp;
}
```

什么时候需要对调呢？就是写数据队列中有数据时，实现代码如下：

```
template <class T>
bool CacheSwap<T>::CanSwap() {
    return _writerCache->size() > 0;
}
```

交换数据类型可以用在哪些场合呢？典型的处理Packet时，Packet被分配到各个线程待处理，有一个写入的操作，同时各个线程在各自的循环中，还要处理已写入的Packet，有一个读取操作。下面来看优化前后的代码对比，以及优化后CacheSwap的使用方式。优化前的代码在04_02_engine_robots目录，优化后的代码在05_01_pf目录。

关键点1：定义

首先用CacheSwap类替换list类。首先，要替换的是Packet，存储Packet的地方。新定义如下：

```
class ThreadObjectList: public IDisposable {
...
protected:
    std::mutex _packet_lock; // 本线程中的所有待处理包
    CacheSwap<Packet> _cachePackets;
};
```

在优化前，由网络接收到的Packet类是由ThreadObject的基类

MessageList来管理的，存储数据结构为List，在同一个线程中，同一个Packet指针会产生几份不同的副本。优化后将Packet放在线程中，以减少不必要的存储。同时，使用了CacheSwap模板交换缓冲区的读写方式。

关键点2：写数据

在对CacheSwap模板交换缓冲区中写入数据时，代码非常简单。还是以Packet类为例，当Packet进入线程时，调用AddPacketToList函数，其实现如下：

```
void ThreadObjectList::AddPacketToList(Packet* pPacket) {
    std::lock_guard<std::mutex> guard(_packet_lock);
    _cachePackets.GetWriterCache()->emplace_back(pPacket);
}
```

在没有优化的代码中，函数AddPacketToList对线程中所有对象进行了遍历，将Packet放置到所有对它感兴趣的ThreadObject中，现在这一步操作被推迟到线程的下一帧。函数AddPacketToList只是将Packet存下来而已。在优化后的代码中，取出CacheSwap类写指针进行保存操作，在上层调用的时候，不需要关心这时的写指针指向的是两个队列中的哪一个队列。

关键点3：读数据

在读数据时也做了相应的优化修改。还是以Packet为例，当它被写入写缓冲之后，在下一帧中，帧函数会对其进行处理。其实现如下：

```
void ThreadObjectList::Update() {
    ...
    _packet_lock.lock();
    if (_cachePackets.CanSwap()) {
        _cachePackets.Swap();
    }
    _packet_lock.unlock();
    auto pList = _objlist.GetReaderCache();
    auto pMsgList = _cachePackets.GetReaderCache();
}
```

```

        for (auto iter = pList->begin(); iter != pList->end(); ++iter)
            auto pObj = (*iter);
            for(auto itMsg = pMsgList->begin(); itMsg != pMsgList->end()
                auto pPacket = (*itMsg);
                if (pObj->IsFollowMsgId(pPacket))
                    pObj->ProcessPacket(pPacket);
            }
            pObj->Update();

            ...
        }
        pMsgList->clear();
    }
}

```

在之前的帧函数处理数据时，采用了std::copy创建副本的方式，将整个队列复制一份。现在仅调用了CanSwap函数和GetReaderCache函数，效率的提升是非常显著的。本例中增加了一种数据结构，即可将之前使用std::copy的耗时操作替代了，是一种非常取巧的数据策略。

在优化后的代码中，将之前写数据时的某些操作移到了读数据时进行。在优化之前的代码中，当一个Packet到来时，ThreadMgr会遍历所有线程，调用AddPacketToList函数将Packet传到所有线程中的每个对象去判断，确认是否对Packet的MsgId感兴趣，如果感兴趣，就将Packet指针转存到MessageList中。假设在某一帧中收到了10个Packet，在10个进程中也就需要循环 $10 \times 10 \times N$ 次，这个N取决于线程中有多少类对象。而新的解决方案只需要在线程中把Packet存起来，收集所有Packet，统一在下一帧中一起处理，从而减少了循环。

优化前后，这两种方法的结果一样，但它们在网络层花费的时间却完全不一样。图5-7展示了在两种情况下的时间开销。

图5-7左侧为优化前的情况，如果遍历线程对象的耗时为0.01秒，那么在网络层所在的线程，一次Packet会产生 $N \times 0.01$ 秒的时长，N为线程数量。而优化后，将这0.01秒分散在了多个线程中，不会影响网络层的所有线程。还记得Epoll超时中那个50毫秒吗？一帧中的50毫秒降低了50倍的效率，所以不要小看这0.01秒。使用多线程锁时需要注意尽可能让锁住代码的执行时间变短，这样线程与线程之间的影响就会减少。

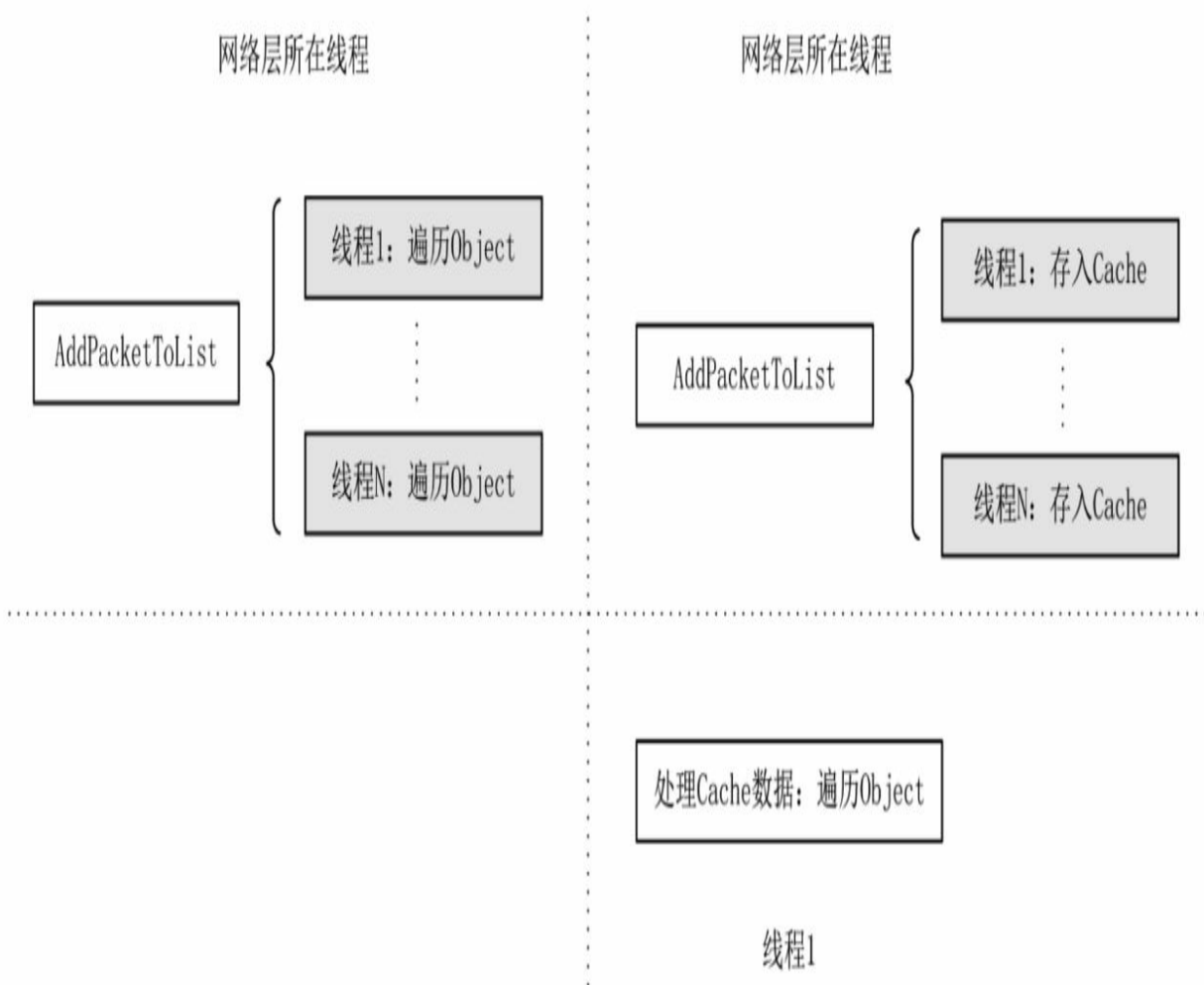


图5-7 优化前后的时间开销

5.2.2 刷新型数据结构

除了交换型数据结构外，工程中还增加了一个刷新型数据结构，定义在05_01_pf目录的login工程的cache_refresh.h文件中，其定义如下：

```
template<class T>
class CacheRefresh :public IDisposable {
public:
    std::list<T*> Swap();        // 返回值为删除的Obj，后续是否有内存回收处
    bool CanSwap();
    ...
protected:
    std::vector<T*> _reader;
    std::vector<T*> _add;
    std::vector<T*> _remove;
};
```

CacheRefresh类与CacheSwap类的区别在于，CacheSwap用于数据块的交换，而CacheRefresh用于对象的增加与删除。一个典型的使用是线程中的对象队列以及后面将要讲到的对象池。下面来看线程管理基类ThreadObjectList是如何使用它的。

```
class ThreadObjectList: public IDisposable {
public:
    void AddObject(ThreadObject* _obj);
    ...
protected:
    // 本线程的所有对象
    std::mutex _obj_lock;
    CacheRefresh<ThreadObject> _objlist;
};
```

因为线程中的对象总是动态增加或者减少的，导致每次使用队列时必须加锁。而使用CacheRefresh类，加锁的频率可以大幅度降低。

```
void ThreadObjectList::AddObject(ThreadObject* obj) {
```

```

        std::lock_guard<std::mutex> guard(_obj_lock);
        ...
        _objlist.GetAddCache()->emplace_back(obj);
    }

```

在需要增加对象时向增加的队列写数据，在需要减少对象时向删除队列写数据，而当前的可读队列在当前帧不会有数量的变更，也就不会有加锁与不加锁的烦恼。同样，在更新函数中，调用Swap函数将删除与增加的数据应用到读队列中。

```

void ThreadObjectList::Update() {
    _obj_lock.lock();
    if (_objlist.CanSwap()) {
        auto pDelList = _objlist.Swap();
        for (auto pOne : pDelList) {
            pOne->Dispose();
            delete pOne;
        }
    }
    _obj_lock.unlock();
    ...
    auto pList = _objlist.GetReaderCache();
    auto pMsgList = _cachePackets.GetReaderCache();
    for (auto iter = pList->begin(); iter != pList->end(); ++iter)
        auto pObj = (*iter);
        ...
        pObj->Update();
        if (!pObj->IsActive()) {
            _objlist.GetRemoveCache()->emplace_back(pObj); // 非激活
        }
    }
    ...
}

```

在CacheRefresh数据结构中采用了延迟应用的原则。虽然执行了增加或者删除的函数，但只是将数据进行缓存，在本帧中不会做出真正的处理。

这个原则除了可以减少锁的应用外，还可以阻断递归。在编码的时候，可能会遇到这种情况：当在一个对象List循环中删除满足条件的对象A时，对象A或许又删除了对象B，而对象B可能就在正在循环的对象

List中，这时服务器必定宕机。采用延迟应用的原则，将删除、增加对象的操作移到本次循环逻辑之外，在下一帧再对数据进行处理。

现在编译执行一下robots工程，就会发现执行之后没有显而易见的函数瓶颈了。

5.3

gprof

除了Windows系统上的Visual Studio，在Linux上也有很多简单好用的性能分析工具，本章简要介绍几种，其中之一就是gprof。它是GNU套件中的一个工具，分析的内容与Windows系统大致相似，如每个函数的调用次数、调用时长，可以方便找到系统的瓶颈。查看gprof的版本：

```
[root@localhost bin]# gprof -v
GNU gprof version 2.27-28.base.el7_5.1
```

CentOS 7上默认安装的是2.27版本。先用一个简单的程序来说明gprof的使用方式，本例源代码在05_02_gprof目录中。

在目录中有一个test_gprof.cpp文件。使用gprof的时候需要注意一点，生成可执行文件时需要加参数“-pg”。在目录下已准备好了CMakeLists.txt文件，可以使用CMake进行编译，会生成一个test_pd可执行文件。整个测试文件非常简单：

```
void fun2() {
    for (int i = 0; i < 10000000; i++);
}
void fun1() {
    for (int i = 0; i < 10000000; i++);
    fun2();
}
int main(void) {
    fun1();
    return 0;
}
```

```
}
```

在主函数中调用了10 000 000次fun1，fun1中又调用了10 000 000次fun2，下面看看执行效果：

```
[root@localhost 05_02_gprof]# cmake3 ./
[root@localhost 05_02_gprof]# make
[root@localhost 05_02_gprof]# ./test_pd
```

执行test_pd之后，在同目录下就会多出一个分析结果gmon.out文件。注意gmon.out名字是写死的，也就是说同目录下如果有多个可执行文件，那么最后生成的都是gmon.out，官方文档中说gmon.out会被最后一个写文件的进程覆盖。也就是说，如果在同一目录下同时执行多个带有-pg参数的程序，那么最后只会有一个gmon.out，也就是最后关闭的那个进程的gmon.out文件。查看分析文件时，使用gprof打开gmon.out文件，就会打印出很多分析信息：

```
[root@localhost 05_02_gprof]# gprof test_pd gmon.out
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative    self           self         total
time  seconds    seconds   calls  ms/call  ms/call  name
50.72    0.04      0.04         1     40.58    81.16   fun1()
50.72    0.08      0.04         1     40.58    40.58   fun2()
...
```

从代码可以知道fun1中调用了fun2，在分析数据中，fun1的total（总的时长）包括fun2的调用时长，从上面的打印数据中可以看到self.call和total.call的区别，正好是Windows系统下讲的独占与非独占的区别。

使用如下命令可以把这个结果导出：

```
[root@localhost 05_02_gprof]# gprof -b test_pd gmon.out > output.t
```

这样结果就保存到TXT文件中了，当函数比较多时，用导出的方式比Visual Studio工具更便于我们查看数据。上面的命令中，参数“-b”去掉了一些说明文件，在查看gmon.out文件时也可以使用gprof-b test_pd gmon.out。去掉了说明性的解释，打印数据更加简洁。

5.3.1 gprof调用堆栈图

除此之外，还可以使用gprof查看函数调用堆栈图。为了导出可以看到堆栈图，需要用到一个Python脚本。一般系统已经默认安装了一个Python的版本。不太了解Python的语法也没有关系，我们只是调用，不需要修改Python文件。

```
[root@localhost ~]# python
Python 2.7.5
```

执行python命令查看版本，以确保Python已经成功安装。在安装gprof之前最好更新一下yum。再安装gprof流程图生成的工具，步骤如下：

```
[root@localhost ~]# yum -y update
[root@localhost ~]# yum install graphviz
```

关键点1：安装gprof2dot

源代码位于<https://github.com/jrfonseca/gprof2dot>，源代码页面包含使用说明文档。较新的版本是gprof2dot-2017.09.19.tar.gz，在本书源代码目录的dependence目录下有一个复制文件。先解压安装文件，进入目录进行安装，步骤如下：

```
[root@localhost dependence]# tar xf gprof2dot-2017.09.19.tar.gz
[root@localhost gprof2dot-2017.09.19]# python setup.py install
```

如果在执行安装时出现“No module named setuptools”错误，就安装Python setuptools工具，命令为“yum-y install python-setuptools”。

关键点2：生成PNG文件

将用gprof导出的TXT文件用gprof2dot工具生成可查看文件，命令如下：

```
[root@localhost 05_02_gprof]# gprof2dot output.txt > output.dot
[root@localhost 05_02_gprof]# dot -Grankdir=LR -Tpng output.dot >
```

也可以用通道合并起来执行：

```
[root@localhost bin]# gprof2dot -n30 output.txt | dot -Grankdir=LR
```

命令dot中的Tpng指定输出PNG图，而Grankdir是按LR从左到右排列的，默认为从上到下排列。更多的排列属性可以执行“dot-help”查看。因为本例非常简单，所以图也不复杂，其堆栈如图5-8所示。

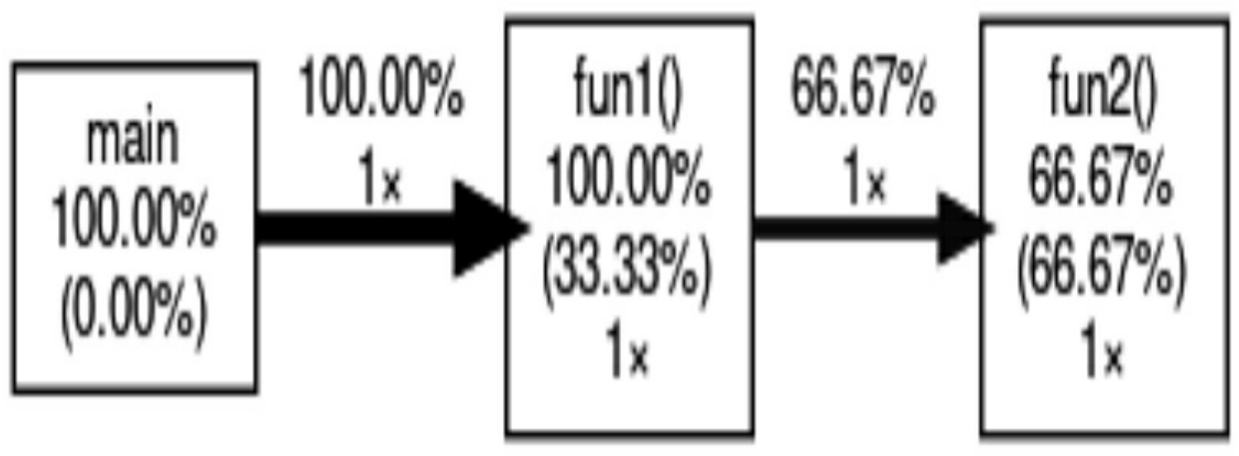


图5-8 gprof测试堆栈图

堆栈图更为直观地呈现了程序的调用关系，方便编码时厘清思路。关于gprof的更多信息可参见GNU的最新版说明文档。

5.3.2 让进程安全退出

了解了gprof的基本使用，下面用它查看一下我们的程序。以5.2节的源代码05_01_pf为蓝本，在login或者robots工程的CMakeLists.txt文件的CMAKE_CXX_FLAGS_DEBUG属性中增加了-pg参数，修改如下：

```
set(CMAKE_CXX_FLAGS_DEBUG "-D_DEBUG -O0 -g -ggdb -pg")
```

编译之后，在bin目录下生成了login和robots两个可执行文件，但是我们发现，在执行了login，用Ctrl+C组合键将它中断之后并没有out文件生成。原因在于gprof只能在程序正常结束退出时才能生成out文件，我们的程序并没有正常退出，是用的中断方式强制让它退出的。这是我们第一个需要修改的功能，在Linux下不论哪一个分析工具，几乎都需要进程安全退出才能进行分析。在Windows下，关闭窗口就退出了程序，往往忽略了“安全退出”这个功能。

退出进程不就是关闭进程，这有什么好考虑的。在Windows下，将命令行窗口关掉不就退出进程了吗？考虑一种情况：如果需要关服时还有玩家在线，一般是如何操作的？大部分情况下会有公告滚动提示玩家下线，但总有一些玩家不会下线。在这种情况下，如果直接关闭进程，玩家数据就会丢失。另一种情况：排行榜数据长驻于内存中，如果在关闭进程之前没有对它进行存储操作，数据就会丢失。

所谓“安全退出”，即完成了所有该完成的事情，把该存的数据即时存储，再按正常方式执行到main函数的return语句上，退出进程。我们需要重新审视一下退出流程，“安全退出”是游戏服务器应有的基本功能，有以下几点需要注意：

关键点1：以信号的方式退出进程

捕捉退出信号。一般来说，这里可以分为两种情况：一种情况是捕捉系统发出的退出信号；另一种是从第三方进程发来退出协议，例如GM管理工具。先处理第一种情况，参考源代码在05_03_engine_gprof目录工程中。要捕捉系统信号，需要一个系统函数：`signal`，该函数注册了信号的处理函数，这个函数关联的头文件在Windows和Linux这两个操

作系统中是不同的：

```
#if ENGINE_PLATFORM != PLATFORM_WIN32
#include <signal.h>
#else
#include <csignal>
#endif
```

在Linux下使用vim查看“/usr/include/signal.h”文件，可以看到signal的原型：

```
typedef void (*__sighandler_t) (int);
__sighandler_t signal (int __sig, __sighandler_t __handler);
```

打开libserver工程的server_app.cpp文件，在初始化时调用了这个底层函数。

```
ServerApp::ServerApp(APP_TYPE appType) {
    signal(SIGINT, Signalhandler);
    ...
}
void ServerApp::Signalhandler(const int signalValue) {
    switch (signalValue) {
    #if ENGINE_PLATFORM != PLATFORM_WIN32
        case SIGSTOP:
        case SIGQUIT:
    #endif
        case SIGTERM:
        case SIGINT:
            Global::GetInstance()->IsStop = true;
            break;
    }
}
```

Linux下的信号种类繁多，这里不逐一举例，有一个重点需要处理的信号是SIGTERM。这是一个中断信号，也就是说在程序收到SIGTERM信号时，系统正在强制中断我们的程序，这个时候对于程序来说，收到这个消息之后需要做一些退出前的善后操作。

在本例中，ServerApp类初始化时，注册信号处理函数，收到信号之后将Global::IsStop的值设为true，表示进程处于关闭状态。

关键点2：停止所有线程

收到停止信号之后，Global::IsStop为true时，需要安全退出所有子线程。下面的代码展示了完整停止主循环的处理方式。

```
void ServerApp::Run() const {
    while (!Global::GetInstance()->IsStop) {
        UpdateTime();
        _pThreadMgr->Update();
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
    // 停止所有线程
    std::cout << "stopping all threads..." << std::endl;
    bool isStop;
    do {
        isStop = _pThreadMgr->IsStopAll();
        std::this_thread::sleep_for(std::chrono::milliseconds(100))
    } while (!isStop);
    // 释放所有线程资源
    std::cout << "disposing all threads..." << std::endl;
    // 1.子线程资源
    bool isDispose;
    do {
        isDispose = _pThreadMgr->IsDisposeAll();
        std::this_thread::sleep_for(std::chrono::milliseconds(100))
    } while (!isDispose);
    // 2.主线程资源
    _pThreadMgr->Dispose();
}
```

在没有退出信号时，正常情况下，逻辑一直处于while (!Global::GetInstance()->IsStop) 这个主循环中，如果IsStop为真，就退出循环。随后等待所有线程退出之后，再退出主线程，如果退出主线程时有子线程没有完全退出，在Linux下就会抛出一个异常——terminate called without an active exception，一旦抛出异常，gprof就不会生成gmon.out分析文件，因为程序没有真正退出。所以，在退出主线程循环之后，检查一下子线程是否完全退出，如果完全退出，

就关闭所有线程，以达到安全退出的目的。

现在按Ctrl+C组合键之后，向系统发起中断信号会触发安全退出，同时框架提供了一条命令来安全退出，即在控制台输入“exit”这样的命令退出程序。为此，工程中写了一个类Console来处理这种情况。无论是在Windows还是Linux下，输入“exit”进程都会安全退出。处理函数在libserver工程的Console类中，本书不再讨论它的实现，有兴趣可以直接查看源代码。

5.3.3 用gprof工具查看框架

下面用gprof查看程序的堆栈图，并查看其效率。本例的测试工程位于目录05_03_engine_gprof中。先以登录一个账号为例，退出之后查看一下生成的gmon文件。启动login进程，再启动robots进程，执行“login-a account”命令，代码如下：

```
[root@localhost bin]# ./robotsd
login -a test1
...
exit
```

输入“exit”退出程序之后得到了gmon.out分析文件。

关键点1：时间函数

```
[root@localhost bin]# gprof -b robotsd gmon.out > output.txt
```

文件output.txt会生成函数调用列表，内容大致如下：

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
14.29	0.01	0.01	69289	0.00	0.00	
14.29	0.02	0.01	56047	0.00	0.00	std::mute
14.29	0.03	0.01	26413	0.00	0.00	...
14.29	0.04	0.01	13292	0.00	0.00	ThreadObj
14.29	0.05	0.01	10875	0.00	0.00	...
14.29	0.06	0.01	8125	0.00	0.00	Network::
14.29	0.07	0.01	8120	0.00	0.00	Network::
...						

我们发现调用时长都是0，原因是在使用多线程时gprof无法统计调用时长，但函数调用次数的统计结果是正确的。鉴于篇幅原因，无法将整个文件打印出来。分析这些数据，发现调用较多的有一个函数：std::chrono::duration，这是一个时间对比的函数。

分析一下代码，发现是因为ServerApp：：UpdateTime函数在主线程会频繁调用，取得当前帧的系统时间。

标准库中的std：：chrono可能没有想象中那么高的效率，下面来做测试。目录05_03_test_time下包含测试代码：

```
[root@localhost 05_03_test_time]# cmake3 ./
[root@localhost 05_03_test_time]# make
```

在测试代码中定义了两个函数，分别对std：：chrono和Linux下的gettimeofday函数进行了性能测试。执行1000万次调用，每一次都取出时间，单位为微秒。看一下测试源代码：

```
#define COUNT 10000000
long long time_interval(const struct timeval* start, const struct
    // 1秒 =10^6微秒
    return ((end->tv_sec * 1000000ll) + end->tv_usec) - ((start->t
}
void fun2() {
    struct timeval start, end;
    gettimeofday(&start, nullptr);
    for (int i = 0; i < COUNT; i++) {
        auto timeValue = std::chrono::time_point_cast <std::chrono
        auto timeTick = timeValue.time_since_epoch().count();
    }
    gettimeofday(&end, nullptr);
    auto interval = time_interval(&start, &end);
    std::cout << "std::chrono spend:" << interval << " us " << int
}
void fun1() {
    struct timeval start, end;
    gettimeofday(&start, nullptr);
    for (int i = 0; i < COUNT; i++) {
        struct timeval tv;
        gettimeofday(&tv, nullptr);
        auto timeTick = tv.tv_sec * 1000 + tv.tv_usec * 0.001;
    }
    gettimeofday(&end, nullptr);
    auto interval = time_interval(&start, &end);
    std::cout << "gettimeofday spend:" << interval << " us " << in
}
int main(void) {
```

```
    fun2();  
    fun1();  
    return 0;  
}
```

看看它们的耗时：

```
[root@localhost 05_03_test_time]# ./test  
std::chrono spend:5415714 us 5.41571 s  
gettimeofday spend:664065 us 0.664065 s
```

调用std::chrono时，1000万次耗时5秒，也就是一秒钟调用了200万次。这种耗时情况不会对整体框架产生严重影响，但是为了让其他耗时的调用突显出来，我们还是选择了gettimeofday，毕竟调用gettimeofday比调用std::chrono快了10倍。gettimeofday是Linux系统下的时间函数，定义在头文件sys/time.h中，定义如下：

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

结构timeval在前面已介绍过，它的定义如下：

```
struct timeval {  
    time_t      tv_sec;        /* seconds */  
    suseconds_t tv_usec;      /* microseconds */  
};
```

第二个参数是timezone，用来取得时区：

```
struct timezone {  
    int tz_minuteswest;        /* minutes west of Greenwich */  
    int tz_dsttime;            /* type of DST correction */  
};
```

因为不需要时区，所以这里可以传入nullptr。现在可以对函数

ServerApp: : UpdateTime进行重构:

```
void ServerApp::UpdateTime() const {
    #if ENGINE_PLATFORM != PLATFORM_WIN32
        struct timeval tv;
        gettimeofday(&tv, nullptr);
        Global::GetInstance()->TimeTick = tv.tv_sec * 1000 + tv.tv_us
    #else
        auto timeValue = std::chrono::time_point_cast<std::chrono:
        Global::GetInstance()->TimeTick = timeValue.time_since_epoch()
    #endif
}
```

在Windows和Linux平台上做了不同的时间函数调用，在Windows下依然调用std: : chrono库来获取时间。

关键点2: 堆栈图

```
[root@localhost bin]# gprof ./robotsd | gprof2dot -n10 -e10 | dot
```

工具生成分析文件之后，可以生成调用堆栈图，因为程序的复杂度较高，生成的调用堆栈图也会非常复杂。如果我们想查看更为完整的调用图，则可以输入以下命令将其转成为SVG图。

```
[root@localhost bin]# gprof ./robotsd | gprof2dot -n0 -e0 | dot -T
```

完整堆栈文件在05_03_engine_gprof\bin\目录中。图5-9是主线程堆栈图，可以非常清晰地看到工程的结构。在robots主线程中放置了RobotsMgr类，该类是一个NetworkConnector类，在某些时刻与服务器进行通信，图5-9非常准确地展示了这一调用堆栈。

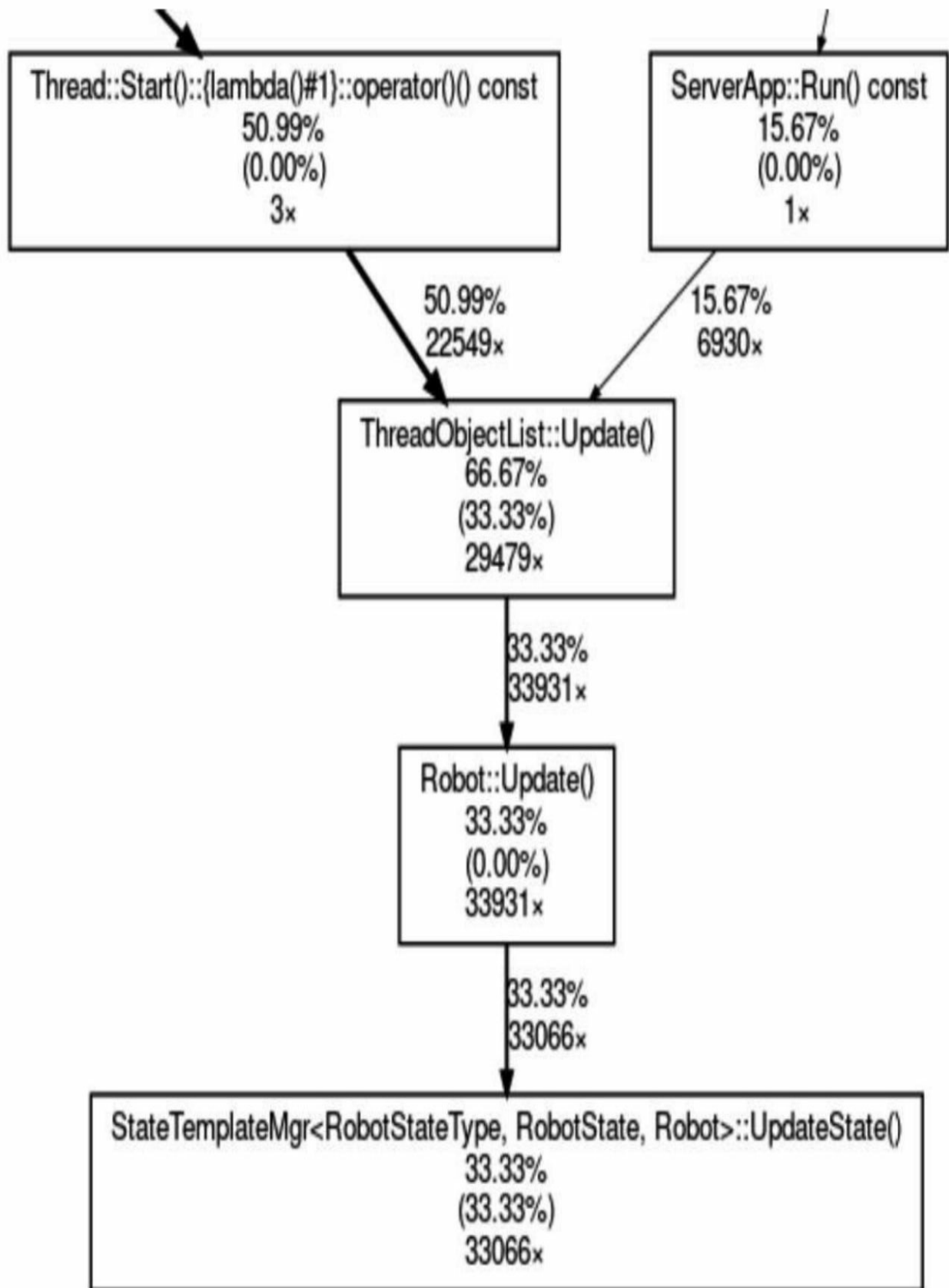


图5-9 gprof生成robots工程调用堆栈图

5.4

valgrind

在Linux上除了gprof外，还有一个内存检测工具valgrind。该工具不需要注入特殊的源代码，在编译时需要“-O0-g”参数，把代码优化等级降低即可。已下载一个较新的版本valgrind 3.15.0，该包已放入本书源代码目录dependence中了。解压编译步骤如下：

```
[root@localhost dependence]# tar xf valgrind-3.15.0.tar.bz2
[root@localhost dependence]# cd valgrind-3.15.0/
[root@localhost valgrind-3.15.0]# ./configure
[root@localhost valgrind-3.15.0]# make
[root@localhost valgrind-3.15.0]# make install
[root@localhost valgrind-3.15.0]# valgrind --version
valgrind-3.15.0
```

最后一步，查看版本号。一切就绪之后，就可以在执行程序中直接调用了。下面还是以一个简单的例子切入，参考源代码在05_04_valgrind目录中。目录中有一个文件test_valgrind.c，内容如下：

```
void f(void) {
    int* x = (int*)malloc(10 * sizeof(int));
    x[10] = 0;
}
int main(void) {
    f();
    return 0;
}
```

在这个例子中有两个问题：第一个问题是x的内存没有释放，第二个问题是x[10]越界访问。我们编译一下，具体步骤如下：

```
[root@localhost 05_04_valgrind]# cmake3 ./
```

让valgrind启动可执行文件，命令如下：

```
[root@localhost 05_04_valgrind]# valgrind ./test_vd
...
==13781== Invalid write of size 4
==13781==      at 0x400710: f() (test_valgrind.cpp:6)
==13781==      by 0x400721: main (test_valgrind.cpp:11)
==13781== Address 0x5b07ca8 is 0 bytes after a block of size 40 a
==13781==      at 0x4C29E03: malloc (vg_replace_malloc.c:309)
==13781==      by 0x400703: f() (test_valgrind.cpp:5)
==13781==      by 0x400721: main (test_valgrind.cpp:11)
...
```

从上面的打印结果中可以看出，valgrind非常明确地指出了这两处错误（包括错误所在的文件和行号），使用非常方便。在使用valgrind时默认的选项是memcheck，即内存检查，但除了内存检查之外，valgrind也提供一些其他的功能，可以通过“--tool=tool name”指定其他的工具。除了可以检查内存之外，valgrind也可以输出调用堆栈。用5.3节的gprof工程为例来对比一下两者有何差异。进入05_02_gprof目录，执行如下命令：

```
[root@localhost 05_02_gprof]# valgrind --tool=callgrind ./test_pd
```

完成之后会发现目录中多了一个名为callgrind.out.4171的文件，其中4171是进程ID，每次执行时的进程ID是不一样的，所以每次生成的文件都不一样。生成PNG堆栈图的命令如下：

```
[root@localhost 05_02_gprof]# gprof2dot -f callgrind -n10 -s callg
[root@localhost 05_02_gprof]# dot -Grankdir=LR -Tpng valgrind.dot
```

图5-10是valgrind生成的堆栈图，可以结合5.3节gprof的堆栈图一起看。同一个工程中，valgrind的输出比gprof的输出详细一些。还可以执行如下命令，让调用堆栈更详细：

```
gprof2dot -f callgrind -n0.05 -e0 -s callgrind.out.4171 > valgrind
```

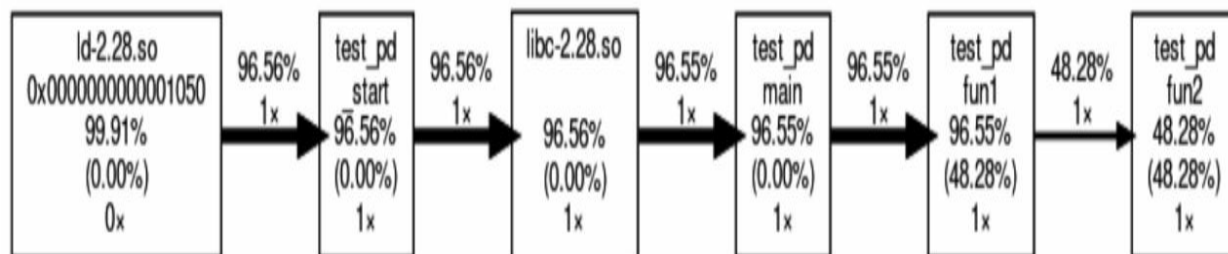


图5-10 valgrind堆栈图

该命令生成的dot节点详尽到底层库文件。生成图片的结果如图5-11所示。

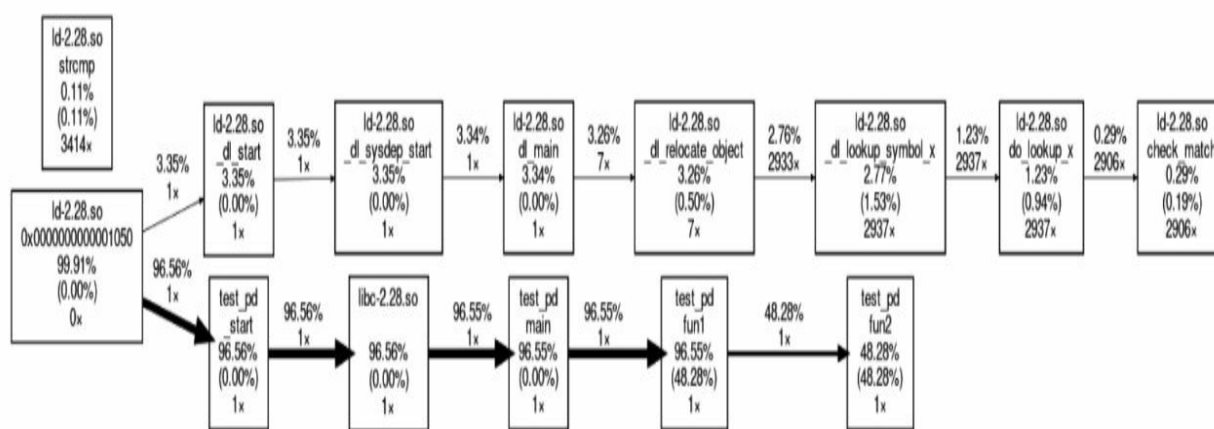


图5-11 valgrind堆栈详细图

由于纸质书页面宽度的限制，该插图不是特别清楚，读者可以在自己的计算机上生成一个详细图试试。在查看堆栈时，可以看到Linux底层的函数。关于valgrind的使用，可参阅它的官方说明。本书关于它的介绍就此结束，下面以本书的框架为例来看看valgrind工具可以做什么

么优化。

使用valgrind工具查看框架

回到项目中，打开05_03_engine_gprof工程，在做valgrind测试时，最好把CMakeLists.txt中grpof使用的“-pg”参数去掉，使测试结果更客观一些。下面来看一下登录一个账号时，robots进程产生的流程图。

```
[root@localhost bin]# valgrind --tool=callgrind ./robotstd
...
login -a test
```

退出robots进程之后，目录下生成了一个名为callgrind.out.19023的文件，19023是进程号。执行如下命令生成调用堆栈图，如图5-12所示。

```
[root@localhost bin]# gprof2dot -f callgrind -n10 -s callgrind.out
```

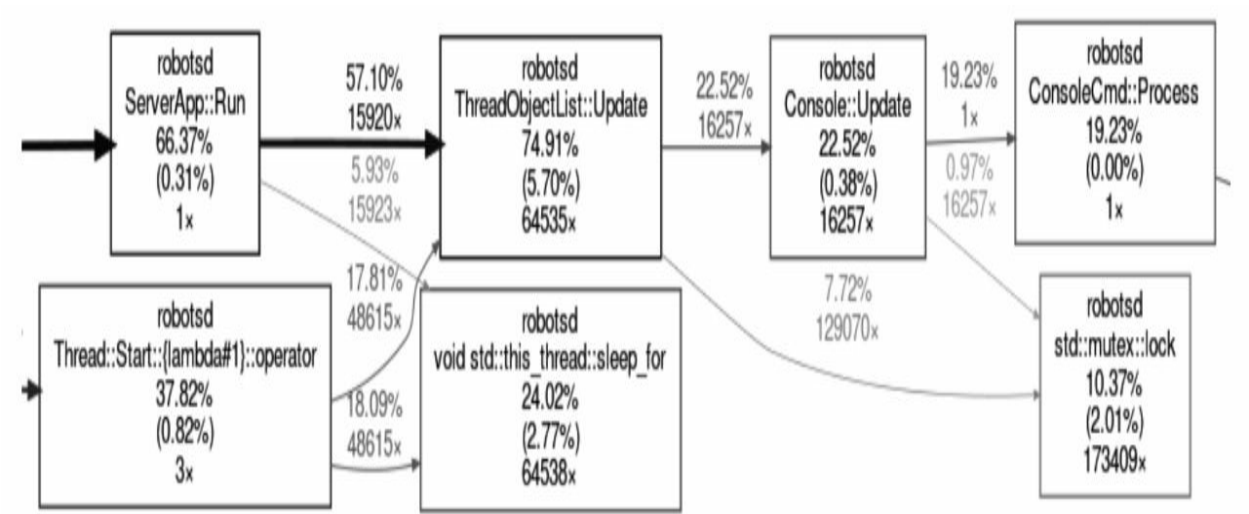


图5-12 robots堆栈图

读者可以按照实际步骤操作一下，看看生成图。因为纸质书页面宽

度的关系，图5-12可能看不太清楚，去掉一些细节，修改参数n50，新的结果如图5-13所示。

```
[root@localhost bin]# gprof2dot -f callgrind -n50 -s callgrind.out
```

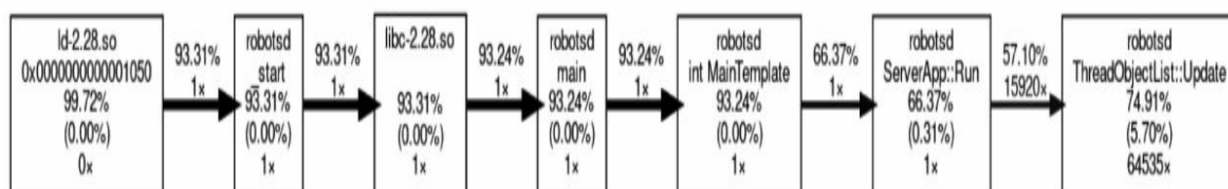


图5-13 robots堆栈图n50

图5-13显示了耗时占50%以上的所有调用，一些细枝末节的调用函数已经去掉了。-n50就是这个意思，如果我们写成-n0，就会出现一张完整的调用图，显示所有调用函数。如果认真观察就会发现，在这张调用图中只有主线程的调用，而没有子线程的调用。如果需要查看子线程的调用，在生成callgrind.out分析文件时就需要加一个特别的参数“-separate-threads”，来看看批量登录200个账号的调用堆栈。

工程login使用valgrind命令启动，命令如下：

```
[root@localhost bin]# valgrind --tool=callgrind --separate-threads
```

启动工程robots，输入“login-ex test 200”，登录完成之后，login进程打印“Login-Logined over”之后，按Ctrl+C组合键退出。查看目录，生成了6个文件：

```
-rw----- 1 root root      0 Dec 19 09:44 callgrind.out.3226
-rw----- 1 root root 807341 Dec 19 09:45 callgrind.out.3226-01
-rw----- 1 root root 460828 Dec 19 09:45 callgrind.out.3226-02
-rw----- 1 root root  24664 Dec 19 09:45 callgrind.out.3226-03
-rw----- 1 root root 227029 Dec 19 09:45 callgrind.out.3226-04
-rw----- 1 root root 227029 Dec 19 09:45 callgrind.out.3226-05
```


以数字后缀结尾的都是子线程的输出文件。为什么会有5个线程呢？

(1) 在ServerApp初始化的时候创建了3个子线程。

(2) CMD启动了一个线程。

(3) 主线程。

在工程中提供了一个Shell脚本来批量生成图片，在bin目录下，这个脚本的文件名为gen_png.sh。脚本的目的就是遍历目录下的callgrind.out文件，生成相对应的图片。执行gen_png.sh 10，让耗时占10%的函数都显示出来：

```
[root@localhost bin]# ./gen_png.sh 10
gprof2dot -f callgrind -n10 -s callgrind.out.3226 | dot -Grankdir=
gprof2dot -f callgrind -n10 -s callgrind.out.3226-01 | dot -Grankd
gprof2dot -f callgrind -n10 -s callgrind.out.3226-02 | dot -Grankd
gprof2dot -f callgrind -n10 -s callgrind.out.3226-03 | dot -Grankd
gprof2dot -f callgrind -n10 -s callgrind.out.3226-04 | dot -Grankd
gprof2dot -f callgrind -n10 -s callgrind.out.3226-05 | dot -Grankd
```

运行之后，就得到5张图，valgrind-3226.png是张空白图，因为callgrind.out.3226文件是一个空文件。下面简短说明生成图的细节。

关键点1：主线程堆栈图

在主线程中只有一个时间更新函数，所以它的调用相对简单。

在图5-14中，ThreadObjectList::Update占用时间为18%左右，而sleep_for约为14%，两者数据相当，说明主线程中一帧的执行时间大约与sleep_for的时长相差无几。

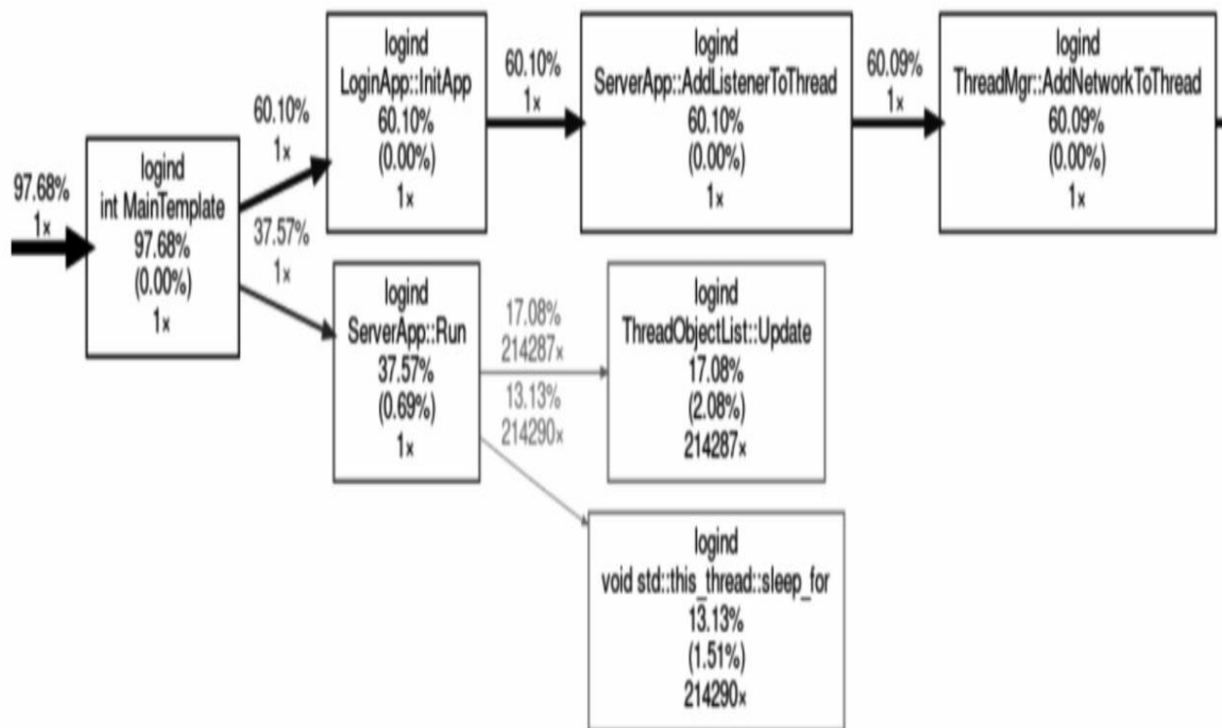


图5-14 login主线程堆栈图

关键点2：子线程堆栈图

每个子线程堆栈图都不一样，因为对象不同，调用堆栈图就不一样。图5-15是其中一个子线程堆栈图的展示图，在这个线程中，正好有HttpRequest类，可以看出其中50%以上的运行时间被HttpRequest::Update占用了，就是对外验证账号的功能占用了大部分运行的时间。值得说明的是，libcurl库的实际运行效率并不低，只是我们目前的这个程序功能太少，因而它的耗时占比被凸显了出来。

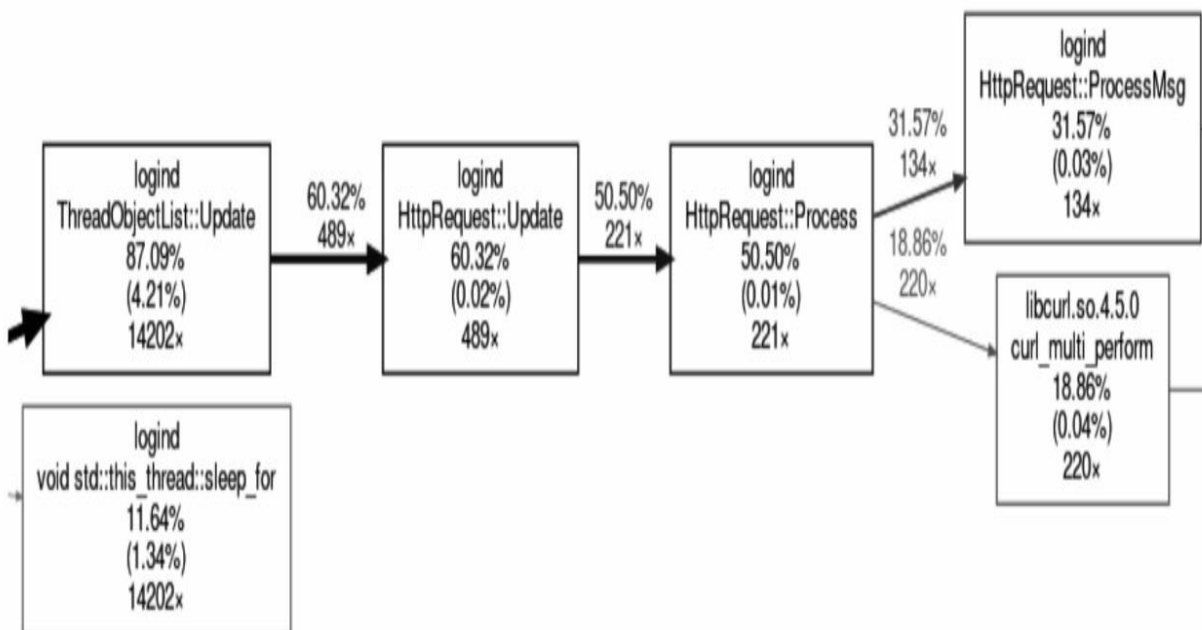


图5-15 login子线程堆栈图

更多的图就不在本书中分析了。读者可以手动查看一下这些图，从这些堆栈图中进一步加深对于框架结构的理解。

到目前为止，我们通过例子展示了如何查看代码的执行效率，Windows和Linux系统下的工具都有其独特性，有时可以结合起来一起使用。现在功能还比较少，瓶颈不是太明显。在工作中，随着功能的增多，在开发后期可以多次、反复使用这些工具来提升整体执行性能。

对象池

前面介绍了使用工具来检查性能瓶颈。工具只是我们的手段，但想要从本质上提升性能，还需要从流程或数据结构上下功夫。5.2节介绍了两个数据结构，本节将介绍另一个重要的组织数据方式——对象池。

如果读者曾手动执行过valgrind工具，可能会有这种经历，就是在执行valgrind时看到提示信息弹出来提醒我们，程序存在内存泄漏问题。用valgrind启动login工程，进入05_03_engine_gprof目录中，编译后在bin目录执行如下命令：

```
[root@localhost bin]# valgrind --tool=memcheck --leak-check=full .
```

启动robots，正常登录一个账号，在login进程中，按Ctrl+C组合键退出，这时可以看到很多打印信息，提示我们有内存泄漏问题。比如这一段：

```
==15541==      at 0x4C2A462: operator new(unsigned long) (vg_replace  
==15541==      by 0x448433: LoginObjMgr::AddPlayer(...) (login_obj_m  
==15541==      by 0x444063: Account::HandleAccountCheck(Packet*) (ac
```

每一次有新账号登录时，都会创建一个LoginObj类来保存当前玩家的登录数据，但是这个LoginObj实例用完之后没有被释放。在前面的例子中，为了突出重点逻辑，没有管理内存对象，都只是创建了对对象，使用过后都没有释放对象。本节要引入一个非常重要的概念——对象池。

那么，什么是对象池呢？

所谓池，一定是众多的。可以这样理解对象池，就是提前生成若干个对象实例并放在池中，在需要时提出，不需要时重新回到池中。对象池的好处在于不需要频繁地创建和销毁对象。

5.5.1 对象池代码分析

本节的源代码位于05_05_pool目录中，在框架中实现了对象池，原则是预先生成一定数量的对象，有需要的时候从对象池中取一个出来，用完之后再归还到对象池中。需要用时随用随取，而不是每次使用时创建对象。在框架后期几乎所有对象都从对象池中取出使用，框架使用对象池的另一个作用是对内存中的对象严密监控。下面分析一下具体的代码。

关键点1：对象池对象基类

首先为了规范所有对象池中的对象，规定每个在对象池中的对象都需要继承自ObjectBlock类。其定义在libserver工程的object_block.h文件中。

```
class ObjectBlock :virtual public SnObject, virtual public IDispos
public:
    ObjectBlock(IDynamicObjectPool* pPool);
    virtual void Dispose() override;
    virtual void BackToPool() = 0;
protected:
    IDynamicObjectPool* _pPool{ nullptr };
};
```

从定义中可以看到每个想放入对象池的对象需要自己实现一个虚函数BackToPool。不再需要该对象、归还该对象时调用该函数。该函数的作用是清空对象内的内存数据，而清空的具体数据由对象自己控制。

关键点2：对象池类

对象池的类在object_pool.h文件中，定义如下：

```
template <typename T>
class DynamicObjectPool :public IDynamicObjectPool {
public:
    template<typename ...Targs>
```

```

    T* MallocObject(Targs... args);
    void Update() override;
    void FreeObject(ObjectBlock* pObj) override;
    ...
private:
    std::mutex _freeLock;
    std::queue<T*> _free;
    std::mutex _inUseLock;
    CacheRefresh<T> _objInUse;
};

```

把细枝末节的函数和属性去掉，看看它主要的属性和函数。在对象池中管理着两个集合：其中一个是队列free，作用是存放对象池未使用的对象；另一个是列表objInUse，存放对象池正在使用的对象。对象池中未使用的对象集合free的类型为std::queue，因为是全局数据，所以每个线程都可能访问该数据，操作该数据时需要加锁。已使用对象则是由CacheRefresh模板来维护的。模板CacheRefresh中有3个队列：Add、Remove和Reader。当一个对象进入Add队列中时，说明它被唤醒了。而一个对象进入Remove队列中，说明它的生命周期已经结束，可以回收了。

对象池中的两个重要函数：MallocObject和FreeObject。MallocObject函数分配一个对象，这个函数是一个变量函数。原因在于，每一个对象从对象池中唤醒时，需要传入的参数不同。而FreeObject函数是在对象用完之后归还到对象池中时调用。两个函数分别用来分配和释放对象。

图5-16展示了ConnectObj对象生成与销毁时，调用这两个函数与对象池中两个集合之间的关系。

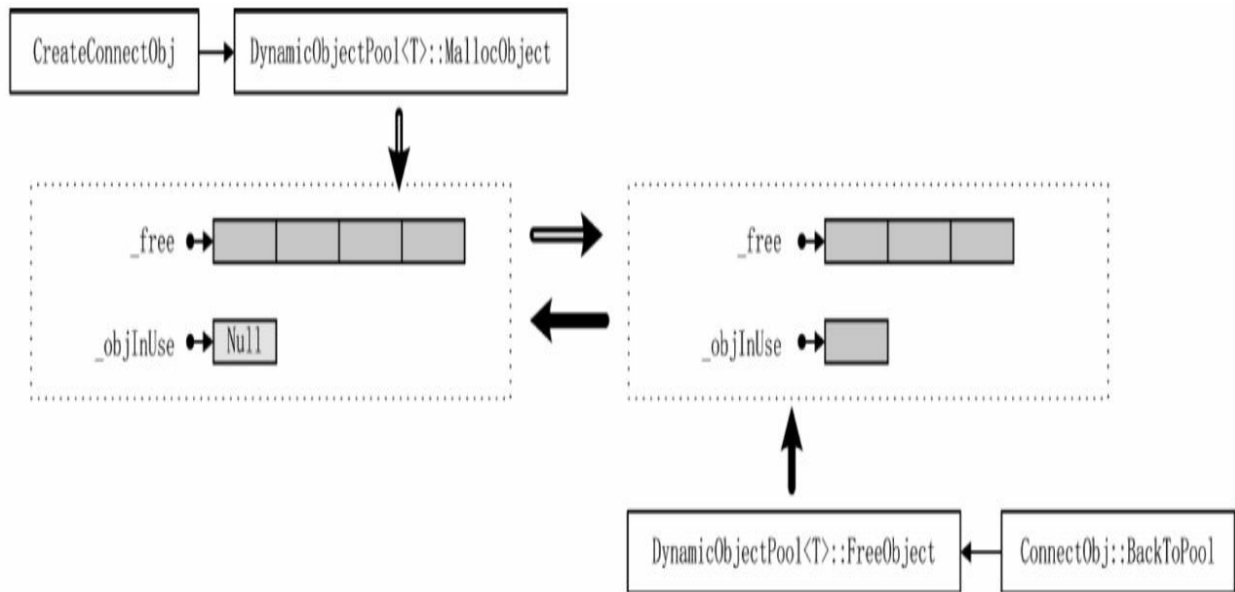


图5-16 对象池的内存对象

图5-16中的灰色表示一个可使用的对象。从对象池中取出一个对象时，调用`MallocObject`函数，就从`free`队列中拿走一块数据，该对象放入`objInUse`管理。对象用完之后，调用`FreeObject`函数再次回到对象池中。

关键点3：从对象池中创建对象

当需要从对象池中获取一个对象时，调用`MallocObject`函数，得到一个没有使用的对象，然后进行初始化。函数`MallocObject`的实现如下：

```
template <typename T>
template <typename ... Targs>
T* DynamicObjectPool<T>::MallocObject(Targs... args) {
    _freeLock.lock();
    if (_free.size() == 0) {
        CreateOne();
    }
    auto pObj = _free.front();
    _free.pop();
    _freeLock.unlock();
    pObj->ResetSN();
}
```



```
pObj->TakeoutFromPool(std::forward<Targs>(args)...);
_inUseLock.lock();
_objInUse.GetAddCache()->push_back(pObj);
_inUseLock.unlock();
return pObj;
```

缓存不够时调用CreateOne函数，增加一个实例，也可以批量增加。将空闲的对象从队列free中取出，加入objInUse集合中。对取出的对象做了两件事情：

```
pObj->ResetSN();
pObj->TakeoutFromPool(std::forward<Targs>(args)...);
```

函数ResetSN将对象的SN做了重置。另一个函数TakeoutFromPool是一个变参函数。每一种对象需要的初始数据不同，无法给一个定值，这里使用了变参。这个TakeoutFromPool函数可能是TakeoutFromPool(int, int)，也可能是TakeoutFromPool(std::string)，这是由于不同的类有不同的处理方式。下面来看一个实例，在本例中，我们对ConnectObj进行了对象池的处理。

```
class ConnectObj : public ObjectBlock {
...
void TakeoutFromPool(Network* pNetWork, SOCKET socket);
virtual void BackToPool() override;
}
```

作为对象池对象，ConnectObj类需要实现TakeoutFromPool函数和BackToPool函数，一个是将对象从池中带走时进行初始化，另一个是回归到池中之前进行清理工作。整理一下思路：

(1) 首先ConnectObj类需要从ObjectBlock类继承，并实现两个函数，用于进出对象池时的操作。

(2) 修改ConnectObj被创建的代码，从单纯的创建操作变成从对象池中获取。其代码如下：

```
void Network::CreateConnectObj(SOCKET socket) {
    auto pConnectObj = DynamicObjectPool<ConnectObj>::GetInstance(
        ...
    }
}
```

(3) 当要销毁一个ConnectObj时，不再对其进行Delete操作，而是进行Dispose操作。ConnectObj类没有重载Dispose函数，调用的是基类的ObjectBlock::Dispose函数，其实现代码如下：

```
void ObjectBlock::Dispose() {
    BackToPool();
}
```

ObjectBlock调用了一个虚函数BackToPool，ConnectObj对这个虚函数的实现如下：

```
void ConnectObj::BackToPool() {
    if (!Global::GetInstance()->IsStop) {
        // 通知其他对象有Socket中断了
        Packet* pResultPacket = new Packet(Proto::MsgId::MI_Networ
        MessageList::DispatchPacket(pResultPacket);
    }

    _pNetWork = nullptr;
    _socket = INVALID_SOCKET;
    _recvBuffer->BackToPool();
    _sendBuffer->BackToPool();
    _pPool->FreeObject(this);
}
```

每个类对于BackToPool函数的实现都不相同，在ConnectObj类中，在这个函数中做了3件事：

(1) 当一个网络层对象关闭时，向整个框架发送了一条MI_NetworkDisconnect协议，向系统广播有一个Socket断线了。至于这个协议是由谁去处理的，网络层对象并不关心。

(2) 将接收和发送缓存数据重置，注意这两个缓存是被创建出来

的，但是这里并没有释放缓存数据，而只是重置了缓存位置。

(3) 调用对象池的FreeObject函数，告诉对象池可以将其回收了。

关键点4：回收对象

在对象不再使用之后，将它放到objInUse对象的Remove队列中，后续的处理是在对象池的Update中进行的。也就是说，在回收对象时，并不是马上就回收，而是会等待一帧，收集在这一帧所有需要回收的对象，等到下一帧统一进行回收，数据统一处理可以节省一些开销。回收处理在对象池管理模板类帧函数中，实现如下：

```
template <typename T>
void DynamicObjectPool<T>::Update() {
    std::list<T*> freeObjs;
    _inUseLock.lock();
    if (_objInUse.CanSwap()) {
        freeObjs = _objInUse.Swap();
    }
    _inUseLock.unlock();
    std::lock_guard<std::mutex> guard(_freeLock);
    for (auto one : freeObjs) {
        _free.push(one);
    }
}
```

当objInUse对象的Remove队列中有数据时，必然会触发objInUse.Swap函数，这个函数将新生成的对象加入Reader队列中，同时将Remove队列中的数据从Reader队列中删除。该函数返回的数据就是被删除的数据队列，再将这些已经不用的数据重新放回到对象池的free队列中，等待下次使用。

DynamicObjectPool<T>::Update函数是在主线程的ServerApp::Run函数中被触发的。这里不再展示代码，有兴趣的读者可以直接查看工程的源代码。

5.5.2 使用cmd命令查看对象池

在05_05_pool工程中新增了一条指令查看当前的对象池状态，输入“pool-show”就可以查看ConnectObj对象池中现有多少数据。当login进程接收完300个登录请求之后，输入“pool-show”指令，控制台输出如下：

```
[root@localhost bin]# ./logind
pool -show
*****
pool total count: 201
free count:      0
in use count:    201
total call:      201
```

当前对象池中一共有201个对象，使用中的对象为201个，空闲对象是0个。total call意味着程序调用对象池的总次数。在robots进程中创建了200个Robot类，这里为什么会有201个ConnectObj对象呢？因为RobotMgr也创建了一个连接。使用Ctrl+C组合键关闭Robots进程之后，再来看一下对象池的情况，login进程控制台输出如下：

```
pool total count: 201
free count:      201
in use count:    0
```

可以看到，201个对象全部回到对象池中，使用中的对象为0个。pool指令的工作流程这里不再详细介绍，有兴趣的读者可以查看源代码，其定义在console_cmd_pool.h文件中。

05_05_pool工程并不是一个完美的工程，有关Packet的内存泄漏并没有处理。以ConnectObj的使用来简要说明对象池的使用方式，除了ConnectObj之外，在后面的框架中，将会对框架中的对象启用对象池，以方便管理和查看对象，同时方便查看是否有内存泄漏的情况。

5.6

总结

本章使用工具查看框架的性能，引入了提高性能的数据结构和对象池。对于工具讲得比较简略，作为工具来讲，需要在使用过程中不断累积经验。本章我们放慢了编码速度，优化了数据与结构，是为了第6章要引入的全新概念打下基础，那就是ECS框架。

第6章

搭建ECS框架

在之前的章节中，我们已经大致了解了服务器在多线程下的工作流程，引入了一些可以提高效率的数据结构，也加入了一个简单的对象池，虽然没有将游戏的整个过程编写完整，但基于目前的框架完全可以开发出一个多进程、多线程的游戏。

简单来说，不论是多线程还是单线程，不断更新逻辑，对每个对象执行Update操作，是目前比较常用的一种框架体系。但这个框架中有一些问题，例如需要每一个对象都继承自ThreadObject类，如果功能复杂，就可能出现虚继承的情况。代码越写越多，其复杂度越来越高，继承的层数也会变得更深，这给编码带来了不小的麻烦。

在此基础上，本章引入一个新的架构思路——ECS框架，从ECS的基本概念开始，在改动极少的情况下完成一次框架重构。本章包括以下内容：

- 了解什么是ECS。
- 重构框架，引入ECS的设计思路，并与线程结合起来。
- 增加服务端配置文件。
- 增加服务端日志功能。

一个简单的ECS工程

ECS的全称为Entity Component System, Entity是实体, Component是组件, System则指系统。Entity中可以包括无数个Component, 具备某组件便具有某功能, 而System是控制这些组件的动作。

ECS和我们之前的框架有什么区别呢? 我们用图6-1来详细说明这一区别。在之前的框架中, 线程或者进程上会存在很多Object (对象), 这些Object都是继承自ThreadObject, 这带来了一个问题, 并不是所有在线程上存在的实例都需要Update帧操作, 例如login工程中的Account类虽然继承了ThreadObject, 也实现了Update函数, 但这个函数是空的。继承自ThreadObject的类都需要实现Update函数, 所以这里只能写一个空函数。

为了解决空函数的问题, 提出了一个System框架, 在System中定义了几种类型的动作。例如InitializeSystem是初始化动作, UpdateSystem是更新系统, 每一个动作都是一个接口。这意味着一个对象可以实现按需定义, 如果需要初始化就实现InitializeSystem接口, 如果需要Update就实现UpdateSystem接口。

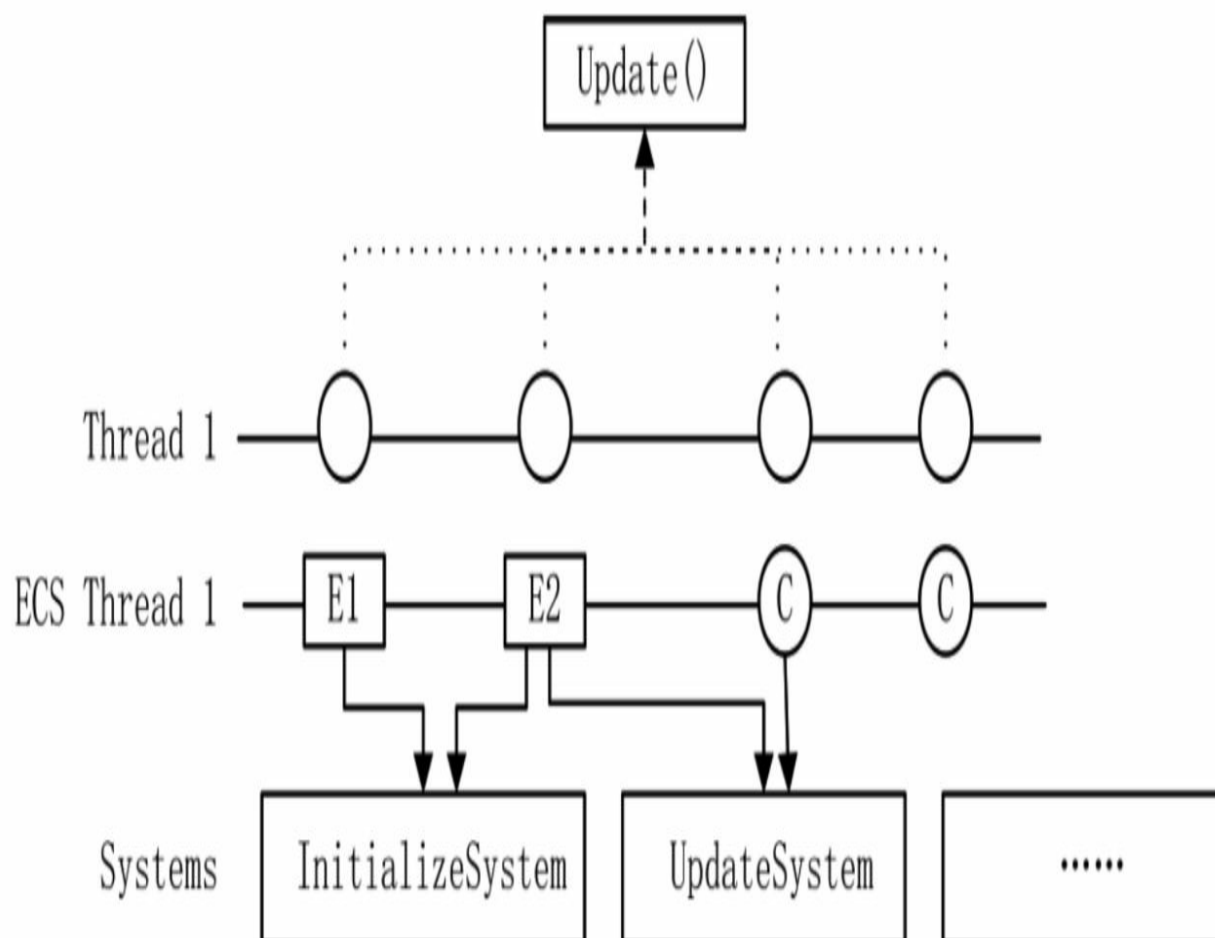


图6-1 旧框架与ECS框架对比

以图6-1为例，实体E1实现了InitializeSystem动作，而E2同时实现了InitializeSystem和UpdateSystem需要的接口，组件C仅实现了UpdateSystem需要的接口。对于UpdateSystem来说，它需要更新的只有E2和C对象。

优化后的框架中有一系列实体，每个实体都可以根据自己的需求来选择，以是否有这个动作来决定是否实现这些接口。

需要说明的是，这个解决方案并不是ECS的最终模型，但是可以从这个思路开始理解ECS，从面向对象过渡到组件式编程。本章中，我们重构的框架是一种折中方案，以便读者可以快速了解ECS中的实体与组件。在后面的章节会逐步重构当前框架，重构是一个循序渐进的过程，这个过程并不能一步到位。在每个例子中，为了方便读者由浅入深地理

解ECS框架的原理，理解这些实体、组件和系统的作用，会尽量讲述“它为什么会变成这样”的过程，即编程思路。

在将ECS引入框架之前，先用一个简单的ECS框架例子来认识一下它。

本节的源代码位于06_01_ecs目录中。

6.1.1 组件类Component

组件类Component定义在component.h文件中，其定义如下：

```
class Component {
public:
    virtual ~Component() = default;
    virtual void Dispose() = 0;
    void SetParent(Entity* pObj);
    Entity* GetParent() const;
    long GetSN() const;
    void SetSN(const long sn);
private:
    Entity* _parent{ nullptr };
    long _sn{ 0 };
};
```

作为一个基类，Component的数据非常少，每个组件都有一个唯一标识SN和一个父对象指针，这个指针并不一定赋值，也就是说组件也有可能没有父类。组件的父类一定是一个实体，组件不能成为组件的父类，一个类有了组件则一定是实体。

6.1.2 实体类Entity

实体类Entity定义在entity.h文件中，其定义如下：

```
class Entity :public Component {
public:
    void Dispose() override;
    void AddComponent(Component* obj);
    template<class T>
    T* GetComponent();
protected:
    std::map<long, Component*> _components;
};
```

在Entity类中实现了一个增加组件的函数AddComponent，它将一个组件绑定到Entity中。Entity本身也是一个Component，也就是说它可以作为其他Entity的组件。

那么，问题来了，如何判断一个类该定义为组件还是实体呢？简单来说，如果一个类中还需要有组件，它就是一个实体，如果一个类中不需要再有更小的组件，它就是一个基本的组件。

6.1.3 系统类System

最后，我们定义System的接口，其定义在system.h文件中：

```
class ISystem {
protected:
    ISystem() = default;
public:
    virtual ~ISystem() = default;
};
class IInitializeSystem : virtual public ISystem {
protected:
    IInitializeSystem() = default;
public:
    virtual ~IInitializeSystem() = default;
    virtual void Initialize() = 0;
};
class IUpdateSystem : virtual public ISystem {
protected:
    IUpdateSystem() = default;
public:
    virtual ~IUpdateSystem() = default;
    virtual void Update() = 0;
};
```

这里定义了两个系统，一个是IInitializeSystem，另一个是IUpdateSystem，它们分别实现初始化和更新操作。这两个系统的功能依然采用继承方式。

6.1.4 管理类EntitySystem

为了维护Entity和Component类，需要一个管理类EntitySystem，其定义在entity_system.h文件中，定义如下：

```
class EntitySystem {
public:
    template <class T, typename... TArgs>
    T* CreateComponent(long sn, TArgs&&... args);
    virtual bool Update();
    void Dispose();
protected:
    std::list<IInitializeSystem*> _initializeSystems;
    std::list<IUpdateSystem*> _updateSystems;
};
```

在EntitySystem类中提供了CreateComponent函数来创建一个组件或实体，因为每一个类的构造函数参数不同，所以这是一个多参的创建函数，其实现代码如下：

```
template<class T, typename ... TArgs>
T* EntitySystem::CreateComponent(TArgs&& ... args) {
    std::cout << "create obj:" << typeid(T).name() << std::endl;
    auto component = new T(std::forward<TArgs>(args)...);
    const auto objInit = dynamic_cast<IInitializeSystem*>(component)
    if (objInit != nullptr) {
        _initializeSystems.push_back(objInit);
        return component;
    }
    // 有些组件可能不需要 IInitializeSystem, 直接进入UpdateSystem
    const auto objUpdate = dynamic_cast<IUpdateSystem*>(component)
    if (objUpdate != nullptr) {
        _updateSystems.push_back(objUpdate);
        return component;
    }
    return component;
}
```

在上面的代码中，新建组件之后，判断它属于哪一个系统，如果是

InitializeSystem，就加到维护InitializeSystem对象的集合中。如果一个组件进入InitializeSystem队列中，就不再进入其他队列中了，只有初始化完成之后它才进入后继的系统中，后续的功能就交给帧函数EntitySystem::Update。其实现如下：

```
bool EntitySystem::Update() {
    // 初始化
    while (_initializeSystems.size() > 0) {
        auto pComponent = _initializeSystems.front();
        pComponent->Initialize();
        _initializeSystems.pop_front();

        const auto objUpdate = dynamic_cast<IUpdateSystem*>(pComponent);
        if (objUpdate != nullptr)
            _updateSystems.push_back(objUpdate);
    }
    for (auto& iter : _updateSystems) {
        iter->Update();
    }
    return true;
}
```

函数EntitySystem::Update会查看初始化列表中是否有数据，如果有就进行处理。初始化完成之后，对象如果有IUpdateSystem接口，就加入UpdateSystem集合中。一定要先初始化之后再进入UpdateSystem集合中。在初始化中，我们可以检查依赖项是否存在，只有当所有条件都准备好了，一个组件或实体才被认为初始化成功。

原因很简单，如果有两个类A和B都实现了UpdateSystem接口，在A类的Update函数中，我们需要用到B类的对象，伪代码如下：

```
A::Update() {
    auto pB = GetComponent<B>();
    ...
}
```

如果没有经过初始化的依赖判断，在执行上面的代码时，有可能B类的对象并不存在。使用一个变通的方式来解决这个问题，伪代码如下：

```
A::Update() {  
    If (!HasComponet<B>()) {  
        Auto pB = GetComponet<B>();  
        ...  
    }  
}
```

如果一个Update函数中每次都要判断B类的对象是否存在，显然是不合理的。因此，如果有依赖，就先在初始化的时候处理。

6.1.5 测试

清楚了几个大类的定义与职责，执行本小节中的例子，以便更清楚地了解各类之间是如何配合工作的。在工程06_01_ecs目录中，main函数的定义如下：

```
int main() {
    EntitySystem eSys;
    auto pEntity1 = eSys.CreateComponent<TestEntityWithInitAndUpdate>();
    auto pEntity2 = eSys.CreateComponent<TestEntityWithUpdate>();

    const auto pCInit = eSys.CreateComponent<TestCInit>();
    pEntity1->AddComponent(pCInit);

    const auto pCUpdate = eSys.CreateComponent<TestCUpdate>();
    pEntity2->AddComponent(pCUpdate);

    while (true) {
        eSys.Update();
    }
}
```

在主函数中创建了两个实例：TestEntityWithInitAndUpdate和TestEntityWithUpdate。与它们的名字一样，前一个是继承了两个接口的实体，后一个是继承了Update接口的实体。同时，在这两个实体上分别创建了组件TestCInit和组件TestCUpdate。

先来看一下程序结果，编译一下：

```
[root@localhost 06_01_ecs]# cd ecs/
[root@localhost ecs]# cmake3 ./
[root@localhost ecs]# make
[root@localhost ecs]# ./ecsd
create obj:27TestEntityWithInitAndUpdate           // 创建类
create obj:20TestEntityWithUpdate                   // 创建类
create obj:9TestCInit                               // 创建类
create obj:11TestCUpdate                             // 创建类
P27TestEntityWithInitAndUpdate::Initialize           // 执行了初始化函数
P9TestCInit::Initialize                             // 执行了
P20TestEntityWithUpdate::Update                     // 执行了更
```

```
P11TestCUpdate::Update // 执行了更
P27TestEntityWithInitAndUpdate::Update // 执行了更新函
```

在Linux下，类的名字前面有一些随机字符，可以忽略掉。所有的Entity和Component的生成都是由EntitySystem来管理的。组件和实体是相互独立的，但是因为我们执行了AddComponent，组件和实体又是可以相互访问的。按照面向对象框架的做法，如果TestCUpdate类需要执行Update，就需要由它的父类来调用，但是现在这层逻辑变得扁平化，是统一由EntitySystem来执行的，而它的父类可能连Update事件也没有，降低了实体与其组件之间的复杂度。

这个简单的例子也许和我们最终的框架相差甚远，但在这个例子中，我们看到了实体、组件与系统的关系，组件依附于实体，但却不依赖于它。同时，在本例中还做了一个简单的System，这个System中有两个事件：一个是初始化，另一个是更新。下面我们来看看如何将这些内容引入之前的框架中。

基于ECS框架的libserver

ECS框架是一个整体，如果在本书之前的多线程框架中加入一个ECS框架就必须加锁，加锁会引起工作量和效率的连锁反应，所以要将ECS框架思路引入线程中，就需要每一个线程都有一个ECS架构，也就是说每个线程中都有一个EntitySystem来管理线程中的对象。图6-2是对象与EntitySystem的分布图。

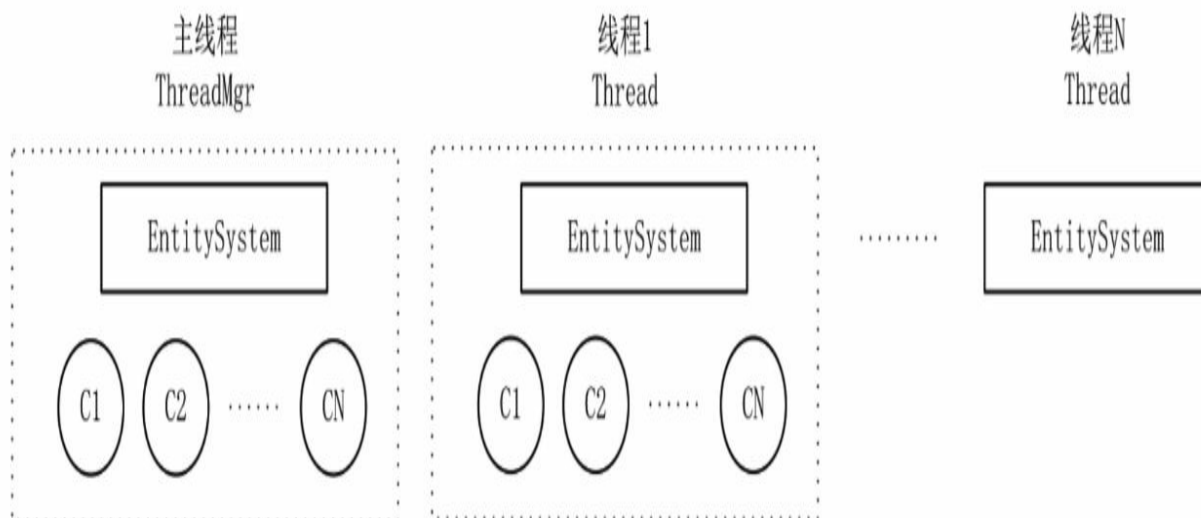


图6-2 对象与EntitySystem的分布图

每个子线程都是ECS架构，主线程也是一个ECS架构。

主线程中的EntitySystem用来管理全局对象。所谓全局数据，是生成之后一般不会改变的数据。就像后面会加入的游戏逻辑配置读取、服务器配置读取这种全局数据，放到主线程中。这些数据有一个特点，一

旦生成就不会发生改变，不存在线程冲突，不需要加锁，是可以共享给其他线程使用的。要把旧的libserver修改为如上框架，需要修改的功能点如下：

（1）现在我们不再需要基类ThreadObject，而由Entity或者Component来替代它。

（2）线程管理类ThreadMgr作为一个主线程的对象，它除了管理线程之外，还需要继承EntitySystem类，在主线程中也有全局的Entity或者Component类需要管理。

（3）线程类Thread需要集合EntitySystem类的功能，Thread管理std::thread对象，而基类EntitySystem打理线程中的对象。

（4）增加一些基础的System类，例如UpdateSystem和MessageSystem，用于更新和处理消息。

在新的ECS框架中，生成对象是由EntitySystem对象统一来生成的，生成对象的同时需要对这些对象的特征进行分析，放置到不同的系统中，新对象是否实现了IMessageSystem接口决定了它是否需要进行消息处理，实现了IUpdateSystem接口决定了它需要每帧更新。

6.2.1 通过字符串动态创建类

在开始重构前需要解决几个棘手的问题，其中一个就是类的动态创建。回顾一下在login工程中生成线程包裹类的流程。以Account类为例，代码如下：

```
void LoginApp::InitApp() {  
    ...  
    Account* pAccount = new Account();  
    _pThreadMgr->AddObjToThread(pAccount);  
}
```

上面的代码中，先是新建一个Account对象，然后将这个对象放到某个线程中去。现在这种方式不再适合了。

因为在新框架中有了EntitySystem，用于维护和创建对象，所有的类都在这里被创建和销毁，以方便管理。首先，我们不希望这条规则被破坏，其次，如果需要跨线程创建对象，就存在加锁的问题。实际上，跨线程创建的对象非常有限，为了一些有限的类，每帧都要加锁，显然是一个不值得的事情。

因此，我们要解决一个问题，如何根据一个字符串（类名）创建一个类。如果解决了这个问题，就可以向指定的线程发送一个类名，由线程中的EntitySystem自己去创建这个类，这样就不会存在加锁的问题。图6-3对比了这两种方式的不同。

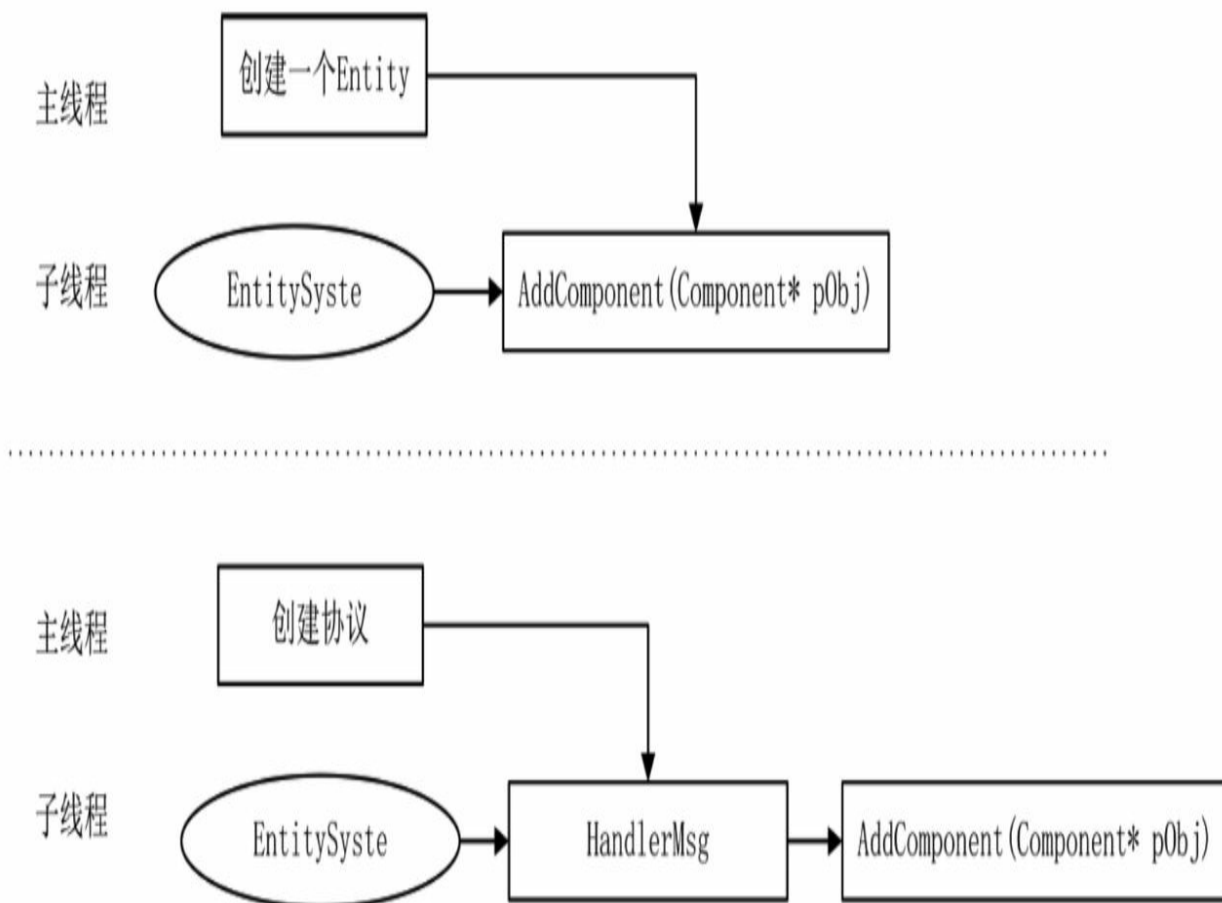


图6-3 创建Entity

本书整个框架是基于Actor模式的，实例只接收消息，不开放任何其他函数给外部调用，所以有一个最优选择——当需要创建对象时，依然采用消息的方式。

如果直接加锁修改EntitySystem中的对象，产生的后果就是在EntitySystem相关的地方都需要加锁。使用消息的方式，EntitySystem类中不用加锁，因为对象的增加、删除操作都在Update函数中进行。因此，通过协议创建类就显得十分必要。分析一下协议创建类的流程，有几个重点需要进行突破：

(1) 注册。用类名进行注册绑定，否则程序如何知道传入“classname”是生成Class1对象，而不是生成Class2对象。

(2) 写一个生成类的协议。有了注册信息后，就可以根据协议传

入的数据生成一个对象。下面来看看在C++中如何实现它。为了便于理解，将测试生成这一部分的代码独立出来，单独放在06_02_class_factory目录中。

关键点1：准备工作

在测试工程中，准备两个需要动态生成的类，定义在文件Test1.h中，名为Test1和Test2，其定义如下：

```
class Test1 : public Component {
public:
    Test1( const std::string p1 ) {
        std::cout << "create test1. p1:"<< p1.c_str( ) <<std::endl
    }
};

class Test2 : public Component {
public:
    Test2( const int p1 ) {
        std::cout << "create test2. p2:"<< p1<<std::endl;
    }
};
```

两个类都继承自组件Component类，Test1和Test2的构造函数的参数完全不同。在两个类的构造函数中都写了打印信息，便于区分在执行中的数据是否正常。

关键点2：类工厂

ComponentFactory类定义在ComponentFactory.h文件中。类工厂的作用是为需要生成的类注册其生成函数，并在适当的时机进行调用以生成实例。这是一个模板类，定义如下：

```
template<typename ...Targs>
class ComponentFactory {
public:
    bool Regist( const std::string& strTypeName, std::function<Com
    Component* Create( const std::string& strTypeName, Targs&&...
```

```
private:
    std::unordered_map<std::string, std::function<Component*( Targ
};
```

ComponentFactory类提供了Regist和Create两个函数，分别用于注册和创建。Regist注册函数的代码如下：

```
template<typename ...Targs>
bool ComponentFactory<Targs...>::Regist(const std::string& strType
    if (nullptr == pFunc) {
        return (false);
    }
    const bool bReg = _map.insert(std::make_pair(strTypeName, pFun
    return (bReg);
}
```

Regist函数传入一个字符串与一个std::function参数，并将它们一对一保存到字典中，字符串即类名，std::function函数则是这个类的生成函数。对于每一个注册的类，有一个专门的生成函数。

在类工厂中进行注册之后，需要创建时，调用Create函数，同时传入创建参数，就可以生成对应的实例。Create创建函数的代码如下：

```
template<typename ...Targs>
Component* ComponentFactory<Targs...>::Create(const std::string& s
    auto iter = _map.find(strTypeName);
    if (iter == _map.end()) {
        return (nullptr);
    } else {
        return (iter->second(std::forward<Targs>(args)...));
    }
}
```

创建时，直接调用了与类名对应的函数，那么这个函数是如何产生的呢？

关键点3：注册类

了解了工厂创建类，再来看一下注册类RegistCreator。该类的作用是为每一个类提供一个静态创建函数，并在工厂类中注册。其定义在RegistCreator.h文件中。

```
template<typename T, typename...Targs>
class RegistCreator {
public:
    RegistCreator() {
        std::string strTypeName = typeid(T).name();
        ComponentFactory<Targs...>::Instance()->Regist(strTypeName)
    }
    static T* CreateObject(Targs&& ... args) {
        return new T(std::forward<Targs>(args)...);
    }
};
```

注册类看上去没有几行，却非常重要。RegistCreator是一个模板类，该模板类有两个参数类型：一个是typename T，是需要注册的类的类型；另一个是变参，是该类的构造函数参数。因为完全不知道这些类的构造函数是什么样的，所以采用变参的方式。

RegistCreator的构造函数中调用typeid函数取得类的类名，调用类工厂进行注册。每个不同的模板都有一个静态类CreateObject，这个函数就是该类的创建函数。在向类工厂进行注册的时候，向ComponentFactory<Targs...>传入了当前CreateObject函数实例。

上面的代码中，ComponentFactory<Targs...>::Instance()这个实例就是按参数类型生成的实例。初次看这段代码，读者也许有一点疑惑。一般来说，编写类模板函数的时候，采用的方式多半是以类型为参数的模板，即ComponentFactory<Class>::Instance()。但在上面的代码中，类工厂却是采用按参数生成实例的方式，目的是减少实例生成的个数。

关键点4：使用

打开ClassFactory.cpp文件，查看main函数：

```

template<typename ...Targs>
Component* CreateComponenttr( const std::string& strTypeName, Targs
    Component* p = ComponentFactory<Targs...>::Instance( )-> Creat
    return( p );
}
int main( ) {
    // 声明Test1、Test2的创建方式
    RegisterCreator<Test1, std::string>( );
    RegisterCreator<Test2, int>( );
    // 创建
    CreateComponenttr( typeid( Test1 ).name( ), std::string( "Test1
    CreateComponenttr( typeid( Test2 ).name( ), 2 );
    return 0;
}

```

在main函数中，我们先调用RegisterCreator注册了两个类：一个类为Test1，参数为std::string类型；另一个类为Test2，参数为int类型。随后通过调用模板CreateComponenttr传入不同的参数，生成类实例对象。执行结果如下：

```

create test1. p1:Test1
create test2. P2:2

```

关键点5：内存数据分析

在这段程序中大量使用了模板，代码虽少，生成的类却不少，梳理一下以上程序的内存数据：

(1) RegisterCreator类，生成了两个实例，一个是RegisterCreator<Test1, std::string>，另一个是RegisterCreator<Test2, int>。

(2) ComponentFactory类，也生成了两个实例类，一个是ComponentFactory<std::string>，另一个是ComponentFactory<int>。

在main函数中，调用函数ComponentFactory<std::string>::Create时，根据传入的类回调到了RegisterCreator的CreateObject函数

数，生成了Test1。修改程序，增加一个Test3类，该类的构造函数与Test1一致。假设调用如下：

```
RegistCreator<Test1, std::string>( );
RegistCreator<Test2, int>( );
RegistCreator<Test3, std::string>( );
CreateComponenttr( typeid( Test1 ).name( ), std::string( "Test1" ) );
CreateComponenttr( typeid( Test2 ).name( ), 2 );
CreateComponenttr( typeid( Test3 ).name( ), std::string( "Test3" ) );
```

再来看一下内存实例：

(1) RegistCreator类，生成了3个实例类，一个是RegistCreator<Test1, std::string>，另两个是RegistCreator<Test2, int>和RegistCreator<Test3, std::string>。

(2) ComponentFactory类，依然生成了两个实例类，一个是ComponentFactory<std::string>，另一个是ComponentFactory<int>。

当调用ComponentFactory<std::string>::Create时，根据传入的类型名回调到了RegistCreator<Test1, std::string>的CreateObject函数，生成了Test1。Test3的调用也是通过ComponentFactory<std::string>::Create这个实例实现的，只是回调函数不一样，它回调的是RegistCreator<Test3, std::string>的CreateObject函数。图6-4展示了工厂实例中的内存结构。

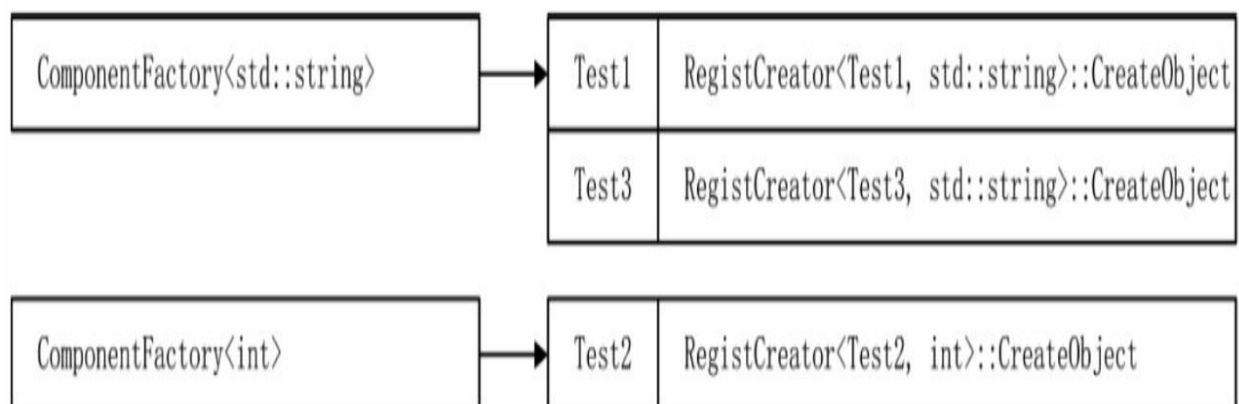


图6-4 类工厂实例

总结一下，要用字符串创建类，先要有一个类的创建函数，并注册到ComponentFactory实例类。每个工厂ComponentFactory实例类中都有类名与创建函数的对应字典。另一种方式可以避开RegistCreator类直接向ComponentFactory注册。

之前的调用方式为：

```
RegistCreator<Test1, std::string>();
```

现在变为：

```
ComponentFactory<std::string>::Instance()->Regist(typeid(Test1).na
```

后一种调用要手动编写一个静态创建函数Test1：：
CreateObject，每一个需要动态创建的类都要编写一个静态创建类。手动编写创建函数的方式内存中会少一个RegistCreator类，但需要多编写一个CreateObject函数，两种方式各有利弊。为了少编写一个函数，本书中采用第一种动态创建方式。

6.2.2 提供多参变量来创建实例

解决了动态创建类的问题，还需要解决一个问题。真正在框架中创建类的时候，是不可能像6.2.1小节的CreateComponentr函数一样使用定参的。也就是说，类似CreateComponentr (typeid (Test1).name (), std::string ("Test1"))这样的调用方式行不通。关于这一点，本书将采用动态传参的方式来实现。

先来看创建类的protobuf消息定义，在msg.proto文件中：

```
message CreateComponentParam {
    enum ParamType {
        Int = 0;
        String = 1;
    }
    ParamType type = 1;
    int32 int_param = 2;
    string string_param = 3;
}
message CreateComponent {
    string class_name = 1;
    repeated CreateComponentParam params = 2;
}
```

参数是通过结构CreateComponentParam传递过来的，需要生成的类多种多样。事先并不确定它到底有多少个参数，也不知道它是什么类型的，即使知道有多少个，参数中有int类型的，也有string类型的，它们的顺序也不能明确。鉴于此，我们必须编写一个模板函数来处理才行。现在需要解决的问题是如何将一个数组对象变成函数的多参。

本节的源代码位于06_03_dynamic_create目录，完整的实现方案写在main3.cpp中，这种解决方案是将数据传递给std::tuple，然后传入多参函数。

关键点1: DataInfo结构

在本节的例子中，用DataInfo结构模拟从protobuf消息中序列化回来的结构。DataInfo结构定义如下：

```
struct DataInfo {
    bool IsInt;
    int IntVal;
    std::string StrVal;
    DataInfo(const int value) {...}           // 以int为参数
    DataInfo(const std::string value) {...}    // 以string为参数
};
```

DataInfo结构中，bool值IsInt表示当前这个值是否是int类型的，如果是int类型的，就读取IntVal数据，否则读取StrVal数据。在main函数中的调用如下：

```
int main() {
    RegistToFactory<C1, std::string, std::string>();
    std::list<DataInfo> params;
    params.push_back(DataInfo("v1"));
    params.push_back(DataInfo("test c1"));
    DynamicCall<5>::Invoke(typeid(C1).name(), std::make_tuple(), p
    ...
}
```

在main函数中，首先调用RegistToFactory对C1类进行了创建注册，C1类构造函数的两个参数都是std::string。所以，上面的代码中压入了两个std::string来初始化，将C1需要的数据放在std::list<DataInfo>中模拟从protobuf中取得的数据。

关键点2: DynamicCall模板

注意在main函数中的如下代码：

```
DynamicCall<5>::Invoke(typeid(C1).name(), std::make_tuple(), param
```

参数构造完成之后调用了动态生成函数，调用代码如上一行所示。

下面来详细看看这一行代码的背后都干了些什么，DynamicCall::Invoke函数的实现如下：

```
template<int ICount>
struct DynamicCall {
    template<typename...TArgs>
    static void Invoke(std::string className, std::tuple<TArgs...>
        if (params.size() == 0) {
            ComponentFactoryEx(className, t1, std::make_index_sequ
            return;
        }
        const DataInfo info = (*params.begin());
        params.pop_front();
        if (info.IsInt) {
            auto t2 = std::tuple_cat(t1, std::make_tuple(info.IntV
            DynamicCall<ICount - 1>::Invoke(className, t2, params)
        } else {
            auto t2 = std::tuple_cat(t1, std::make_tuple(info.StrV
            DynamicCall<ICount - 1>::Invoke(className, t2, params)
        }
    }
};
```

先解释一下这段代码的含义，这段代码将std::list<DataInfo>所有的数据展开，最终会传入ComponentFactoryEx模板函数中。这段代码看似短小，但却难以理解，它使用了C++14的一个语法——std::index_sequence。首先来看DynamicCall这个结构，在结构中有一个模板函数，该模板函数有一个递归操作，操作的目的是根据传入的params将数据一个一个转到std::tuple类型中。std::tuple是一个可以包括所有类型的容器。在每一个递归调用中，从params取出第一个数据，再将这个数据增加到std::tuple<TArgs...>t1数据的后面，不断循环，直到params没有数据为止。当params大小等于0时，std::tuple<TArgs...>t1就有了所需的所有参数。这时，调用模板ComponentFactoryEx生成实例类。模板ComponentFactoryEx的实现如下：

```
template <typename... TArgs, size_t... Index>
void ComponentFactoryEx(std::string className, const std::tuple<TA
    auto c1 = ComponentFactory<TArgs...>::GetInstance()->Create(c1
}
```

函数ComponentFactoryEx将std::tuple中的数据展开，传递给类工厂的构造函数ComponentFactory<TArgs...>::GetInstance()->Create。std::index_sequence的语法这里不多介绍，有兴趣的读者可自行搜索一下。本例运行效果如下：

```
[root@localhost TestArgs]# cmake3 ./
[root@localhost TestArgs]# make
[root@localhost TestArgs]# ./testd
call c1. string1:v1 string2:test c1
call c2. int value:1 string value:test c2
```

关键点3：模板函数的编译

在编译本节的例子时，读者可能发现了一个问题，这个小小的程序并没有几行，但是编译的时间似乎比较久。

在main3.cpp的第31行，故意留下了一个警告。在Linux系统下编译本小节中的工程，警告打印得非常详细，某个对象被创建出来，但是没有使用。为什么我们明明只生成了两个对象，却有那么多对象没有使用？这里有一个模板函数编译规则的问题。虽然模板方便了我们使用，但是在编译时模板是被实例化的。如何理解这句话呢？

结构DynamicCall使用了递归的调用。编译器会根据代码生成所有实体类，查看警告可以发现，编译器比我们想象中工作得更多，这意味着在编译阶段，它就已经生成了如下一些类定义：

```
DynamicCall<int, int, int>
DynamicCall<std::string, std::string, std::string>
DynamicCall<int, std::string, std::string>
```

编译器生成了类似这样的所有组合，更要命的是，如果我们去掉DynamicCall<0>的定义，编译器就一直不会停下来，直到提示“编译器的堆空间不足”。那么DynamicCall<0>是如何阻断了编译的这种不能停下来的“疯狂”呢？下面是它的实现，来看一下：

```
template<>
struct DynamicCall<0> {
    template<typename...TArgs>
    static void Invoke(std::string className, std::tuple<TArgs...>
}
```

DynamicCall<0>的实现看上去非常奇怪，它实现了Invoke函数，却是一个空函数。在递归调用时，每递归一次就要调用DynamicCall<ICount - 1>::Invoke(className, t2, params)，当ICount - 1等于0时，也就是到了DynamicCall<0>的时候，Invoke函数变成了空函数，阻断了递归。因为递归收到结束的信号，所以编译就结束了，如果没有这个空函数，编译就会一直进行下去。

好了，我们已经掌握了如何根据协议动态创建类。现在稍微做一些修改，将这个流程融合到框架中。

6.2.3 EntitySystem的工作原理

库libserver重构后的源代码位于06_04_engine目录中。在这个工程中融合了之前两个测试工程，采用协议的方式创建对象，同时去掉了ThreadObject类。下面具体分析一下重点代码。

关键点1：基础类

首先谈论的还是实体Entity和组件Component两个类，这两个类定义在libserver库的component.h和entity.h文件中，需要查看以下几个重要的属性和函数。组件的定义如下：

```
class IComponent : virtual public SnObject {
public:
    ...
    virtual void BackToPool() = 0;
    virtual void ComponentBackToPool();
private:
    IEntity* _parent{ nullptr };
    EntitySystem* _pEntitySystem{ nullptr };
    IDynamicObjectPool* _pPool{ nullptr };
};
```

基础的组件类包括3个基本的变量；一个是父类指针，没有父类时其值为nullptr；另一个是当前所处的EntitySystem类指针；最后一个是对象池指针。对象池指针在该组件不再使用，放回对象池时使用。组件类提供了一个需要子类实现的虚函数BackToPool，该函数在放回对象池之前调用，用于清理本类中的数据。

实体类继承自组件类，但多了一些自己的变量，其定义如下：

```
class IEntity :public IComponent {
public:
    void BackToPool() override;
    template <class T, typename... TArgs>
    void AddComponent(TArgs... args);
    template<class T>
```

```
T* GetComponent();  
...  
protected:  
    std::map<uint64, IComponent*> _components;  
};
```

实体可以拥有自己的组件，存放在std::map<uint64, IComponent*>字典中，同时，它实现了虚函数BackToPool，代码实现如下：

```
void IEntity::BackToPool() {  
    for (const auto& one : _components) {  
        one.second->ComponentBackToPool();  
    }  
    _components.clear();  
}  
void IComponent::ComponentBackToPool() {  
    _parent = nullptr;  
    _pEntitySystem = nullptr;  
    ...  
    BackToPool();  
    if (_pPool != nullptr)  
        _pPool->FreeObject(this);  
}
```

在实体被释放时，它的所有组件都要被释放。每个组件被释放时会调用BackToPool函数，以实现其自身数据的释放。

关键点2: EntitySystem类

在之前的框架中，线程Thread类有两个重要的任务：一是用来管理线程中的对象；二是处理消息。线程管理类ThreadMgr有3个任务：一是管理线程；二是处理消息；三是管理主线程中的全局对象。现在，我们需要将线程类和线程管理类中关于对象以及消息管理的部分提炼出来，这一部分的功能交由新的EntitySystem类来完成，它的定义在entity_system.h文件中。这个类的任务就是管理所有组件、实体类以及消息的处理。

```

class EntitySystem : virtual public SnObject, public IDisposable {
public:
    ...
protected:
    std::list<IUpdateSystem*> _updateSystems;
    std::list<IMessageSystem*> _messageSystems;
    std::map<uint64, IComponent*> _objSystems; // 所有对象
    CacheSwap<Packet> _cachePackets;          // 本线程中的所有待
};

```

去除细枝末节，看看它重要的几个变量。其中，std::map<uint64, IComponent*>存储着本地所有的组件和实体，CacheSwap<Packet>是线程中待处理的网络包，逻辑和之前的完全一样。网络底层收到协议之后，由Network通过ThreadMgr分发到线程中缓存，以便下一帧处理。

EntitySystem类中还有两个集合：一个是IUpdateSystem；另一个是IMessageSystem。这两个接口定义在system.h文件中。对于IUpdateSystem接口来说，它需要继承对象实现Update函数，其定义如下：

```

class IUpdateSystem : virtual public ISystem {
protected:
    IUpdateSystem() = default;
public:
    virtual ~IUpdateSystem() = default;
    virtual void Update() = 0;
};

```

而IMessageSystem接管了MessageList的功能。新的IMessageSystem接口几乎没有修改MessageList的具体功能，但要求实现该接口时实现虚函数RegisterMsgFunction。下面是IMessageSystem接口的定义：

```

class IMessageSystem : virtual public ISystem {
public:
    virtual void RegisterMsgFunction() = 0;
    void AttachCallBackHandler(MessageCallBackFunctionInfo* pCallb
    bool IsFollowMsgId(Packet* packet) const;

```

```

        void ProcessPacket(Packet* packet) const;
protected:
        MessageCallBackFunctionInfo* _pCallBackFuns{ nullptr };
};

```

可以这样说，IUpdateSystem和IMessageSystem将之前ThreadObject的功能一分为二。还记得旧的ThreadObject的定义吗？其代码如下：

```

class ThreadObject : public IDisposable, public MessageList {
public:
    virtual bool Init() = 0;
    virtual void RegisterMsgFunction() = 0;
    virtual void Update() = 0;
    ...
};

```

作为基类ThreadObject承担了初始化、消息注册、更新的所有操作。现在这些操作被提取出来了，变成了一个又一个接口。图6-5展示了IUpdateSystem和IMessageSystems集合的内存数据。

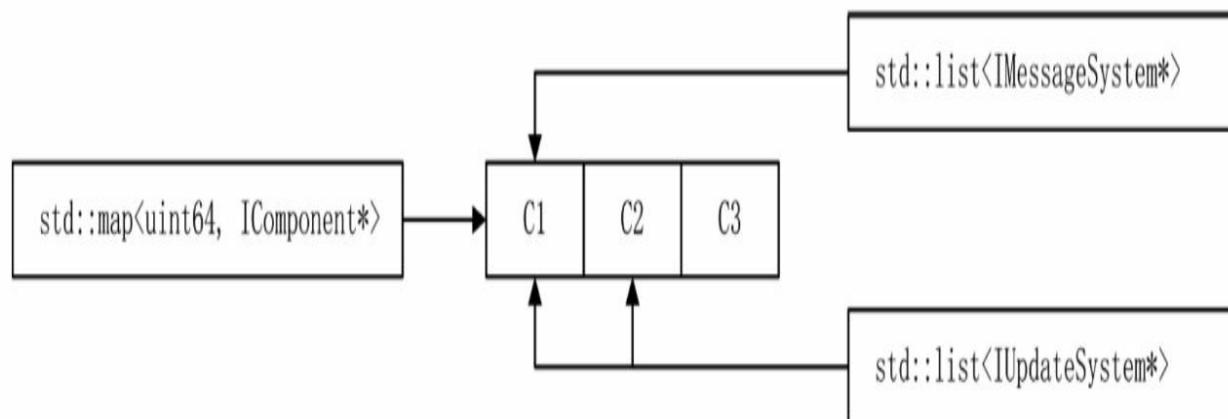


图6-5 组件与接口

从图6-5中可以看出，当实体或者组件既实现了IUpdateSystem接口又实现了IMessageSystem接口时，其指针会分别放入std：：list<IUpdateSystem*>和std：：list<IMessageSystem*>中，以方便读

取。

关键点3：创建实体和组件

了解了EntitySystem类的数据组成，EntitySystem类管理着所有的组件和实体，有3种方式可以创建组件或者实体：

- (1) 在EntitySystem中直接创建一个实体或者组件。
- (2) 由一个已存在的实体创建组件。
- (3) 通过协议创建，这种方式就要用到我们前面两个小节提到的功能，通过类名动态创建类。

先来看看创建组件的第一种方法，如何在EntitySystem中直接创建。在EntitySystem类中封装了一个创建函数AddComponent，其实现代码位于entity_system.h文件中，具体代码如下：

```
template <class T, typename ... TArgs>
T* EntitySystem::AddComponent(TArgs... args) {
    auto pComponent = DynamicObjectPool<T>::GetInstance()-> Malloc
    AddToSystem(pComponent);
    return pComponent;
}

void EntitySystem::AddToSystem(IComponent* pComponent) {
    pComponent->SetEntitySystem(this);
    _objSystems[pComponent->GetSN()] = pComponent;
    const auto objUpdate = dynamic_cast<IUpdateSystem*>(pComponent)
    if (objUpdate != nullptr)
        _updateSystems.emplace_back(objUpdate);
    const auto objMsg = dynamic_cast<IMessageSystem*>(pComponent);
    if (objMsg != nullptr) {
        objMsg->RegisterMsgFunction();
        _messageSystems.emplace_back(objMsg);
    }
}
```

一个组件被创建完成之后，将它放到EntitySystem类的std::map<uint64, IComponent*>字典中，同时会判断其是否具有IUpdateSystem或者IMessageSystem接口，如果具有，就放置到不同的

队列中。如果是一个IMessageSystem接口，在归入队列时就会调用RegisterMsgFunction函数进行消息注册。

创建组件的第二种方法是由实体发起的，调用的是IEntity::AddComponent函数。实现函数如下：

```
template <class T, typename... TArgs>
inline void IEntity::AddComponent(TArgs... args) {
    auto pComponent = DynamicObjectPool<T>::GetInstance()-> Malloc
    AddToSystem(pComponent);
}
void IEntity::AddToSystem(IComponent* pComponent) {
    pComponent->SetParent(this);
    _components.insert(std::make_pair(pComponent->GetSN(), pCompon
    GetEntitySystem()->AddToSystem(pComponent);
}
```

实体类为自己创建了组件之后，也会调用EntitySystem::AddToSystem，将组件放在std::map<uint64, IComponent*>字典中。当一个组件被创建之后，它的动作会被固定下来，如果它有IUpdateSystem接口，那么每帧调用都会更新函数；如果有IMessageSystem接口，那么每个Packet包到来时都会询问实体是否需要处理该类型的Packet。

创建组件的第三种方法，“通过创建协议进行创建”是非常复杂的一种方法，后面再详细讲解。总之，通过这些方法可以自由灵活地创建一个又一个的组件或者实体，这些组件或者实体在创建时都调用了DynamicObjectPool对象池类，都是从对象池中生成的。

关键点4：实体更新操作

组件创建完成之后，下一个需要弄清楚的问题是IUpdateSystem和IMessageSystem是如何运作的。我们来看一下新的线程类：

```
class Thread : public EntitySystem {
public:
    void Start();
    bool IsRun() const;
```

```
        bool IsStop() const;
        bool IsDispose();
    private:
        ThreadState _state;
        std::thread _thread;
};
```

新的Thread变得十分简单，功能也相对单一，只负责本线程的启动与关闭。Thread类继承了EntitySystem，所以Thread类也是一个实体对象管理类。调用Start函数启动了一个线程，该线程只要在生命周期中，就会不停地调用Update帧函数，这个Update函数就是EntitySystem类中的Update，在帧函数中。其代码如下：

```
void Thread::Start() {
    _thread = std::thread([this]() {
        InitComponent();
        _state = ThreadState_Run;
        while (!Global::GetInstance()->IsStop) {
            Update();
            std::this_thread::sleep_for(std::chrono::milliseconds(
        ));
        }
        _state = ThreadState_Stopped;
    });
}

void EntitySystem::Update() {
    UpdateMessage();
    auto iter = _updateSystems.begin();
    while (iter != _updateSystems.end()) {
        auto pComponent = dynamic_cast<IComponent*>(*iter);
        ...
        (*iter)->Update();
        ++iter;
    }
}
```

当组件或实体进入EntitySystem系统之后，带有IUpdateSystem接口的实例会被放到IUpdateSystem队列中，而每一帧在这个队列中的实例都会执行Update函数，以达到更新的目的。除非进程退出，否则线程中对Update函数的调用永不会停止。

关键点5：新的消息系统

在前面的框架中，在处理消息时，采用的方式是继承底层类ThreadObject，并实现RegisterMsgFunction虚函数来达到对消息的处理。

在实际应用中，发现并不是所有的类都需要消息传递，有一些类可能只是一个动作的执行者，例如HttpRequest类，所以在新的框架中设计了一个IMessageSystem接口。当对象继承自该接口时，就具有了消息处理的功能。前面已经展示过IMessageSystem接口的定义，现在看看IMessageSystem是如何工作的？

在之前的框架中，线程收到一个消息，都会对线程上的包裹类进行一次查询，查询该实例是否对该消息有兴趣。而新的ECS框架中仅在有IMessageSystem接口的对象中进行询问，从而过滤掉不需要消息通知的实例。其实现代代码如下：

```
void EntitySystem::UpdateMessage() {
    _packet_lock.lock();
    if (_cachePackets.CanSwap()) {
        _cachePackets.Swap();
    }
    _packet_lock.unlock();
    auto pMsgList = _cachePackets.GetReaderCache();
    for (auto itMsg = pMsgList->begin(); itMsg != pMsgList->end();
        auto pPacket = (*itMsg);
        for (auto iter = _messageSystems.begin(); iter != _message
            auto pObj = (*iter);
            if (pObj->IsFollowMsgId(pPacket))
                pObj->ProcessPacket(pPacket);
        }
    }
    pMsgList->clear();
}
```

函数EntitySystem::UpdateMessage是由EntitySystem::Update函数调用的，以达到每一帧检查处理消息的目的。而消息处理的流程和之前一致，这里就不再重复阐述了。

现在或许还看不出旧框架与新框架的区别，不就是从类变成了组件吗？下面我们继续对login和robots中的对象重构来继续发掘这个新框架的优势。

6.3

基于ECS框架的login和robots工程

重构login和robots工程中的类比重构libserver中的类要容易许多。因为这些类是按Actor原则组织的代码，所以可以在不变更一行功能代码的前提下对这些类进行一次全新的重构。本节的源代码位于06_05_login目录中。

6.3.1 Account类

将login进程中的类迁移到新框架，各个类的修改大同小异，我们先从login工程的Account类开始，这个类负责账号的验证。新的定义如下：

```
class Account :public Component<Account>, public IMessageSystem, p
public:
    void AwakeFromPool() override {}
    void RegisterMsgFunction() override;
    virtual void BackToPool() override;
    ...
private:
    LoginObjMgr _playerMgr;
};
```

在旧框架体系中，所有逻辑都需要继承自ThreadObject，实现Init、RegisterMsgFunction和Update函数。实际上，Account类中的Init和Update两个函数都是空函数。而新框架体系中，Account类仅继承了IMessageSystem接口来实现消息处理的需求。

当Account类被实例化之后，它会进入EntitySystem系统中，这个系统检测到该类实现了IMessageSystem接口，会调用该类实现的RegisterMsgFunction函数完成消息注册，再将它纳入消息实例的队列中，以便消息到来时进行处理。

在Account类中有一处修改值得注意，就是创建HttpRequestAccount类进行账号验证时，旧代码如下：

```
auto pHttp = new HttpRequestAccount(protoCheck.account(), protoChe
ThreadMgr::GetInstance()->AddObjToThread(pHttp);
```

在上面的代码中，创建了一个HttpRequestAccount对象，通过AddObjToThread函数将它放在了某个线程中。重构后的代码如下：

```
ThreadMgr::GetInstance()->CreateComponent<HttpRequestAccount>(prot
```

在新框架中统一调用CreateComponent函数传入类名与参数，动态创建一个实体。这个创建的HttpRequestAccount可能与Account类并不在同一个线程中。下面重点看看这个动态创建的过程。

6.3.2 动态创建组件或实例

在前面的章节中讲到过创建组件或实体的3种方法，再来回顾一下：

- (1) 在EntitySystem中直接创建一个实体或组件。
- (2) 由一个已存在的实体创建组件。
- (3) 通过协议创建。

前两种方案都在线程内部创建对象，但第三种（也就是马上要讲到的这一种）是跨线程来创建对象的。下面看看它的几个关键点。

关键点1: IAwakeFromPoolSystem接口

在Account类中的实现接口有一个是我们第一次见到的——IAwakeFromPoolSystem接口。这个接口提供对象池中对象初始化的功能。为了便于管理对象，从现在开始所有的对象都从对象池中获取。首先需要说明的是，对象池生成接口IAwakeFromPoolSystem，定义在system.h中：

```
template <typename... TArgs>
class IAwakeFromPoolSystem : virtual public ISystem {
protected:
    IAwakeFromPoolSystem() = default;
public:
    virtual ~IAwakeFromPoolSystem() = default;
    virtual void AwakeFromPool(TArgs... args) = 0;
};
```

所有想要从对象池中生成的组件，都要继承这个接口。再以NetworkListen类为例，新的定义如下：

```
class NetworkListen :public Network, public IUpdateSystem, public
public:
```

```
void AwakeFromPool(std::string ip, int port);  
...  
}
```

NetworkListen类继承了接口IAwakeFromPoolSystem<std::string, int>, 也就是需要实现AwakeFromPool (std::string, int) 函数。当NetworkListen从Pool池中被唤醒时, 就会调用该函数, 调用发生在DynamicObjectPool<T>::MallocObject函数中。

回到Account类上来, 它实现的是IAwakeFromPoolSystem<>, 不需要任何参数, 也就是说Account类需要实现AwakeFromPool () 函数。从对象池被唤醒时, 这个函数将被调用。

关键点2: 创建协议

现在创建一个实体, 不再直接对它进行创建操作。以login工程为例, 在初始化组件时的代码实现如下:

```
void LoginApp::InitApp() {  
    _pThreadMgr->CreateComponent<NetworkListen>(std::string("127.0  
    _pThreadMgr->CreateComponent<RobotTest>();  
    _pThreadMgr->CreateComponent<Account>();  
}
```

在login进程启动时, 创建了NetworkListen、RobotTest、Account三个类。采用协议的方式将创建类的类名和需要的参数以协议的方式传递到线程中, 再由线程自行创建。

在上面的代码中, 类NetworkListen的创建需要两个参数, 监听时需要绑定IP地址与端口, 通过参数的形式传递。而RobotTest类和Account类不需要任何参数。

重构后的代码看上去更直观, 也更简洁。下面来看看CreateComponent函数都做了些什么事情。CreateComponent函数定义在thread_mgr.h文件中。

```

template<class T, typename ...TArgs>
inline void ThreadMgr::CreateComponent(TArgs ...args) {
    std::lock_guard<std::mutex> guard(_create_lock);
    const std::string className = typeid(T).name();
    if (!ComponentFactory<TArgs...>::GetInstance()->IsRegistered(className))
        RegistToFactory<T, TArgs...>();
}
Proto::CreateComponent proto;
proto.set_class_name(className.c_str());
AnalyseParam(proto, std::forward<TArgs>(args)...); // 参数分析
auto pCreatePacket = new Packet(Proto::MsgId::MI_CreateComponent);
pCreatePacket->SerializeToBuffer(proto);
_createPackets.GetWriterCache()->emplace_back(pCreatePacket);
}

```

函数CreateComponent是一个模板函数，每一个类都可以调用该函数进行创建。因为每一个类的初始化数据类型都不一样，所以这里必须是一个多参的模板函数。

在这个函数中有一个关键点是之前讨论过的——如何通过类名创建实体的解决方案。在CreateComponent函数中，通过类名在类工厂中查找是否已经有注册函数。如果没有，就根据提供的多参类型进行RegistToFactory注册。随后将类名和参数赋值到protobuf的协议中，协议号为MI_CreateComponent，对应结构为Proto::CreateComponent。循环赋值参数的关键在于AnalyseParam函数，它是一个重载函数，有如下几个声明：

```

template <typename...Args>
void AnalyseParam(Proto::CreateComponent& proto, int value, Args...args) { }
template <typename...Args>
void AnalyseParam(Proto::CreateComponent& proto, std::string value, Args...args) { }
void AnalyseParam(Proto::CreateComponent& proto) { }

```

函数AnalyseParam有3个实现：一个处理int类型的数据，一个处理string类型的数据，最后一个是空函数。就用创建NetworkListen来说明这几个函数的调用关系。生成NetworkListen类的调用代码如下，有string类型和int类型两个参数。

```

CreateComponent<NetworkListen>(std::string("127.0.0.1"), 2233);

```

函数AnalyseParam在匹配的时候，因为第一个参数是string类型的，第一次对应的是函数AnalyseParam(proto, string, args)，其实现代码如下：

```
template<typename ... Args>
void ThreadMgr::AnalyseParam(Proto::CreateComponent& proto, std::s
    auto pProtoParam = proto.mutable_params()->Add();
    pProtoParam->set_type(Proto::CreateComponentParam::String);
    pProtoParam->set_string_param(value.c_str());
    AnalyseParam(proto, std::forward<Args>(args)...);
}
```

该函数的第3个参数args既可以是零个参数，又可以是多个参数。该函数在Proto::CreateComponent对象中Proto::CreateComponentParam对象，用于存放创建组件需要的参数。在创建NetworkListen这个例子中，存放的是NetworkListen类需要的第一个参数，即将std::string("127.0.0.1")这个变量转存到了Proto::CreateComponentParam结构中，参数类型为CreateComponentParam::String。之后，函数再次递归调用了AnalyseParam(proto, args)，注意这个调用抛弃了string值，因为它已经处理过了，参数args展开之后，第一个值是int类型的，所以此时匹配到的函数是AnalyseParam(proto, int, args)，其实现代码如下：

```
template<typename ... Args>
void ThreadMgr::AnalyseParam(Proto::CreateComponent& proto, int va
    auto pProtoParam = proto.mutable_params()->Add();
    pProtoParam->set_type(Proto::CreateComponentParam::Int);
    pProtoParam->set_int_param(value);
    AnalyseParam(proto, std::forward<Args>(args)...);
}
```

同理，该函数创建了一个新的Proto::CreateComponentParam对象，用于存放第二个参数数据，函数处理的最后一行同样是AnalyseParam(proto, args)，但真实情况args是一个空值，也就是说最后它调用的是void AnalyseParam(Proto::CreateComponent&proto)，已经没有数据可以处理了，该函数为一个

空函数，编译器遇到它不会再递归了。

以此类推，所有的参数都赋值到Proto：：CreateComponent对象中，函数SerializeToBuffer将这个协议数据附加到网络数据传输对象Packet的Buffer中。最后将Packet放入待处理的createPackets集合中，将在下一帧对该数据进行处理。

集合createPackets在ThreadMgr中的定义如下：

```
class ThreadMgr :public Singleton<ThreadMgr>, public EntitySystem
public:
    ...
Private:
    size_t _threadIndex{ 0 };    // 实现线程对象均衡
    // 创建组件消息
    std::mutex _create_lock;
    CacheSwap<Packet> _createPackets;
};
```

在帧函数ThreadMgr：：Update中，集合createPackets中的Packet会被放置到某个线程中，至于放到哪一个线程是由变量threadIndex来决定的，每次使用后加1，基本保证了线程中创建的对象是均衡的。

```
void ThreadMgr::Update() {
    _create_lock.lock();
    if (_createPackets.CanSwap()) {
        _createPackets.Swap();
    }
    _create_lock.unlock();
    auto pList = _createPackets.GetReaderCache();
    for (auto iter = pList->begin(); iter != pList->end(); ++iter)
        const auto packet = (*iter);
        if (_threadIndex >= _threads.size())
            _threadIndex = 0;
        _threads[_threadIndex]->AddPacketToList(packet);
        _threadIndex++;
    }
    pList->clear();
    EntitySystem::Update();
}
```

这样，创建组件的Packet落到某一个线程上。每个线程中都有CreateComponentC组件对这个协议进行处理。

关键点3: CreateComponentC组件

创建对象的Packet被发送到每个线程，这时CreateComponentC类开始工作，它会处理协议号MI_CreateComponent。这个类的定义在create_component.h文件中，定义如下：

```
class CreateComponentC :public Entity<CreateComponentC>, public IM
public:
    void AwakeFromPool() override {}
    void RegisterMsgFunction() override;
    void BackToPool() override;
private:
    void HandleCreateComponent(Packet* pPacket) const;
    ...
};
```

在每一个线程中都有CreateComponentC类的实例，启动线程时就创建好了。该代码位于EntitySystem::InitComponent函数中，在线程启动时该函数被调用。EntitySystem::InitComponent函数创建线程需要使用到的基本组件，目前只有一个CreateComponentC类。

```
void EntitySystem::InitComponent() {
    AddComponent<CreateComponentC>();
}
```

CreateComponentC类是一个非常普通的组件，从对象池中创建，实现了IAwakeFromPoolSystem接口。该组件本身没有什么多余的数据，不需要更新操作，所以它没有IUpdateSystem接口，但它需要处理协议MI_CreateComponent，实现了IMessageSystem接口。其注册和处理函数的实现代码如下：

```
void CreateComponentC::RegisterMsgFunction() {
    auto pMsg = new MessageCallBackFunction();
```

```

        this->AttachCallbackHandler(pMsg);
        // 注册MI_CreateComponent的处理函数
        pMsg->RegisterFunction(Proto::MsgId::MI_CreateComponent, BindF
    }
    void CreateComponentC::HandleCreateComponent(Packet* pPacket) const
    {
        Proto::CreateComponent proto = pPacket->ParseToProto<Proto::Cr
        const std::string className = proto.class_name();
        if (proto.params_size() >= 5) {
            std::cout << " !!!! CreateComponent failed. className:" <<
            return;
        }
        auto params = proto.params();
        const auto pObj = DynamicCall<5>::Invoke(GetEntitySystem(), cl
        if (pObj == nullptr) {
            std::cout << " !!!! CreateComponent failed. className:" <<
        }
    }
}

```

当收到MI_CreateComponent消息后，将数据从Packet中取出来，然后调用函数DynamicCall<5>().Invoke。最终，将调用EntitySystem::AddComponentByName函数生成实体。

```

template<typename ...TArgs>
inline IComponent* EntitySystem::AddComponentByName(std::string cl
    auto pFactory = ComponentFactory<TArgs...>::GetInstance();
    auto pComponent = pFactory->Create(className, std::forward<TAr
    if (pComponent == nullptr)
        return nullptr;
    AddToSystem(pComponent);
    return pComponent;
}

```

ComponentFactory::Create函数调用的就是之前在RegistToFactory中注册的函数。它的变参args的个数、类型都和ThreadMgr::CreateComponent调用时的个数、类型完全一致。

关键点4：类工厂如何从对象池中取出数据

前面已经有例子说明了类工厂ComponentFactory的创建对象流程，它最终调用RegistToFactory类中的静态CreateComponent函数生成对

象，其定义如下：

```
template<typename T, typename...Targs>
class RegistToFactory {
public:
    RegistToFactory() {
        ComponentFactory<Targs...>::GetInstance()->Regist(typeid(T)
    }
    static T* CreateComponent(Targs... args) {
        return DynamicObjectPool<T>::GetInstance()-> MallocObject(
    }
};
```

当我们调用RegistToFactory的构造函数时，实际上是向ComponentFactory注册了类名与RegistToFactory::CreateComponent函数的关系。从上面的代码可以看出，调用CreateComponent之后真实的创建函数是DynamicObjectPool::MallocObject提供的，是从对象池中取出的对象，该函数的实现如下：

```
template <typename T>
template <typename ... Targs>
T* DynamicObjectPool<T>::MallocObject(Targs... args) {
    ...
    auto pObj = _free.front();
    _free.pop();
    ...
    pObj->ResetSN();
    pObj->AwakeFromPool(std::forward<Targs>(args)...);
    ...
    return pObj;
}
```

从空闲对象池中取出一个对象，调用该对象的AwakeFromPool函数，这正是继承自IAwakeFromPoolSystem接口所实现的函数。以NetworkListen类为例，它创建时的代码如下：

```
CreateComponent<NetworkListen>(std::string("127.0.0.1"), 2233)
```

则它在对象池中被唤醒时，调用函数参数如下：

```
NetworkListen::AwakeFromPool(std::string ip, int port)
```

在MallocObject函数中唤醒了一个未使用的对象，同时完成了初始化工作，图6-6描述了整个流程。

总的来说，跨线程生成对象这个过程有点复杂，你可能充满疑惑，为什么我们要大费周章地跨线程生成对象？要回答这个问题，关键的一点就是依赖关系。

我们生成的组件是从对象池中提取出来的，在调用AwakeFromPool函数初始化时，是一个非常不确定的调用。在这个过程中可能产生新的组件或实体，也可以调用一些组件或实体，如果一个组件在初始化时需要调用的是线程中的实体，那么在主线程中新建组件就会产生异常。

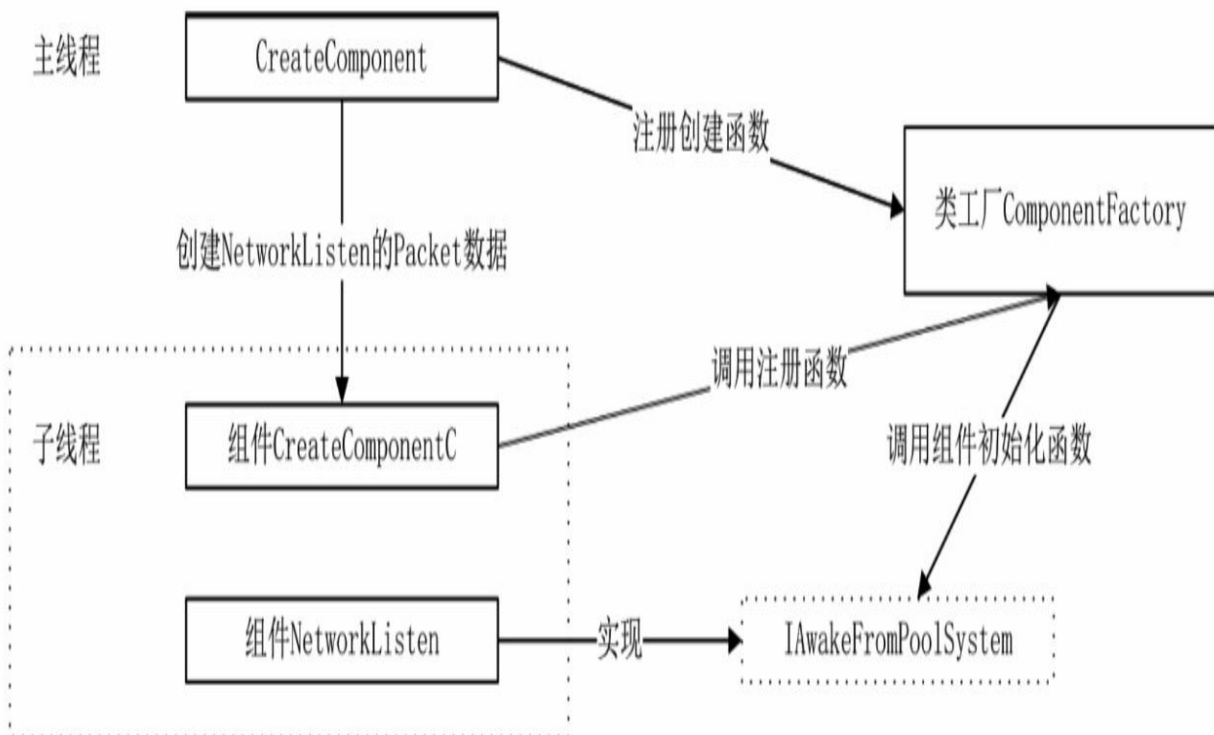


图6-6 创建组件的流程

关于动态创建组件就介绍到这里。总之，我们通过动态创建组件创建了Account组件和NetworkListen组件。Account组件相对简单，下面重点来看一下在ECS框架下，NetworkListen组件中的数据发生了些什么样的改变。

6.3.3 ECS框架下的网络通信

在login工程中，为了适应ECS设计思路，重构了Account类，也重构了HttpRequestAccount类，逻辑功能修改并不大。在robots工程中，Robot类去掉了构造函数和Init初始化函数，由AwakeFromPool函数统一处理，其他逻辑保持不变。Robot和RobotMgr的优化仅限于修改了几个实现的接口，这里就不再一一进行比较了。

从这些细节中可以发现，实际上，当前的ECS框架将ThreadObject的功能进行了拆分。因为并不是每个类都需要实现Update、处理协议这类操作，使用接口的方式让每个类按需实现接口，这样的设计使得类变得更加简洁。

下面我们用重构的login工程与robot工程来重点看网络部分的相关重构。本节的参考源代码位于06_05_login目录中。

关键点1: Network类

首先来看网络部分的代码移入新框架中需要重点修改的部分。Network类修改前是一个ThreadObject，重载并实现了ThreadObject的函数；Network类修改后是一个Entity对象。旧定义与新定义对比如下：

```
class Network : public ThreadObject, public ISocketObject { ... }
class Network : public Entity<Network>, public IMessageSystem, pub
```

Network类中的函数基本保持不变，在新框架中，Network类继承了IMessageSystem接口，也就是说它需要实现RegisterMsgFunction函数，可以对自己关心的协议注册处理函数。除此之外，增加了INetwork接口，其定义如下：

```
class INetwork : public ISocketObject {
public:
    virtual ~INetwork() = default;
    virtual void SendPacket(Packet*& pPacket) = 0;
```

```
};
```

该接口是为了跨线程时使用的，不论何时都可以向一个Network调用SendPacket函数发送Packet数据，这一流程是做了加锁处理的。

关键点2: NetworkListen类和NetworkConnector类

在NetworkListen类中，新旧代码的功能没有大的变化。重构后的类增加了IAwakeFromPoolSystem接口，希望从对象池中获取数据。NetworkListen类还实现了IUpdateSystem接口用于更新，实现网络层的数据收发功能。与之前的处理一样，Update函数本身没有任何修改，只是它的调用主体与之前不一样了。NetworkListen类的定义如下：

```
class NetworkListen :public Network, public IUpdateSystem, public
public:
    void AwakeFromPool(std::string ip, int port);
    void Update() override;
protected:
    virtual int Accept();
};
void NetworkListen::AwakeFromPool(std::string ip, int port) {
    if (::listen(_masterSocket, SOMAXCONN) < 0) {
        std::cout << "::listen failed." << _sock_err() << std::endl;
        return;
    }
    ...
}
```

旧框架中的Listen函数被AwakeFromPool替代。创建完成之后，网络监听就做完了。不需要编写任何多余的代码调用额外的函数。连接NetworkConnector类的情况与NetworkListen的情况基本一致，仅做了实现接口的变更。

- 重构前定义：class NetworkConnector: public Network{···}。
- 重构后定义：class NetworkConnector: public Network, public IUpdateSystem, public IAwakeFromPoolSystem<std::string, int>{···}。

在重构前后，基本流程并没有发生大的改变，函数Connect的显式

调用改为初始化时的隐式调用。

```
void NetworkConnector::AwakeFromPool(std::string ip, int port) {  
    ...  
    Connect(ip, port);  
}
```

每个在对象池中的对象都在AwakeFromPool函数中实现了自身特点的编码，这使得所有类具有一致性，底层不关心类的真正作用，只需要调用AwakeFromPool函数即可。

关键点3: NetworkLocator组件

为了便于发送协议，新增名为NetworkLocator的组件，该组件是一个全局组件。这是一个网络层的定位组件，因为在某个进程中可能存在多个Network网络实例，这些网络对象可能是对外连接的NetworkConnector，也可能是一个监听对象NetworkListen。NetworkLocator组件的作用是为了快速找到NetworkListen或NetworkConnector的实例，用于发送数据。组件定义在network_locator.h文件内：

```
class NetworkLocator : public Component<NetworkLocator>, public IA  
public:  
    void AwakeFromPool() override {};  
    void BackToPool() override;  
    void AddListenLocator(INetwork* pNetwork, NetworkType networkT  
    INetwork* GetListen(NetworkType networkType);  
    ...  
private:  
    std::mutex _lock;  
    std::map<APP_TYPE, std::map<int, INetwork*>> _connectors;  
    // 可能存在两个listen，即tcp和http同时存在  
    std::map<NetworkType, INetwork*> _listens;  
};
```

在ServerApp类的构造函数中，这个组件就被创建出来了，放在了主线程中。

```
ServerApp::ServerApp(APP_TYPE appType) {  
    ...  
    _pThreadMgr->AddComponent<NetworkLocator>();  
}
```

当NetworkListen类被唤醒时，调用了NetworkLocator::AddListenLocator函数，将其注册到Locator组件中。根据类型将创建的NetworkListen指针放在std::map<NetworkType, INetwork*>字典中以备使用。

```
void NetworkListen::AwakeFromPool(std::string ip, int port) {  
    auto pNetworkLocator = ThreadMgr::GetInstance()->GetComponent<  
        pNetworkLocator->AddListenLocator(this, NetworkTcpListen);  
    ...  
}  
void NetworkLocator::AddListenLocator(INetwork* pNetwork, NetworkT  
    std::lock_guard<std::mutex> guard(_lock);  
    _listens[networkType] = pNetwork;  
}
```

需要发送数据时，IMessageSystem中提供了一个静态函数SendPacket，在SendPacket中使用到了NetworkLocator组件，其实现如下：

```
void IMessageSystem::SendPacket(Packet* pPacket) {  
    auto pNetworkLocator = ThreadMgr::GetInstance()->GetComponent  
    auto pNetwork = pNetworkLocator->GetNetworkConnector(pPacket->  
    if (pNetwork != nullptr) {  
        pNetwork->SendPacket(pPacket);  
        return;  
    }  
    // 最后试着向listen发送数据  
    pNetwork = pNetworkLocator->GetListen(NetworkTcpListen);  
    pNetwork->SendPacket(pPacket);  
}
```

在发送Packet数据时，根据Packet中的Socket值可以找到一个对应的INetwork实例，这个INetwork实例可能是NetworkConnector，也可能

是NetworkListen中的一个Socket通道。因此，先找一找有没有对应的NetworkConnector实例，如果没有找到，就转到NetworkListen中进行处理。目前，我们的工程还比较简单，所以只会转到NetworkListen进行处理。

6.3.4 执行效率

到目前为止，框架有了一些变化。虽然加入了对象池，但是在Packet的使用上还是沿用的创建对象的方式，这一点在随后的示例中会进行修改。现在，已经可以在这个新的框架上运行数据了，虽然它还不完美，但是基础的组件已经可以在其中运行了。我们来看看测试结果，运行1000个账号的执行效率：

```
[root@localhost bin]# ./logind
Login-Connecting over. time:0.230887s
Login-Connected over. time:0.230958s
Login-Logged over. time:1.25748s
```

执行1000个账号的登录大约需要1.2秒。在本次重构中，我们摆脱了ThreadObject这个基类，再也不需要有冗长的继承关系了，实现了按需继承。理解ECS的思路与流程之后，为了巩固这个思路，我们为框架增加两个工具类：一个是YAML配置读取；另一个是log4日志打印。接下来深入应用一下这个新的框架体系。

YAML文件

我们建立了服务端基本框架，不过作为一个可以运营的游戏来说，这还是太少了，仅仅只是开始。回头看看，我们完成了不少工作，在随后的章节会进一步增加功能点。首先需要增加的功能点是服务器的配置。

需要说明一下，对于游戏开发来说有两种配置：

- (1) 策划文档的配置。
- (2) 程序运行时的配置。

策划文档与游戏逻辑息息相关。例如，有多少张地图，有多少个NPC，每个NPC有什么样的属性，每个角色有多少技能，每个技能有什么效果。策划文档的配置方案在程序端的实现非常多样，可以用CSV，可以用TXT，可以挂Lua脚本，也可以使用Python，方法多种多样，没有好坏之分。

另一种配置是运行时的服务端参数配置，这种配置与游戏逻辑没有直接关系，却是运行服务的重要配置。将一些在代码中的参数暴露在配置文件中，不重新编译程序就可以得到我们想要的效果。例如，在配置线程个数时，现在的方案是在代码中写死3个线程，如果每次都修改代码，那么肯定是不够方便的。除此之外，请求验证账号的URL也是写死在代码中的，这肯定是不合理的。鉴于此，需要有一个服务端参数配置文件。

参数配置的文件类型是很多样的，有INI、XML，当然也可以挂Lua、Python等脚本语言。在本书中使用YAML文件进行配置。YAML已有

20多年的历史，与XML这类标记性语言不同，它有着非常强的可读性，无冗余标记，近几年日渐流行起来。我们还是先安装编译需要的YAML库版本，再来介绍如何使用它。

6.4.1 YAML编译安装

本书使用的YAML库为yaml-cpp，我们为本书下载的版本存放在本书配套源代码根目录的lib目录下。

关键点1：编译Linux版

在安装Linux环境时，我们可以先在yum仓库下查看有没有现成可用的版本。具体查看步骤如下：

```
[root@localhost ~]# yum search yaml-cpp
yaml-cpp-debuginfo.x86_64 : Debug information for package yaml-cpp
yaml-cpp-devel.x86_64 : Development files for yaml-cpp
...
[root@localhost ~]# yum info yaml-cpp-devel.x86_64
```

使用info指令可以看到，yum仓库是0.5版本。这个版本需要boost支持，如果要使用该版本，那么还需要安装boost库，0.6以后的版本不再使用boost，所以我们下载0.6.2版本。

(1) 下载yaml-cpp源代码，下载地址为<https://github.com/jbeder/yaml-cpp/tags>，本书下载的文件为yaml-cpp-yaml-cpp-0.6.2.tar.gz。源代码的dependence目录中提供了该文件。

(2) 在Linux下解压并进入目录。执行CMake配置工程，编译动态库。

```
[root@localhost dependence]# tar xf yaml-cpp-yaml-cpp-0.6.2.tar.gz
[root@localhost dependence]# cd yaml-cpp-yaml-cpp-0.6.2/
[root@localhost yaml-cpp-yaml-cpp-0.6.2]# cmake3 -DCMAKE_CXX_FLAG
```

(3) 执行make，将编译工程安装到系统目录下：

```
[root@localhost yaml-cpp-yaml-cpp-0.6.2]# make
[root@localhost yaml-cpp-yaml-cpp-0.6.2]# make install
```

(4) 在我们的框架中使用YAML时，修改CMakeLists.txt文件的CMAKE_CXX_FLAGS属性，增加-lyaml-cpp参数。

关键点2：编译Windows版

在Windows下编译YAML需要用到Windows版的CMake。本书已经准备了编辑好的Windows库，如果读者对编译部分不感兴趣，那么可以跳过这一小段。

(1) 打开cmake-gui.exe。在source code栏写上YAML的源代码目录，build目录是输出目录，可以随便命名目录名，只要方便使用即可。

(2) 单击Configure按钮来配置工程，会弹出配置界面，选择编译器指向我们安装的Visual Studio，平台选择Win32，注意编译生成的版本是x86版本。

(3) 单击Generate生成工程。打开build目录就会发现已有YAML_CPP.sln工程文件，启动编译器，只需要编译yaml-cpp static md。在编译过程中，可能会出现exceptions.h (123) : error C3646: “_NOEXCEPT” (未知重写说明符) 的错误。打开exceptions.cpp和exceptions.h两个文件，屏蔽如下代码：

```
//#ifdef _MSC_VER
//    #define YAML_CPP_NOEXCEPT _NOEXCEPT
//#else
//    #define YAML_CPP_NOEXCEPT noexcept
//#endif
```

(4) 最后生成的文件位于目录yaml-cpp-yaml-cpp-0.6.2\build\Debug\中。将文件libyaml-cppmdd.lib复制到根目录下的libs\windows公共目录中，以后所有的工程都可以使用。

(5) 库生成之后，还需要将yaml-cpp-yaml-cpp-0.6.2\include目

录复制到本书源代码根目录include\windows下。这样，在Windows下所有工程都可以使用该目录。

（6）打开目录06_06_yaml中的工程，修改login、robots工程的配置，选择“连接器”→“输入”→“附加依赖项”，追加libyaml-cppmdd.lib库。

这样，我们就可以在项目中使用YAML了。

6.4.2 读取YAML配置文件

本节的参考源代码位于06_06_yaml目录中，在该目录新增了一个res目录，其中有一个新的engine.yaml配置文件。

YAML语法中没有Tab制表符，只有空格，一旦输入了Tab制表符，YAML就会报错。如果读者之前没有接触过YAML，这里有一个简单的对比方式，看看XML文件与YAML文件的区别：

```
<?xml version="1.0" encoding="utf-8"?>
<ServerConfig>
  <LoginConfig
    thread_num="2"
    url_login="HTTP://192.168.0.120/member_login_t.php"
    ip="192.168.0.100" port="5550"/>
  </LoginConfig>
</ServerConfig>
```

相同内容，采用YAML文件配置如下：

```
login:
  thread_num: 2
  url_login: HTTP://192.168.0.120/member_login_t.php
  ip: 127.0.0.1
  port: 5550
```

从上面的对比中可以看出YAML比XML更加简洁，没有多余的无关紧要的文字。YAML对于列表的配置也相当简洁。与XML格式的对比如下：

```
<LoginConfig>
  <Login id="1" ip="192.168.0.100" port="5551"/>
  <Login id="2" ip="192.168.0.100" port="5552"/>
</LoginConfig>
```

采用YAML配置如下：

```
login:
  apps:
    - id: 1
      ip: 192.168.0.100
      port: 5551
    - id: 2
      ip: 192.168.0.100
      port: 5552
```

后者的配置更让人一目了然，“-”语法简洁地标识出了数组。服务端的配置文件常用的还是INI类型的文件，INI是一个赋值型的配置文件，一个属性对应一个值，比XML简洁，但在数组配置上表现得非常差，要配置出一个数组是非常难的。因此，我们选择YAML作为服务器参数的配置文件。

在engine.yaml文件中定义了login工程和robot工程的属性，按照之前章节的介绍，后面还会有game、space工程的相关配置。在本例中，该文件的内容如下：

```
login:
  thread_num: 2
  url_login: http://192.168.0.120/member_login_t.php
  ip: 127.0.0.1
  port: 5550
robot:
  thread_num: 2
```

对于login和robots工程而言都是需要多线程的，字段thread_num指定了子线程的数量。而在login工程配置中，当前比较多变的数据就是验证账号的接口，将其提取出来，即url_login字段，同时还将监听地址和端口，分别存于ip和port字段中。

定义好需要的数据之后，下面需要为YAML编写一个读取类，其类在libserver工程的yaml.h中。首先为每一个配置数据属性定义结构，定义如下：

```
struct YamlConfig {
    virtual ~YamlConfig() {}
```

```

};
struct AppConfig :public YamlConfig {
    int ThreadNum{ 1 };
};
struct CommonConfig : public AppConfig {
    std::string Ip{ "127.0.0.1" };
    int Port{ 8080 };
};
struct LoginConfig : public CommonConfig {
    std::string UrlLogin;
};
struct RobotConfig : public AppConfig { };

```

为login工程属性编写了一个结构LoginConfig，基于CommonConfig结构。这里编写了好几个基类，这样做的好处在于，将来可能还有其他的结构，这样可以将一些共用的属性集合起来。LoginConfig拥有3个属性：IP、Port和UrlLogin，还有一个来自基类的ThreadNum属性，用于定义进程中启动几个线程。而RobotConfig只有ThreadNum属性。

在工程中，读取YAML文件的Yaml类，定义如下：

```

class Yaml : public Singleton<Yaml> {
public:
    Yaml();
    YamlConfig* GetConfig(APP_TYPE appType);
private:
    void LoadConfig(APP_TYPE appType, YAML::Node& config);
private:
    std::map<APP_TYPE, YamlConfig*> _configs;
};

```

Yaml类实例全局唯一，是一个单例，该类中值得注意的是std::map<APP_TYPE, YamlConfig*>字典和GetConfig函数，该函数是根据服务进程的类型返回一个YamlConfig结构指针。它的实现代码如下：

```

Yaml::Yaml() {
    auto pResPath = ResPath::GetInstance();
    if (pResPath == nullptr) {
        std::cout << "yaml awake failed. can't get ResPath." << st
        return;
    }
}

```

```

    }
    const std::string path = pResPath->FindResPath("/engine.yaml")
    YAML::Node config = YAML::LoadFile(path);
    LoadConfig(APP_TYPE::APP_LOGIN, config);
    LoadConfig(APP_TYPE::APP_ROBOT, config);
}
YamlConfig* Yaml::GetConfig(const APP_TYPE appType) {
    if (_configs.find(appType) != _configs.end()) {
        return _configs[appType];
    }
    return nullptr;
}

```

Yaml类构造函数里实现了对YAML文件的分析。函数GetConfig从字典std::map<APP_TYPE, YamlConfig*>中取出数据。在读取YAML文件时使用了一个新的工具类ResPath，用于找到配置文件所在的位置，该类的定义保存在res_path.h文件中。其主要逻辑分成两步：

(1) 从环境变量中取出ENGINE_RES_PATH属性。我们可以在系统的环境变量中指定游戏res的目录。

(2) 如果没有环境变量，就直接从执行文件的上一层目录内找到res目录。

ResPath类的具体代码可以查看源代码，这里不再过多地阐述。读出文件之后使用库函数YAML::LoadFile将文件加载为YAML库中的Node格式，再对其进行详细处理。文件加载分析函数如下：

```

void Yaml::LoadConfig(const APP_TYPE appType, YAML::Node& config)
{
    std::string appTypeName = AppTypeMgr::GetInstance()->GetAppNam
    YAML::Node node = config[appTypeName];
    if (node == nullptr)
        return;
    AppConfig* pAppConfig;
    switch (appType) {
    case APP_LOGIN: {
        auto pConfig = new LoginConfig();
        pConfig->UrlLogin = node["url_login"].as<std::string>();
        pAppConfig = pConfig;
        break;
    }
    case APP_ROBOT: {

```

```

        auto pConfig = new RobotConfig();
        pAppConfig = pConfig;
        break;
    }
    default: {
        pAppConfig = new CommonConfig();
        break;
    }
}
auto pCommon = dynamic_cast<CommonConfig*>(pAppConfig);
if (pCommon != nullptr) {
    pCommon->Ip = node["ip"].as<std::string>();
    pCommon->Port = node["port"].as<int>();
}
pAppConfig->ThreadNum = node["thread_num"].as<int>();
_configs.insert(std::make_pair(appType, pAppConfig));
}

```

从YAML文件中可以看出，每一个进程都有一个对应的结构。login工程对应LoginConfig结构，该结构基于AppConfig。从代码中可以看出，将YAML文件中的url_login属性读取出来，赋值给LoginConfig的UrlLogin属性，这是账号登录时需要调用的验证URL地址。之前是固定在代码中的，现在修改如下：

```

void HttpRequestAccount::AwakeFromPool(std::string account, std::s
...
auto pYaml = Yaml::GetInstance();
const auto pLoginConfig = dynamic_cast<LoginConfig*>(pYaml->Ge
_url = pLoginConfig->UrlLogin;
}

```

这样，在每次账号请求时使用的是在YAML文件中配置的地址。修改账号验证的URL不需要再重新编译代码了。除了登录的账号验证URL之外，我们还配置了进程监听的IP和端口。在login进程初始化时，从YAML文件中读取地址与端口，作为NetworkListen类的参数。其实现代码如下：

```

void LoginApp::InitApp() {
    auto pYaml = Yaml::GetInstance();
    const auto pLoginConfig = dynamic_cast<LoginConfig*>(pYaml->Ge

```

```
        _pThreadMgr->CreateComponent<NetworkListen>(pLoginConfig->Ip,  
        ...  
    }
```

在robots工程中，连接login时，可以从YAML配置中读取出login的连接地址。

```
void Robot::AwakeFromPool(std::string account) {  
    ...  
    auto pYaml = Yaml::GetInstance();  
    const auto pLoginConfig = dynamic_cast<LoginConfig*>(pYaml->Get  
        this->Connect(pLoginConfig->Ip, pLoginConfig->Port);  
}
```

本小节简单地说明了YAML的配置文件及其读取方式，在后续的章节中，我们还会对engine.yaml文件做大量的配置，将所有的服务端启动参数放在engine.yaml文件中，以避免分散配置带来的麻烦。

6.4.3 合并线程

在engine.yaml文件中有一个配置项为thread_num，设置了当前进程中的线程数量。其生成线程的代码实现如下：

```
ServerApp::ServerApp(APP_TYPE appType) {  
    ...  
    const auto pLoginConfig = dynamic_cast<AppConfig*>(Yaml::GetIn  
for (int i = 0; i < pLoginConfig->ThreadNum; i++) {  
        _pThreadMgr->CreateThread();  
    }  
    ...  
}
```

启动进程的时候会根据这个值创建相对应的线程实例。当我们将线程配置为0时，这表示不使用线程，这时主线程就变成了进程中的唯一线程。因为程序不再启动多线程，所以整个程序就变为单线程运行。在某些情况下，单线程便于我们查找程序中的Bug，为了在调试时具有这个功能，因而工程中提供了这类兼容单线程和多线程的代码。

在协议分发时，协议是分发到各个线程中的。当没有多线程时，协议直接放到主线程的EntitySystem中处理。因为是单线程，实体和组件对象都需要放在主线程中。和Thread类一样，主线程中的ThreadMgr也是一个EntitySystem，所以完全可以实现这样的转换。下面来看Update函数对于创建实体的处理方式：

```
void ThreadMgr::Update() {  
    _create_lock.lock();  
    if (_createPackets.CanSwap()) {  
        _createPackets.Swap();  
    }  
    _create_lock.unlock();  
    auto pList = _createPackets.GetReaderCache();  
    for (auto iter = pList->begin(); iter != pList->end(); ++iter)  
        const auto packet = (*iter);  
    if (_threads.size() > 0) {  
        if (_threadIndex >= _threads.size())  
            _threadIndex = 0;  
        _threads[_threadIndex]->AddPacketToList(packet);  
    }
```

```
        _threadIndex++;
    } else {
        // 单线程
        AddPacketToList(packet);
    }
}
pList->clear();
EntitySystem::Update();
}
```

集合createPackets存放着跨线程创建组件的协议，在动态创建类的时候，如果是多线程，就分发到某个子线程中处理，但是如果是单线程，就直接由当前主线程的EntitySystem处理。这时生成的类自然就在主线程中。

在多线程中，ThreadMgr类管理的只是一些公用的组件。在单线程中，逻辑组件也被放置到了主线程中。如果将robots工程的线程配置数变为0，那么所有的Robot类会被放置在主线程中。

在使用线程配置时，看上去合并线程和扩展线程都相当方便。因为框架严格遵循了Actor的原则，使得组件与组件之间的耦合几乎为零，两个交集为零的实体，不论放在哪里都无所谓。

Account类需要HttpRequestAccount类的结果，但是Account类并不依赖于HttpRequestAccount类，也和HttpRequestAccount实例没有任何关系，所以Account类是否与HttpRequestAccount类放在同一个线程显然并不重要。而这些类到底是放在主线程还是子线程也不重要。只要它处于某个EntitySystem中，就会起到自己应尽的责任。

log4cplus 日志

6.4节讲解了服务端的配置，本节将讲解服务端的日志。服务端有两种日志：一种是常规日志，用于后台的打印输出，以便在需要时查看；另一种是对游戏逻辑内的物品进行跟踪的日志，这类日志需要存入数据，以方便进行物品跟踪，其数据也可作为分析策划所用。本节阐述的日志是第一类。

日志系统我们采用名为log4cplus的第三方库。用过j2EE开发的人都非常熟悉，Apache有一个比较常用的日志系统log4j，log4cplus可以视为log4j的C++版本，其使用灵活，线程安全，下载地址为<https://github.com/log4cplus/log4cplus/releases>。我们下载一个较新的版本log4cplus-2.0.4.tar.gz，该下载文件已经放在本书源代码dependence目录下了。

6.5.1 log4cplus的编译安装

在本书源代码根目录的lib目录下已经准备了编译好的Windows版的库，编译版本为Visual Studio 2019 debug x86，如果读者对编译部分不感兴趣，那么可以跳过这一节。

关键点1：编译Linux版

(1) 解压缩进入目录中，执行./configure：

```
[root@localhost dependence]# tar xf log4cplus-2.0.4.tar.gz
[root@localhost dependence]# cd log4cplus-2.0.4/
[root@localhost log4cplus-2.0.4]# ./configure
```

(2) 执行make，编译工程，安装到系统目录下：

```
[root@localhost log4cplus-2.0.4]# cmake3 -DCMAKE_CXX_FLAGS="-std=c
[root@localhost log4cplus-2.0.3]# make
[root@localhost log4cplus-2.0.4]# make install
```

(3) 工程中使用log4cplus时，修改CMakeLists.txt文件的CMAKE_CXX_FLAGS属性，增加-llog4cplus参数。

(4) 修改环境变量：

```
[root@localhost log4cplus-2.0.4]# find /usr/ -name liblog4cplus.so
/usr/local/lib64/liblog4cplus.so
```

安装完成之后，看一下SO文件的位置，位于/usr/local/lib64下。这个路径没有在环境变量下。使用vim打开“~/.bash_profile”，在LD_LIBRARY_PATH中增加/usr/local/lib64。

关键点2：编译Windows版

(1) 解压缩log4cplus-2.0.4.tar.gz，在msvc14目录下打开log4cplus.sln解决方案。

其中log4cplus.vcxproj编译出来的是一个动态库，log4cplusS.vcxproj为静态库，这里选用静态库。同样，生成的lib放在根目录的libs\windows目录中。在Windows下编译需要注意运行库和字符集的设置，我们的原工程采用的是Unicode字符集，在编译log4cplus时，字符集需要修改为“使用Unicode字符集”。

(2) 增加Include文件。与YAML一样，将log4cplus-2.0.4\include下的log4cplus复制到根目录的include\windows目录下。

(3) 这一步与YAML的步骤一致。在login和Robot工程中打开“属性”页，选择“连接器” → “输入” → “附加依赖项”，在最后加上log4cplusSD.lib。

6.5.2 配置文件

在使用log4cplus时，首先需要为它编写一个配置文件。本节工程的源代码位于06_07_log4目录，在res目录下创建一个log4目录，编写一个log4_login.properties的配置文件，这是login进程的配置文件。log4_login.properties的内容如下：

```
#不能有空行，至少以#开头
log4cplus.rootLogger=DEBUG, ALL_MSGS_TO_CONSOLE, ALL_MSGS_TO_FILE
#
log4cplus.appender.ALL_MSGS_TO_CONSOLE=log4cplus::ConsoleAppender
log4cplus.appender.ALL_MSGS_TO_CONSOLE.layout=log4cplus::PatternLa
log4cplus.appender.ALL_MSGS_TO_CONSOLE.layout.ConversionPattern=%d
#
log4cplus.appender.ALL_MSGS_TO_FILE=log4cplus::RollingFileAppender
log4cplus.appender.ALL_MSGS_TO_FILE.MaxFileSize=1MB
log4cplus.appender.ALL_MSGS_TO_FILE.MaxBackupIndex=100
log4cplus.appender.ALL_MSGS_TO_FILE.layout=log4cplus::PatternLayou
log4cplus.appender.ALL_MSGS_TO_FILE.layout.ConversionPattern=%d ->
log4cplus.appender.ALL_MSGS_TO_FILE.File=./log4_login.log
```

解释一下此文件的含义。

(1) log4的输出分成几个等级，包括TRACE、DEBUG、INFO、WARNING、ERROR、FATAL，一个等级比一个等级高。如果用圈层结构进行说明，那么TRACE层位于最外圈，包括FATAL；而FATAL层不包括TRACE。在上面的文件中，指定以DEBUG等级输出日志，也就是说除了TRACE之外，其他的日志都要打印。当配置指定为DEBUG层级时，高于DEBUG的所有宏是有效的，TRACE是无效的。如果在配置中写入的是INFO，那么LOG_DEBUG的调用也是无效的。在正式上线时，可以屏蔽掉调试阶段的DEBUG层级的打印数据。

(2) log4有几种记录方式，如控制台或文件，每一种记录方式都需要用到Logger实例。在上面的文件中，我们指定了log4cplus.rootLogger。

```
log4cplus.rootLogger=DEBUG, ALL_MSGS_TO_CONSOLE, ALL_MSGS_TO_FILE
```

该Logger是默认的Logger，每一个Logger实例写入的文件不同。假设还需要写一个Logger，可以这样定义：

```
log4cplus.Logger.GM=DEBUG, GM_MSGS_TO_FILE
```

这样，就有一个新的仅记录GM指令的日志了。

(3) Appender是对Logger的一个补充，用它来指定一些属性，例如以特定格式输出到终端，这些终端是指定屏幕或文件等。

在上面的文件中，对log4cplus.appender指定了ALL_MSGS_TO_CONSOLE和ALL_MSGS_TO_FILE。我们对ALL_MSGS_TO_CONSOLE和ALL_MSGS_TO_FILE进行不同格式的配置：

```
log4cplus.appender.ALL_MSGS_TO_CONSOLE=log4cplus::ConsoleAppender  
log4cplus.appender.ALL_MSGS_TO_FILE=log4cplus::RollingFileAppender
```

终端有几种方式，包括ConsoleAppender（控制台）、RollingFileAppender（限制大小的文件）和DailyRollingFileAppender（按计划产生文件）。无论哪一种都需要配置排版布局，即消息如何展示出来，也就是属性layout。layout常用的是log4cplus::PatternLayout属性，可以在打印数据中使用表示符，就如同C++的printf中的%d之类的标识。

log4cplus::PatternLayout是一个模板布局。设置了PatternLayout，就必须设置ConversionPattern，设置如下：

```
log4cplus.appender.GM_MSGS_TO_FILE.layout=log4cplus::PatternLayout  
log4cplus.appender.GM_MSGS_TO_FILE.layout.ConversionPattern=%d ->
```

在输出格式时使用了%d，表示的是UTC时间；如果要使用当前本地时间，那么可以用%D。%p表示输出层级，%m表示输出打印数据，%n是换行符。除此之外，还有一些其他的常用表示符，%F输出文件名，%L输出

文件行。

(4) 输出到文件时有两个常用配置：一个是RollingFileAppender；另一个是DailyRollingFileAppender。DailyRollingFileAppender会每天生成一个文件。在配置这两个属性时用到了MaxBackupIndex属性，该属性表示最大存在的文件数。当存储的文件数目超过MaxBackupIndex+1时，会删除最早生成的文件，保证整个文件数目等于MaxBackupIndex，确保日志系统产生的文件不会越堆越多，超过硬盘的大小。

在RollingFileAppender配置中，MaxFileSize属性用于限制文件大小。

在DailyRollingFileAppender中有两个特有的属性：

- Schedule：可用值为MONTHLY、WEEKLY、DAILY、TWICE_DAILY、HOURLY、MINUTELY。
- DatePattern：DatePattern格式化之后的文本作为文件名字的后缀。例如当指定Schedule为DAILY时，DatePattern可以设置为%Y-%m-%d，表示使用了年月日来作为日志文件的名字。

(5) Filter过滤器，简单来说过滤器可以为我们过滤掉不重要的数据，而留下重要的数据。修改log4cplus.rootLogger如下：

```
log4cplus.rootLogger=DEBUG, ERROR_TO_FILE, ALL_MSGS_TO_CONSOLE, AL
#
log4cplus.appender.ERROR_TO_FILE=log4cplus::RollingFileAppender
log4cplus.appender.ERROR_TO_FILE.MaxFileSize=1MB
log4cplus.appender.ERROR_TO_FILE.MaxBackupIndex=100
log4cplus.appender.ERROR_TO_FILE.File=./log4_login_error.log
log4cplus.appender.ERROR_TO_FILE.filters.1=log4cplus::spi::LogLeve
log4cplus.appender.ERROR_TO_FILE.filters.1.LogLevelToMatch=ERROR
log4cplus.appender.ERROR_TO_FILE.filters.1.AcceptOnMatch=true
log4cplus.appender.ERROR_TO_FILE.filters.2=log4cplus::spi::DenyAll
log4cplus.appender.ERROR_TO_FILE.layout=log4cplus::PatternLayout
log4cplus.appender.ERROR_TO_FILE.layout.ConversionPattern=%d -> [%
```

上面的配置中增加了一个名为ERROR_TO_FILE的新输出，在这个输

出中使用了filter，目的是过滤其他打印日志，只留下ERROR层级之上的日志。在打印日志时，同时把代码的文件和行号显示出来，即ConversionPattern的%l。

过滤信息使用filter属性，过滤方式为log4cplus::spi::LogLevelMatchFilter，按LogLevel进行匹配过滤，条件是上面设置的LogLevelToMatch=ERROR。一个filter可以有多个条件，filter的执行顺序是后写的条件会先被执行，以上面的代码为例，在执行时会先执行filters.2的过滤条件，关闭所有过滤器，然后执行filters.1匹配ERROR信息。filter的匹配方式有以下几种：

- LogLevelMatchFilter：匹配一个LogLevel，过滤条件包括LogLevelToMatch和AcceptOnMatch（true|false）。
- LogLevelRangeFilter：匹配一个范围，过滤条件包括LogLevelMin、LogLevelMax和AcceptOnMatch。
- StringMatchFilter：匹配LogLevel字符串，过滤条件包括StringToMatch和AcceptOnMatch。

对于log4cplus，我们有一个大概的印象即可，关于log4cplus的更多说明，在log4cplus中有写好的注释和API说明。也可以参考其官网说明，这里就不再一一列举了。

6.5.3 使用log4cplus

讲完了定义格式，我们来看看如何使用这个配置。为了在代码中调用log4cplus，工程中专门编写了一个Log4类。在libserver工程的log4.h文件中的定义如下：

```
class Log4 : public Singleton<Log4> {
public:
    Log4(int appType);
    ~Log4();
    static std::string GetMsgIdName(Proto::MsgId msgId);
protected:
    void DebugInfo(log4cplus::Logger logger) const;
private:
    APP_TYPE _appType{ APP_TYPE::APP_None };
};
```

在该类生成时完成了初始化，初始化代码如下：

```
Log4::Log4(int appType) {
    _appType = static_cast<APP_TYPE>(appType);
    auto pResPath = ResPath::GetInstance();
    auto pAppTypeMgr = AppTypeMgr::GetInstance();
    const std::string filename = strutil::format("/log4/log4_%s.pr
    std::string filePath = pResPath->FindResPath(filename);
    if (filePath.empty()) {
        std::cout << " !!!!! log4 properties not found! filename:"
        return;
    }
    log4cplus::initialize();
    const log4cplus::tstring configFile = LOG4CPLUS_STRING_TO_TSTR
    log4cplus::PropertyConfigurator config(configFile);
    config.configure();
    LOG_DEBUG("Log4::Initialize is Ok.");
}
```

Log4类的初始化工作主要是读取properties文件与调用log4cplus::initialize函数。在上面的代码中，根据传入的进程类型读取了不同的配置文件进行初始化。初始化完成之后，类的使用十分

简单，只需要调用宏定义即可。宏定义在log4_help.h文件中，主要有LOG_TRACE、LOG_DEBUG、LOG_INFO、LOG_WARN和LOG_ERROR等常规宏。鉴于篇幅，这里就不一一列举了。

工程中为每种LogLevel写了一个宏。为了便于区分，警告LOG_WARN和出错LOG_ERROR这两个宏有不同的颜色。鉴于Windows和Linux下的控制台前景色调用方式不同，对于颜色部分写了两个版本。在Linux下的颜色，本书专门编写了一个脚本，位于本书源代码的linux目录下，名为color.sh。可以看到在Linux下，控制台可以打印出一些颜色与背景。读者可以自行执行查看。

有了Log4类之后，之前使用std::cout的地方都可以使用相应的宏来替换。来看一个应用场景，编译执行本小节的源代码，在使用命令行输入时，输入一个随机串“lfdsf”，程序就会提示输入出错的信息。因为将std::cout修改为LOG_ERROR时，当输入错误时会出现红色的提示，非常醒目。

Log4类在控制台输出时显然比原生的std::cout更有优势，可以更容易分辨日志的类型。除此之外，Log4还留下了一个日志文件，方便在进程运行时留下日志，以便以后查用。

当框架越来越大时，打印信息就变得越来越臃肿，为了在这些打印中找到关键信息，颜色输出就会变得很重要。现在执行login进程，打印信息如下：

```
[root@localhost bin]# ./logind
app type:8 id:1
ENGINE_RES_PATH=/root/game/06_07_log4/res
[log4] root appender name:ERROR_TO_FILE
[log4] root appender name:ALL_MSGS_TO_CONSOLE
[log4] root appender name:ALL_MSGS_TO_FILE
[DEBUG] - Log4::Initialize is Ok.
[DEBUG] - Yaml awake is Ok.
[INFO] - epoll model. listen 127.0.0.1:5550
```

其中，[INFO]调用的是LOG_INFO，而[DEBUG]调用的是LOG_DEBUG。

6.6

总结

本章在原有框架的基础上做了一个重要的重构，即引入了ECS设计原则，将工程优化为组件与实体的框架。组件与实体的设计方式脱离了继承的桎梏，结合Actor的设计思路，使得组件与组件之间解藕。

同时，本章为整个框架增加了两个重要的基础功能：增加了YAML文件，对服务端进行配置；增加了log4cplus进行日志打印。第7章将用组件的方式实现一系列非常重要的功能——数据库的相关操作。

第7章

MySQL数据库

存储是网络游戏开发中不可缺少的部分。在存储方面，不同的游戏公司采用不同的数据库，有使用MySQL的，也有使用MongoDB的，本书中采用主流的MySQL数据库。本章主要介绍C++与MySQL之间的交互，本章包括以下内容：

- 对数据库进行读、写操作。
- 对数据库进行更新。

MySQL Connector/C

调用MySQL时，使用的第三方库为MySQL的Connector/C。本书没有使用Connector/C++。Connector/C++与Connector/C的区别在于C++版本采用了类的方式操作数据库，而C语言版本只提供API，根据框架的自身特点，C语言版本可能更适合一些，而且它更方便、高效。

关键点1: Linux系统配置

在Linux下不需要进行任何操作，在安装数据库环境时已经在系统中安装了开发环境，只需要在使用时修改CMakeLists.txt文件即可。本节的源代码位于07_01_db目录，打开dbmgr工程中的CMakeLists.txt，新增一行：

```
target_link_libraries(${MyProjectName} -L/usr/lib64/mysql -lmysqlc)
```

关键点2: Windows系统配置

在Windows系统中，使用该库的简单方式是直接在本机上安装MySQL 5.7，因为本书在Linux系统中使用的是5.X版本，所以在Windows系统中也使用5.X版本，以免接口不一致。安装MySQL之后，头文件和需要的动态库或静态库都有了，可以直接使用libmysql.lib与libmysql.dll两个文件。如果不想在本地安装MySQL，又想使用静态库，那么可以自行编译。MySQL自身使用的是Connector/C，所以编译最新的Connector/C库可以从MySQL源代码中进行编译。

本书已经在lib目录中准备好了Windows版本的动态库，可以直接使用。使用时需要修改系统的环境变量Path，指向本书源代码根目录下的lib\windows目录，否则在运行dbmgr时会提示找不到libmysql.dll。

连接时使用的函数说明

要使用MySQL Connector/C，就需要对它的基础API有一个了解，本节先来了解需要用到函数。

函数1：初始化库

```
int mysql_library_init(int argc, char **argv, char **groups)
void mysql_library_end(void)
```

函数mysql_library_init和mysql_library_end是一对用于初始化和释放的函数。

mysql_library_init需要在调用其他函数之前调用，用来初始化MySQL库。而mysql_library_end调用可以释放内存，在调用mysql_close之后调用，以帮助释放内存数据。

函数2：创建一个数据库对象指针

```
MYSQL *mysql_init(MYSQL *mysql)
```

该函数有两种用法：一种是传入MySQL对象；另一种是传入nullptr参数。两种调用都会返回MySQL对象指针，如果有传入值，传入返回的对象就是同一个；如果传入的参数为nullptr，就会返回一个实例。这两者的差别在于谁来管理MySQL对象的实例，如果是在外部创建的对

象，在调用mysql_close之后就需要手动释放该对象，以免造成内存泄漏。如果是由mysql_init创建的对象，就只需要关闭连接。

函数3：销毁现有对象指针

```
void mysql_close(MYSQL *mysql)
```

该函数用于关闭一个连接，如果MySQL实例是库生成的，该函数就会同时释放该对象。

函数4：连接函数

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const ch
```

在这个函数中，第5个参数为想要连接的数据库的名字，可以为nullptr，表示只是想产生一个连接，但不选择具体数据库，后续调用其他函数选定具体的数据库。

当我们连接一个给定名字的数据库时，如果这个数据库不存在，就会出现错误，而MySQL对象也不可再用，需要关闭。鉴于这种情况，在第一次连接时就可以将数据库名设为nullptr，让它进行一个默认的连接，再调用mysql_select_db函数进行数据库的选择，若选择失败，则说明所需的数据库不存在，这时可以创建需要的数据库。

在连接函数中，最后一个参数client_flag是一系列的常量，在本书的示例程序中使用了CLIENT_FOUND_ROWS，官方的解释如下：

```
CLIENT_FOUND_ROWS: Return the number of found (matched) rows, not
```

即返回匹配的行数，而不是修改行数。

若函数mysql_real_connect的返回值是nullptr，则连接失败；若成功，则返回值为MySQL的对象指针。

函数5：设置属性

```
int mysql_options(MYSQL *mysql, enum mysql_option option, const vo
```

该函数是在mysql_init之后、mysql_real_connect之前调用的，对MySQL对象进行一些属性设置。在本书的框架中用到了两个常用的属性：

- MYSQL_OPT_CONNECT_TIMEOUT：设置连接超时。
- MYSQL_OPT_RECONNECT：是否自动连接。

更多的枚举可以在头文件mysql.h中查看。

函数6：选择数据库

```
int mysql_select_db(MYSQL *mysql, const char *db)
```

该函数输入一个需要连接的数据库名。返回值0表示成功，非0即为出错编号。该函数指定一个数据库作为当前选中的数据库。在调用该函数时，如果用户没有指定数据库的权限就会出错。

函数7：错误代码

```
unsigned int mysql_errno(MYSQL *mysql)
```

在每一个函数调用之后，如果出现错误，就可以通过调用函数mysql_errno得到当前错误的编号。

函数8：ping函数

```
int mysql_ping(MYSQL *mysql)
```

该函数检查连接是否处于正常工作中。

在函数mysql_options的设置中，可以打开自动连接的开关，当网络断开时，调用mysql_ping会自动重新连接。

数据库连接组件

介绍完了主要函数，接下来看看框架是如何应用这些API函数来连接MySQL的。用一个实际例子说明，本节源代码位于07_01_db目录中。在源代码中新增一个dbmgr工程，用来处理与数据库相关的操作。

7.3.1 MysqlConnector组件

在dbmgr工程中新增一个MysqlConnector类，提供一些数据库操作，其定义可以在mysql_connector.h文件中找到。

```
class MysqlConnector : public MysqlBase, public Entity<MysqlConnec
class MysqlBase {
public:
    bool ConnectInit();
    virtual void Disconnect();
    ...
protected:
    DBConfig _dbCfg;
    MYSQL* _pMysql{ nullptr };
    MYSQL_RES* _pMysqlRes{ nullptr };
    ...
};
```

MysqlConnector类基于MysqlBase类，该类中实现了IMessageSystem接口，处理诸如查询角色或创建角色这样的协议。MysqlBase类则提供了连接MySQL数据库以及读取与查询的一些基本功能。下面详细分析一下代码。

7.3.2 连接数据库

对于连接数据库来说，首先需要连接给定名字的数据库，这部分功能是在初始化时完成的：

```
void MysqlConnector::AwakeFromPool() {
    auto pYaml = Yaml::GetInstance();
    auto pConfig = pYaml->GetConfig(APP_DB_MGR);
    auto pDbCfig = dynamic_cast<DBMgrConfig*>(pConfig);
    _pDbCfig = pDbCfig->GetDBConfig(DBMgrConfig::DBTypeMysql);
    if (_pDbCfig == nullptr) {
        LOG_ERROR("Failed to get mysql config.");
        return;
    }
    Connect();
}
```

当MysqlConnector类从对象池中唤醒时，它会首先执行AwakeFromPool函数，在执行该函数时，从配置数据中读出数据库的配置信息，配置数据位于engine.yaml文件中。在dbmgr的配置中有两组数据：一组是关于Redis的；另一组是关于MySQL的。这些配置包括数据库IP、端口、数据库名、字符集等，在本例中，取出MySQL相关的配置并保存在MysqlConnector类中。初始化函数最后调用了函数Connect进行连接，调用了基类的连接函数MysqlBase::ConnectInit。该函数进行了一些连接前的基本操作，初始化操作成功之后，调用mysql_real_connect函数发起一个连接。如果经检查没有错误，就认为连接成功。

在基类中初始化MysqlBase::ConnectInit函数的实现代码如下：

```
bool MysqlBase::ConnectInit() {
    Disconnect();
    _pMysql = mysql_init(nullptr);
    if (_pMysql == nullptr) {
        CheckMysqlError();
        LOG_ERROR("mysql_init == nullptr");
        return false;
    }
    // 如果已有一个连接
    // 由库创建出MySQL对象
```

```
// 设置连接等待时间，这里设置的是10秒
int outtime = 10;
mysql_options(_pMysql, MYSQL_OPT_CONNECT_TIMEOUT, &outtime);
bool reConnect = false; // 不自动重连
mysql_options(_pMysql, MYSQL_OPT_RECONNECT, &reConnect);
return true;
}
```

在基类函数中，创建了MySQL指针并对它进行了属性的设置，在此设置了两个属性：一个是连接时的超时设定；另一个用于设定是否自动重连。本例中采用了不自动重连，手动定时查看的方式。

7.3.3 关闭连接

如果在连接过程中出现异常，就可以调用Disconnect函数关闭已生成的连接，以便重新连接。关闭连接函数的代码如下：

```
void MysqlConnector::Disconnect() {
    CleanStmts();
    MysqlBase::Disconnect();
}
void MysqlBase::Disconnect() {
    ...
    if (_pMysql != nullptr) {
        mysql_close(_pMysql);
        _pMysql = nullptr;
    }
}
```

在关闭连接时调用了库函数mysql_close以释放内存。虽然在游戏框架上使用了多线程，但是对于一个MysqlConnector对象而言，它遵照Actor原则，代码没有耦合性，因而线程是安全的。如果需要，那么可以启动多个线程来执行数据库操作，每一个线程都生成一个独立的MysqlConnector实例，每一个线程都相当于一个MySQL客户端，互不影响。

写入数据时使用的函数说明

厘清了数据库的连接与关闭操作之后，再来说说对数据库的插入和更改操作。MySQL库同样提供了一系列函数供我们使用，首先要了解“预处理”这个概念。

什么是预处理呢？

对于需要多次执行的语句，预处理是一种非常高效的方式，其原理是一次生成语句，每次执行时传入参数，以减少数据的传递。一次生成语句的好处是不用每次对SQL语句进行解析，极大地提高了效率。预处理就像一个函数，每次执行时只需要填入不同的参数，就能得到不同的结果。几个重要的预处理函数如下：

函数1：创建一个预处理

```
MYSQL_STMT *mysql_stmt_init(MYSQL *mysql)
```

MySQL库的预处理使用一个名为MYSQL_STMT的结构，调用该函数即可创建一个MYSQL_STMT指针，返回值不是nullptr时则为成功。

函数2：销毁一个预处理

```
my_bool mysql_stmt_close(MYSQL_STMT *stmt)
```

该函数销毁传入的MYSQL_STMT指针。

函数3：初始化预处理

```
int mysql_stmt_prepare(MYSQL_STMT *stmt, const char *stmt_str, uns
```

该函数将一个SQL语句写入预处理结构中，第二个参数即为SQL语句，该语句不需要有结束符分号，即不需要符号“;”。传入的SQL语句在参数的位置用“?”代替。

调用该函数，返回值为非0时，表示有错误，可以用mysql_stmt_error函数查看错误编码。

函数4：出错检查

```
const char *mysql_stmt_error(MYSQL_STMT *stmt)
```

一旦发现某个预处理函数有异常或出错，就可以通过调用该函数来获取错误描述。

函数5：绑定参数

```
my_bool mysql_stmt_bind_param(MYSQL_STMT *stmt, MYSQL_BIND *bind)
```

前面传入的SQL语句中，关于动态参数的部分是用“?”来代替的。而函数mysql_stmt_bind_param是专门为这些“?”准备的，利用MYSQL_BIND结构提供参数。像函数一样，一个预处理在实际执行阶段需要绑定实际的参数。

函数6：执行

```
int mysql_stmt_execute(MYSQL_STMT *stmt)
```

绑定完参数之后的预处理指针就可以调用执行函数来执行。如果返回结果不为0，就表示有错误。

函数7：获取执行结果个数

```
my_ulonglong mysql_stmt_affected_rows(MYSQL_STMT *stmt)
```

预处理执行之后，我们可以通过该函数来获取结果行的个数，即当前执行的预处理结果中有多少行数据。

写入数据示例

一个完整的插入数据的步骤是这样的：首先根据SQL语句创建预处理，在需要使用时传入参数，然后调用执行函数得到结果。

以创建角色为例，先创建“创建角色的预处理”，并将这个预处理实例存入一个字典中，便于每次使用时快速调用。当创建角色的协议发送到dbmgr进程时，调用预处理实例并传入参数，这些参数是玩家名字和玩家的初始数据。最后，调用执行函数写入数据库中。下面用一个实际例子加以说明，本节的源代码位于07_01_db目录中。

7.5.1 创建预处理

在MysqlConnector类创建之初就初始化两个预处理：一个是插入玩家数据的预处理；另一个用于更新玩家数据。

```
void MysqlConnector::InitStmts() {
    DatabaseStmt* stmt = CreateStmt("insert into player ( sn, acco
_mapStmt.insert(std::make_pair(DatabaseStmtKey::StmtCreate, st
stmt = CreateStmt("update player set base=?, misc=?,savetime=n
_mapStmt.insert(std::make_pair(DatabaseStmtKey::StmtSave, stmt
}
```

代码中为两个预处理绑定了两个枚举：DatabaseStmtKey：： StmtCreate用于创建角色；DatabaseStmtKey：： StmtSave用于更新角色数据。同时，将生成的对象存入字典中，以便使用时方便索引。在上面的代码中，预处理被写入了一个封装好DatabaseStmt的结构中。DatabaseStmt的原型如下：

```
struct DatabaseStmt {
    MYSQL_STMT *stmt{ nullptr };
    MYSQL_BIND *bind{ nullptr };
    char *bind_buffer{ nullptr };
    int bind_index;
    int bind_buffer_index;
};
```

在这个结构体中存储了预处理结构MYSQL_STMT的指针以及一个MYSQL_BIND的指针，该指针是为了绑定参数而服务的，是一个数组。表7-1列举了DatabaseStmt结构的每一个值的定义。

表7-1 DatabaseStmt结构表

参 数	说 明
stmt	MYSQL_STMT 指针
bind	MYSQL_BIND 指针数组
bind_buffer	该属性是一个内存地址块，这个内存地址块是为了存储绑定数据的，给了它一个宏定义长度：#define MAX_BIND_BUFFER 40960。绑定的数据，即使是二进制数据，总大小也不会超过 40960
bind_buffer_index	内存块中当前的读写地址位置偏移值
bind_index	bind_index 为当前 bind 的位置偏移值。bind 数组每增加一个数据，bind_index 自增一次

生成DatabaseStmt结构的代码如下：

```
DatabaseStmt* MysqlConnector::CreateStmt(const char* sql) const {
    int str_len = strlen(sql);
    DatabaseStmt* stmt = new DatabaseStmt();
    int param_count = 0;
    stmt->stmt = mysql_stmt_init(_pMysql); // 从MySQL库中创建MYSQL_
    if (mysql_stmt_prepare(stmt->stmt, sql, str_len) != 0) {
        return nullptr; // 失败
    }
    for (int i = 0; i < str_len; i++) {
        if ((sql[i] == '?') || (sql[i] == '@'))
            param_count++; // 统计参数个数
    }
    if (param_count > 0) {
        stmt->bind = new MYSQL_BIND[param_count]; // 为每个参数创建MYSQL_
        memset(stmt->bind, 0, sizeof(MYSQL_BIND) * param_count);
        stmt->bind_buffer = new char[MAX_BIND_BUFFER]; // 数据备用空
    } else {
        stmt->bind = nullptr;
        stmt->bind_buffer = nullptr;
    }
    return stmt;
}
```

}

从上面的代码中不难看出，有多少个参数就要生成多少个MYSQL_BIND数据结构。图7-1展示了结构的内存数据。例如INSERT INTO tabel1 VALUES (? , ? , ?)，参数个数为3，所以这里DatabaseStmt中的属性bind是一个MYSQL_BIND[3]的数组指针。在生成DatabaseStmt结构时，在结构中生成了一块用于缓存数据的buffer，用于存放参数。

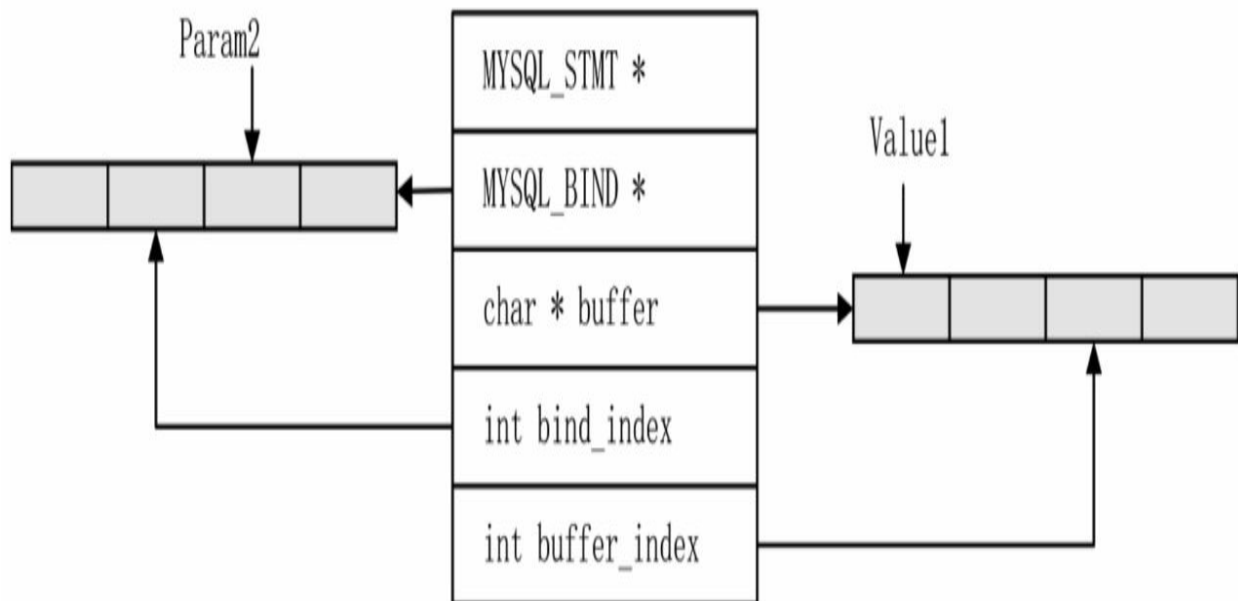


图7-1 DatabaseStmt内存结构

下面用一个实际例子来说明结构中这些数据的用途。

7.5.2 用预处理创建角色

创建角色时，创建角色的协议被MysqlConnector类捕捉到，它的处理函数为HandleCreatePlayer，其实现如下：

```
void MysqlConnector::HandleCreatePlayer(Packet* pPacket) {
    auto protoCreate = pPacket->ParseToProto<Proto::CreatePlayerTo>();
    auto protoPlayer = protoCreate.player();
    DatabaseStmt* stmt = GetStmt(DatabaseStmtKey::StmtCreate);
    if (stmt == nullptr)
        return;

    ...
    // create
    ClearStmtParam(stmt);
    AddParamUint64(stmt, protoPlayer.sn());
    AddParamStr(stmt, protoCreate.account().c_str());
    AddParamStr(stmt, protoPlayer.name().c_str());
    ...
    if (ExecuteStmt(stmt)) {
        protoRs.set_return_code(Proto::CreatePlayerReturnCode::CPR
    }
    ...
}
```

收到创建协议之后，从预处理字典中取出了已经准备好的DatabaseStmt。随后调用了ClearStmtParam、AddParamUint64和AddParamStr三个函数，分别是为了清理旧数据、压入一个uint64参数和压入一个字符串到预处理结构中。下面来分别看一看这几个参数的实现。

```
void MysqlConnector::ClearStmtParam(DatabaseStmt* stmt) {
    stmt->bind_index = 0;
    stmt->bind_buffer_index = 0;
}
```

函数ClearStmtParam用于清除上一次的数据，让缓存回到最初状态。

创建角色的SQL语句“insert into player (sn, account, name, savetime, createtime) value (?, ?, ?, now(), now())”有3个参数，分别是uint64和两个string。下面重点来看AddParamUint64和AddParamStr函数。

```
void MysqlConnector::AddParamUint64(DatabaseStmt* stmt, uint64 val)
{
    MYSQL_BIND* pBind = &stmt->bind[stmt->bind_index];
    pBind->buffer_type = MYSQL_TYPE_LONGLONG;
    pBind->buffer = &stmt->bind_buffer[stmt->bind_buffer_index];
    pBind->is_unsigned = true;
    *static_cast<uint64*>(pBind->buffer) = val;
    stmt->bind_index++;
    stmt->bind_buffer_index += sizeof(uint64);
}
```

函数AddParamUint64将一个uint64的值绑定到MYSQL_BIND结构中。首先需要取出对应这个uint64属性的MYSQL_BIND，即如下一句代码：

```
MYSQL_BIND *mbind = &stmt->bind[stmt->bind_index];
```

这里有一个潜规则，默认在设置参数时是按SQL语句的“？”顺序设置的，即(sn, account, name)，顺序为uint64、string、string，如果按string、string、uint64就会出错。

结构MYSQL_BIND非常复杂，简单列举一下需要用到的几个重要属性：

```
typedef struct st_mysql_bind {
    unsigned long *length;           /* 指向输出长度的指针 */
    void *buffer;                   /* 读写数据的缓冲区 */
    enum enum_field_types buffer_type; /* buffer type */
    my_bool is_unsigned;             /* 设置，如果整数类型是 */
    ...
} MYSQL_BIND;
```

下面就参数逐一说明。

(1) `buffer`为数据缓冲区。输入时，参数数据值绑定到缓冲区`buffer`中，供`mysql_stmt_execute`使用；输出时，将结果绑定到缓冲区`buffer`中，供`mysql_stmt_fetch`使用。对于日期和时间数据类型，缓冲区应指向`MYSQL_TIME`结构。

(2) `buffer_type`为数据类型，常用的有`MYSQL_TYPE_LONG`、`MYSQL_TYPE_BLOB`等。

(3) `is_unsigned`表示数据是否是无符号类型。

(4) `*length`属性指向一个`unsigned long*`无符号长整型指针。该属性只用在字符或二进制数据上。输入时，设置`*length`为`buffer`中的参数值的实际长度，供`mysql_stmt_execute`使用。数值类数据如`long`或`long long`，`*length`没有用，因为`buffer_type`值决定了数值的长度。

得到`MYSQL_BIND`指针之后，需要对其进行初始化。每一个`MYSQL_BIND`实例都需要外部提供一块内存给它存储数据。而我们创建的`DatabaseStmt`结构正好提供了一块内存用来存储这些数据。以下语句是基于`uint64`数据进行的一个赋值。

```
pBind->buffer_type = MYSQL_TYPE_LONGLONG;
pBind->buffer = &stmt->bind_buffer[stmt->bind_buffer_index];
pBind->is_unsigned = true;
*static_cast<uint64*>( pBind->buffer ) = val;
```

首先设置这个`MYSQL_BIND`的数据类型为`MYSQL_TYPE_LONGLONG`，再设置缓存区，自定义结构中的`buffer`提供了一大块连续的数据块，这里派上了用场，在第二行代码中，将数据地址赋值给缓存。是无符号类型，第三行代码属性`is_unsigned`设为`true`。最后的语句将缓存的可写位移进行偏移。

```
stmt->bind_index++;
stmt->bind_buffer_index += sizeof( uint64 );
```

偏移的值为写入数值的内存长度。这里的`uint64`即为`sizeof( uint64 )`。以上是给一个数值赋值，`AddParamStr`函数用于给

字符串赋值，在uint64的例子中，数值的长度是一定的，而字符串的长度却不一定，AddParamStr的实现如下：

```
void MysqlConnector::AddParamStr(DatabaseStmt* stmt, const char* v
    MYSQL_BIND* pBind = &stmt->bind[stmt->bind_index];
    int len = strlen(val);
    pBind->buffer_type = MYSQL_TYPE_STRING;
    pBind->buffer = &stmt->bind_buffer[stmt->bind_buffer_index];
    pBind->length = (unsigned long*)& stmt->bind_buffer[stmt->bind
engine_strncpy((char*)pBind->buffer, len + 1, val, len + 1);
    *(pBind->length) = len;
    pBind->buffer_length = len;
    stmt->bind_index++;
    stmt->bind_buffer_index += (len + 1 + sizeof(unsigned long*));
}
```

同样，先得到MYSQL_BIND指针，设置buffer_type为MYSQL_TYPE_STRING，参数是字符串时，MYSQL_BIND的buffer属性就有了意义，它会指向字符串的地址，这里将DatabaseStmt结构中buffer的当前指针分配给它，并调用strncpy函数将传入的串复制到这块内存块中，同时MYSQL_BIND的length需要指定一个unsigned long*指针，这个指针指向的地址赋值为输入字符串的长度。

值得注意的是，在MYSQL_BIND结构中，buffer和length都是指针，指针指向的是一个外部地址，这样的做法有利于精减内存，没必要在MySQL库内部再为这些参数开辟内存块来存储数据。注意最后一行代码：

```
stmt->bind_buffer_index += ( len + 1 + sizeof( unsigned long* ) );
```

DatabaseStmt中已经用了缓存，不能再使用，需要设置一个偏移值。

图7-2展示了DatabaseStmt的真实内存结构。从内存结构中可以看到所有参数的值都被保存在了缓存中，数值型类型因为长度是固定的，不需要开辟空间保存长度。而字符型参数除了要保存内容之外，还要有一个空间来保存它的长度。

`char*`，但在这个缓存中放置了`uint64`、`string`和`long`。C++的这种特性虽然好用，但使用时需要特别注意，以免产生内存越界。

查询数据时使用的函数说明

前面介绍了写入数据操作，本节介绍查询数据的方法。其步骤为：使用SQL语句进行查询时，不同的SQL语句获取的数据不同，但是都可以得到一个MYSQL_RES集合，遍历集合，最后将行数据转换成自定义的内存数据。其中涉及的API函数如下：

函数1：查询函数

```
int mysql_query(MYSQL *mysql, const char *stmt_str)
int mysql_real_query(MYSQL *mysql, const char *stmt_str, unsigned
```

库中提供了两个查询函数。函数mysql_query执行指定的SQL语句，参数stmt_str可以不带SQL语句的结束符“；”，但必须是有结束符的字符串，即最后以'\0'字符结尾。mysql_query函数使用的SQL语句不能带二进制数据，如果需要带二进制数据，就需要使用函数mysql_real_query。从函数定义上能看得出来，mysql_real_query函数执行SQL语句的时候，使用的参数是char*和它的长度。这个char*的字符串是允许存在'\0'这种结束符的。这就是这两个函数本质上的区别。

如果要查看执行该语句后的结果，那么可以在mysql_query调用之后调用函数mysql_field_count查看有多少列数据。如果执行的语句不是一个select，那么mysql_field_count调用的结果可能为0。另外，执行操作也可以由mysql_query来完成，即传入的SQL语句可以不是一个select语句。

函数2：读取结果

```
MYSQL_RES *mysql_store_result(MYSQL *mysql)
```

调用mysql_query函数之后可以用mysql_store_result得到结果，该函数将全部结果缓存到MYSQL_RES结构中并返回，MYSQL_RES用完之后需要使用mysql_free_result释放数据。

函数mysql_store_result返回为空时，不意味着失败。如果执行语句是insert语句，mysql_store_result就会返回空，因为insert语句并没有集合可以返回。

函数3：获取结果中有多少列

```
unsigned int mysql_num_fields(MYSQL_RES *result)
```

调用函数mysql_store_result的结果不为空时，可以调用mysql_num_fields来判断有多少列。

函数4：读取字段

```
MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result)
```

该函数的使用相当于一个迭代器，对MYSQL_RES的列数据进行一个迭代，当返回值为空时表示没有更多的列了。

函数5：获取行

```
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)
```

同理，函数mysql_fetch_row也是一个迭代器，迭代的是MYSQL_RES集合，也就是mysql_store_result得到的集合。

查询数据示例

一个完整的查询数据步骤为：首先组织一条SQL语句，调用 `mysql_query` 得到一个结果集，再通过调用 `mysql_fetch_row` 函数得到 `MYSQL_ROW` 行数据。而每一行的具体数据则是由 `MYSQL_ROW` 类的操作函数读取数据的。下面我们用一个例子加以说明，本节的源代码位于 `07_01_db` 目录中。

7.7.1 Query查询函数

在MysqlBase类中，函数Query封装了mysql_query函数。其实现如下：

```
bool MysqlBase::Query(const char* sql, my_ulonglong& affected_rows)
{
    if (nullptr != _pMysqlRes) {
        mysql_free_result(_pMysqlRes);
        _pMysqlRes = nullptr;
    }
    if (mysql_query(_pMysql, sql) != 0) {
        LOG_ERROR("Query error:" << mysql_error(_pMysql) << " sql:");
        return false;
    }
    // maybe query is not a select
    _pMysqlRes = mysql_store_result(_pMysql);
    if (_pMysqlRes != nullptr) {
        _numFields = mysql_num_fields(_pMysqlRes);
        _pMysqlFields = mysql_fetch_fields(_pMysqlRes);
    }
    affected_rows = mysql_affected_rows(_pMysql);
    return true;
}
```

上面的代码使用了MySQL库提供的函数mysql_query，得到了一个MYSQL_RES集合，将这个集合的指针保存在类变量pMysqlRes中，该属性是结构MYSQL_RES的指针。同时，将列数据保存在变量pMysqlFields中，该属性是结构MYSQL_FIELD的指针，可以称它为字段数据。

这里一下解释为什么会有字段数据，以及它的作用是什么。

使用select语句时，例如“select sn, name from player”和“select name, sn from player”的字段数据是不一样的。调用函数mysql_fetch_fields得到返回集合的字段信息，对于第一条语句来说，sn在前，name在后。第二条语句正好相反。MYSQL_FIELD主要是对字段描述的一个结构，对于sn来说，描述它的结构为MYSQL_FIELD，name也会生成一个MYSQL_FIELD。结构MYSQL_FIELD其中的一个字段enum_field_types type表示当前这个字段的真实类型。那么，对于上面两条select语句，第一条语句的第一个字段类型和第二条

语句的第一个字段类型显然是不相同的。

7.7.2 查询玩家数据

以查询玩家数据这一流程来具体说明查询函数的使用方法。首先，当收到查询玩家协议L2DB_QueryPlayerList时，MysqlConnector类的处理函数HandleQueryPlayerList的实现如下：

```
void MysqlConnector::HandleQueryPlayerList(Packet* pPacket) {
    auto protoQuery = pPacket->ParseToProto<Proto::QueryPlayerList>
    QueryPlayerList(protoQuery.account(), pPacket->GetSocket());
}

void MysqlConnector::QueryPlayerList(std::string account, SOCKET s
my_ulonglong affected_rows;
std::string sql = strutil::format("select sn, name, base, item
if (!Query(sql.c_str(), affected_rows)) {
    LOG_ERROR("!!! Failed. MysqlConnector::HandleQueryPlayerLi
    return;
}
Proto::PlayerList protoRs;
protoRs.set_account(account.c_str());
Proto::PlayerBase protoBase;
if (affected_rows > 0) {
    std::string tempStr;
    MYSQL_ROW row;
    while ((row = Fetch())) {
        auto pProtoPlayer = protoRs.add_player();
        pProtoPlayer->set_sn(GetUInt64(row, 0));
        pProtoPlayer->set_name(GetString(row, 1));

        GetBlob(row, 2, tempStr);
        protoBase.ParseFromString(tempStr);
        pProtoPlayer->set_level(protoBase.level());
        pProtoPlayer->set_gender(protoBase.gender());
    }
}
// 没有找到也需要返回pResultPacket
SendPacket(Proto::MsgId::L2DB_QueryPlayerListRs, socket, proto
}
```

在查询玩家数据时，直接使用了SQL语句“select sn, name, base, item, misc from player where account='%s'”，接着调用了之前封装好的函数Query，当返回数据时，调用了Fetch函数，其定义如

下:

```
MYSQL_ROW MysqlBase::Fetch() const {
    if (_pMysqlRes == nullptr)
        return nullptr;
    return mysql_fetch_row(_pMysqlRes);
}
```

从上面的代码可以看出，调用mysql_fetch_row函数从集合中取出了当前行的数据。while（（row=Fetch（）））不断取出当前行的数据，直到这个集合被遍历完为止。对于行数据，首先调用GetUint64函数取出当前行中的sn数据，调用GetString函数取出当前行中的name数据，后面以此类推。函数GetUint64的实现方法如下：

```
uint64 MysqlBase::GetUint64(MYSQL_ROW row, int index) {
    if (row[index] == nullptr) {
        LOG_ERROR("!!! Failed. MysqlConnector::GetUint64");
        return 0;
    }
    return atoll(row[index]);
}
```

看看MYSQL_ROW的定义：

```
typedef char **MYSQL_ROW; /* return data as array of strings */
```

它指向了一个数组地址，可以根据不同的类型直接转换成不同的数据。在这些类型中，有一个特别需要注意的是二进制Blob数据，取得Blob的函数实现如下：

```
void MysqlBase::GetBlob(MYSQL_ROW row, int index, std::string& protoStr,
    unsigned long* pLengths = mysql_fetch_lengths(_pMysqlRes);
    long blobLength = pLengths[index];
    if (blobLength <= 0) {
        protoStr = "";
        return;
    }
```

```

        char* blobByte = new char[blobLength + 1];
        GetBlob(row, index, blobByte, blobLength);
        blobByte[blobLength] = '\0';
        protoStr = blobByte;
        delete[] blobByte;
    }
int MysqlBase::GetBlob(MYSQL_ROW row, int index, char* buf, unsigned
    int l = size > 0 ? size : 0;
    if (row[index] == nullptr) {
        LOG_ERROR("!!! Failed. MysqlConnector::GetBlob");
        return 0;
    }
    unsigned long* lengths = mysql_fetch_lengths(_pMysqlRes);
    if (lengths[index] < l)
        l = lengths[index];
    memcpy(buf, row[index], l);
    return l;
}

```

在获取Blob数据时，首先需要知道Blob的长度，调用mysql_fetch_lengths函数从数据集合中得到一个长度数组。根据当前字段所在的index下标，在长度数组中找到这个字段的长度。为了读取二进制数据，需要创建一块内存，调用memcpy函数完全复制数据。取得数据之后，再传递到上层调用中。

数据表的创建与更新

前面几节详细地介绍了MySQL的一些API的使用，本节讨论一下数据库的创建与更新问题。一般来说，服务端每一个版本的代码都对应一个相应的SQL文件。导入SQL文件到数据库之后，服务器才可以正常地使用数据库。但这项功能要求在编译源代码的同时维护一系列SQL文件。长期来讲，这是一件非常令人恼火的事情，代码版本与SQL版本不一致时就会出错。

鉴于此，我们将数据库的结构写入C++的源代码中，并随着版本的推进进行自动更新。这样就可以把目光专注在源代码上，不需要维护SQL文件了。图7-3展示了创建与更新数据库的流程。

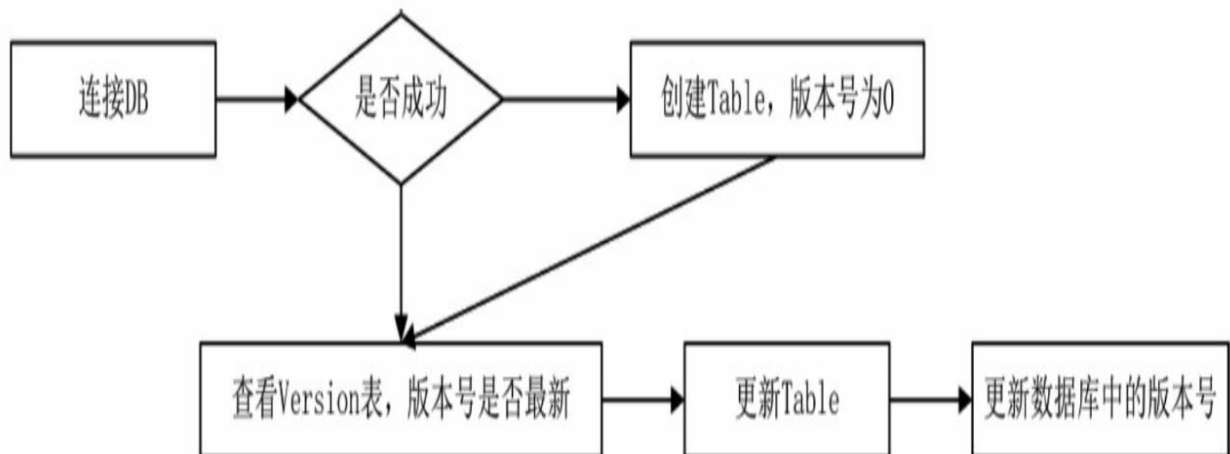


图7-3 创建与更新数据库

概述一下这个流程，首先连接数据库，查看数据库是否连接成功，如果失败，就查询出错的编号；如果编号为1049，就表示数据库不存

在。官方对于1049的解释如下：

```
Error: 1049 SQLSTATE: 42000 (ER_BAD_DB_ERROR)
Message: Unknown database '%s'
```

如果数据库不存在，此时就需要创建一个数据库，并进行数据库的所有表格的创建，同时创建一个名为Version的表，用来记录当前版本号，初始的版本号为0，根据当前代码的版本号进行升级。

如果数据库连接成功，就查询数据库表version中的版本号，以对应当前代码的版本号进行升级。更新完成之后有一个重要的操作，即将数据库中version表中的版本号定为当前代码的版本号。

下面是更新数据库的类MysqlTableUpdate，在07_01_db目录的dgmgr工程的mysql_table_update.cpp文件中。其类定义如下：

```
class MysqlTableUpdate :public MysqlBase, public Singleton<MysqlTa
public:
    void Check();
private:
    bool CreateDatabaseIfNotExist();
    bool UpdateToVersion();           // 检查DB数据，更新到最新版本
    bool Update00();                 // 00版本的数据修改
private:
    typedef std::function<bool(void)> OnUpdate;
    std::vector<OnUpdate> _update_func;
    int const _version = 0;
};
```

在MysqlTableUpdate类中存储了一个当前数据库代码的版本号，即version，每一次变更数据库之后，这个变量就要加1。现在它是第0个版本号。

MysqlTableUpdate类是在main.cpp中被创建的。也就是说在dbmgr启动之初就对数据库进行了更新检查。

```
int main(int argc, char* argv[]) {
```

```

mysql_library_init(0, nullptr, nullptr);
...
auto pTableUpdateObj = MysqlTableUpdate::Instance();
pTableUpdateObj->Check();
pTableUpdateObj->DestroyInstance();
...
return 0;
}

```

在启动线程之前，调用MysqlTableUpdate::Check检查当前数据库版本是否为最新版本。检查数据库函数代码如下：

```

void MysqlTableUpdate::Check() {
    if (!ConnectInit())
        return;
    auto pYaml = Yaml::GetInstance();
    auto pDbMgrCfig = dynamic_cast<DBMgrConfig*>(pYaml->GetConfig
        _pDbCfig = pDbMgrCfig->GetDBConfig(DBMgrConfig::DBTypeMysql);
    if (_pDbCfig == nullptr) {
        LOG_ERROR("Init failed. get mysql config is failed.");
        return;
    }
    if (mysql_real_connect(_pMysql, _pDbCfig->Ip.c_str(), _pDbCfig->
        mysql_select_db(_pMysql, _pDbCfig->DatabaseName.c_str());
    }
    int mysqlerrno = CheckMysqlError();
    if (mysqlerrno == ER_BAD_DB_ERROR) {
        // 1049: Unknown database 没有找到数据库，就新建一个
        if (!CreateDatabaseIfNotExist()) {
            Disconnect();
            return;
        }
        // 创建成功之后，选择到当前数据库
        mysql_select_db(_pMysql, _pDbCfig->DatabaseName.c_str());
        mysqlerrno = CheckMysqlError();
    }
    if (mysqlerrno > 0) {
        Disconnect();
        return;
    }
    // 检查版本，自动更新
    if (!UpdateToVersion()) {
        LOG_ERROR("!!!Failed. Mysql update. UpdateToVersion");
        return;
    }
}

```

```
    ...  
}
```

下面我们详细分析这一段代码。

7.8.1 创建表

在检查数据库时，首先创建一个MySQL连接，此时并没有选择数据库，函数mysql_real_connect中关于数据库名的参数值为nullptr。连接成功之后，调用mysql_select_db函数试探数据库是否存在，如果返回错误码为ER_BAD_DB_ERROR（1049），就认为可以创建一个数据库。创建数据库函数代码如下：

```
bool MysqlTableUpdate::CreateDatabaseIfNotExist() {
    // 是否存在数据库，如果不存在就创建
    std::string querycmd = strutil::format("CREATE DATABASE IF NOT
my_ulonglong affected_rows;
    if (!Query(querycmd.c_str(), affected_rows)) {
        LOG_ERROR("!!! Failed. MysqlConnector::CreateDatabaseIfNot
        return false;
    }
    // 连接DB之后，选择指定的数据库
    if (mysql_select_db(_pMysql, _pDbCfg->DatabaseName.c_str()) !=
        LOG_ERROR("!!! Failed. MysqlConnector::CreateDatabaseIfNot
        return false;
    }
    // 设置数据库的字符集，从YAML配置中读取需要的字符集
    if (mysql_set_character_set(_pMysql, _pDbCfg->CharacterSet.c_s
        LOG_ERROR("!!! Failed. MysqlConnector::CreateDatabaseIfNot
        return false;
    }
    // 设置数据库区分字母大小写，配置文件中为utf8_general_ci
    querycmd = strutil::format("ALTER DATABASE CHARACTER SET %s CO
    if (!Query(querycmd.c_str(), affected_rows)) {
        LOG_ERROR("!!! Failed. MysqlConnector::CreateDatabaseIfNot
        return false;
    }
    // 创建一个version表，使用了InnoDB方式
    std::string create_version =
        "CREATE TABLE IF NOT EXISTS `version` (" \
        "`version` int(11) NOT NULL," \
        "PRIMARY KEY (`version`)" \
        ") ENGINE=%s DEFAULT CHARSET=%s;";
    std::string cmd = strutil::format(create_version.c_str(), "Inn
    if (!Query(cmd.c_str(), affected_rows)) {
        LOG_ERROR("!!! Failed. MysqlConnector::CreateTable. " << L
        return false;
    }
}
```

```

// 创建一个player表
std::string create_player =
    "CREATE TABLE IF NOT EXISTS `player` (" \
    "`sn` bigint(20) NOT NULL," \
    "`name` char(32) NOT NULL," \
    "`account` char(64) NOT NULL," \
    "`base` blob," \
    "`item` blob," \
    "`misc` blob," \
    "`savetime` datetime default NULL," \
    "`createtime` datetime default NULL," \
    "PRIMARY KEY (`sn`)," \
    "UNIQUE KEY `NAME` (`name`)," \
    "KEY `ACCOUNT` (`account`)" \
    ") ENGINE=%s DEFAULT CHARSET=%s;";
cmd = strutil::format(create_player.c_str(), "InnoDB", _pDbCfg
if (!Query(cmd.c_str(), affected_rows)) {
    LOG_ERROR("!!! Failed. MysqlConnector::CreateTable" << LOG
    return false;
}
// 创建完成后, 修改version表中的version字段, 设为初始的0号版本
cmd = "insert into `version` VALUES ('0')";
if (!Query(cmd.c_str(), affected_rows)) {
    LOG_ERROR("!!! Failed. MysqlConnector::CreateTable." << LO
    return false;
}
return true;
}

```

顾名思义, CreateDatabaseIfNotExist函数是在数据库不存在时创建一个全新的供游戏使用的数据库。在上面的代码中创建了两张表, 两条SQL语句分别创建了version和player表。创建player表时, 同时创建了几个索引, 用于快速查询。数据表创建完成之后, 执行了一条SQL语句: "insert into 'version' VALUES ('0')", 即当前版本为0号的初始版本。

在创建player表时使用了一个关键字InnoDB。InnoDB和MyISAM是使用MySQL时常用的两个表类型。如果需要, 那么这里也可以替换成MyISAM。

7.8.2 更新表

在整个开发或上线的过程中，数据表不可能是一成不变的。有时，需要对表结构进行更新。创建表完成之后调用了UpdateToVersionDB进行升级，升级代码如下：

```
bool MysqlTableUpdate::UpdateToVersion() {
    my_ulonglong affected_rows;
    std::string sql = "select version from `version`";
    if (!Query(sql.c_str(), affected_rows))
        return false;
    MYSQL_ROW row = Fetch();
    if (row == nullptr)
        return false;
    int version = GetInt(row, 0);
    if (version == _version)
        return true;
    // 如果DB版本不匹配，就升级DB
    for (int i = version + 1; i <= _version; i++) {
        if (_update_func[i] == nullptr)
            continue;
        if (!_update_func[i]()) {
            LOG_ERROR("UpdateToVersion failed!!!!, version=" << i);
            return false;
        }
        LOG_INFO("update db to version:" << i);
        // 成功之后，更改DB的版本
        std::string cmd = strutil::format("update `version` set ve
        if (!Query(cmd.c_str(), affected_rows)) {
            LOG_ERROR("UpdateToVersion failed!!!!, change version
            return false;
        }
    }
    return true;
}
```

更新时首先查询当前数据库的版本号，再从注册的更新列表update_func中取得更新函数来更新每一个版本。如同在CreateDatabaseIfNotExist中做的一样，最后一步将version表中的version字段修改为当前的DB版本号。列表update_func的下标从0开始。在初始化时绑定好了version为0的函数为Update00。

```
MysqlTableUpdate::MysqlTableUpdate() {  
    // 注册更新函数，按下标执行，注意顺序  
    _update_func.push_back(BindFunP0(this, &MysqlTableUpdate::Upda  
}  
bool MysqlConnector::Update00( ) {  
    return true;  
}
```

Update00实现为空，我们没有进行任何操作。现在执行dbmgr工程来测试创建数据库的流程。执行之前需要配置dbmgr工程数据，在engine.yaml文件中为dbmgr连接的MySQL配置IP和端口，如果你的IP、端口与库中的配置不同，就需要修改。

7.8.3 测试更新与创建组件

第一次运行dbmgr工程，程序输出信息如下：

```
[DEBUG] - Mysql update connect. 192.168.0.120:3306 starting...
[ERROR] - [CheckMysqlError]MysqlConnector::CheckError. mysql_errno
[DEBUG] - Mysql. try create database:e_gamedata
[DEBUG] - Mysql Update successfully! addr:192.168.0.120:3306
```

首先连接数据库，反馈回1049编号的错误，这时试着创建一个e_gamedata数据库。用工具打开数据库，会发现已经创建了两个表，其内部结构与我们期望的一致。

```
MariaDB [e_gamedata]> show tables;
+-----+
| Tables_in_e_gamedata |
+-----+
| player                |
| version               |
+-----+
2 rows in set (0.00 sec)
MariaDB [e_gamedata]> show columns from player;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| sn         | bigint(20)    | NO   | PRI | NULL    |       |
| name       | char(32)      | NO   | UNI | NULL    |       |
| account    | char(64)      | NO   | MUL | NULL    |       |
| base       | blob          | YES  |     | NULL    |       |
| item       | blob          | YES  |     | NULL    |       |
| misc       | blob          | YES  |     | NULL    |       |
| savetime   | datetime      | YES  |     | NULL    |       |
| createtime | datetime      | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.01 sec)
```

再来测试一下更新效果，使用以下步骤：

(1) 在mysql_table_update.h文件中找到变量version，在旧数据

的基础上加1，即1。

(2) 增加版本1的更新函数Update01。

(3) 修改MysqlTableUpdate类的构造函数，增加更新函数注册：

```
MysqlTableUpdate::MysqlTableUpdate() {  
    // 注册更新函数，按下标执行，注意顺序  
    _update_func.push_back(BindFunP0(this, &MysqlTableUpdate::  
    _update_func.push_back(BindFunP0(this, &MysqlTableUpdate::  
}
```

(4) 实现Update01函数，代码如下：

```
bool MysqlTableUpdate::Update01() {  
    std::string sql = "ALTER TABLE `player` ADD COLUMN `testa`  
    my_ulonglong affected_rows;  
    if (!Query(sql.c_str(), affected_rows)) {  
        LOG_ERROR("!!! Failed. MysqlTableUpdate::Update01. ");  
        return false;  
    }  
    return true;  
}
```

在上面的代码中，在player表的列misc之后添加一个二进制、名为testa的列。更新后执行，其打印数据如下：

```
[DEBUG] - Mysql update connect. 192.168.0.120:3306 starting...  
[INFO] - update db to version:1  
[DEBUG] - Mysql Update successfully! addr:192.168.0.120:3306
```

因为已经有了数据库，没有出现1049编号的错误，所以不会走创建数据库的流程，而是直接连接到数据库上，并进行对应版本号1的更新。打开数据库会发现有两个改变：第一个改变是player表中多了testa列；另一个改变是version表中的version列被更改为1。

```

MariaDB [e_gamedata]> show columns from player;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| sn         | bigint(20)    | NO   | PRI | NULL    |       |
| name       | char(32)      | NO   | UNI | NULL    |       |
| account    | char(64)      | NO   | MUL | NULL    |       |
| base       | blob          | YES  |     | NULL    |       |
| item       | blob          | YES  |     | NULL    |       |
| misc       | blob          | YES  |     | NULL    |       |
| testa     | blob          | YES  |     | NULL    |       |
| savetime   | datetime      | YES  |     | NULL    |       |
| createtime | datetime      | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
9 rows in set (0.00 sec)
MariaDB [e_gamedata]> select * from version;
+-----+
| version |
+-----+
| 1       |
+-----+
1 row in set (0.00 sec)

```

当然，在真实的开发过程中，从一个版本到另一个版本，数据库的修改可能非常复杂，修改的可能不止一张表。

数据表中的数据结构与protobuf结构

现在，框架已经在数据库中建立了player表，本节要讨论的是player表与protobuf的关系。在player表中的字段base使用了二进制数据，在本书的案例中，这些二进制数据都是protobuf结构的。

看到这里可能有些不习惯，大部分人都比较倾向于在数据库的表中创建给定的列。例如，player表中有等级Level列、经验Exp列，打开表格时，就可以直观地看到这些数据，现在将这些数据放到一个二进制的结构中，看上去就不直观了。

那么，为什么还要将这些数据变成不那么直观的二进制数据呢？

如果读者曾经有过做游戏的经验，那么一定有数据库升级的经历。游戏运营之后，基于策划的要求新增了某些功能，需要将某些数据存储到数据库中，于是创建了一个新的数据表，其中可能有col1、col2、...、col5等好几个存储列。运营了一个月之后，新的要求来了，需要对这个功能进行修改，需要删除col2、col3，增加col6、col7。

7.8节讲的更新数据库结构的操作可以帮上忙，编写一个新的更新函数。

但现在有一个更好的方案，不需要更改列就可以实现数据的删除与新增。更新数据时不需要再编写更新函数，也就是使用protobuf定义的结构作为存储数据结构。作为协议使用时，protobuf方便的序列化特性被广泛使用。除此之外，它还有一个非常给力的特性就是兼容性，这个特性用于存储时也非常给力。

从整体效率上来看，使用二进制存取的方式要高于多列存取的方式。

式。这是由于二进制压缩了表的列数，只需要几个列就可以表示大量的数据。列的多少在MySQL的存取上也会影响效率。下面通过例子来看如何将protobuf存到MySQL中。本节的源代码依然存放在07_01_db目录中。工程libserver中新建了一个db.proto文件，该文件中定义的结构是存储在数据库中的结构，定义如下：

```
syntax = "proto3";
package Proto;
...
enum Gender {
    none = 0;
    male = 1;
    female = 2;
}
message PlayerBase {
    Gender gender = 1;
    int32 level = 2;
}
message LastWorld {
    int32 world_id = 1;
    int64 world_sn = 2;
    Vector3 position = 3;
}
message PlayerMisc {
    LastWorld last_world = 1;    // 公共地图
    LastWorld last_dungeon = 2; // 副本地图
    ...
}
message Player {
    uint64 sn = 1;
    string name = 2;
    PlayerBase base = 3;
    PlayerMisc misc = 4;
}
```

在上面的结构中，Proto::Player就是最终的Player表。目前只用到了sn、name、base三个列，其他的列暂时无用。PlayerBase结构就是base列的数据结构，这个结构中有一个int32类型的level数据，还有一个枚举类型来表示性别，以便区分显示模型。PlayerMisc是misc列的数据结构，保存了位置数据，last_world表示公共地图的位置，last_dungeon表示副本地图的位置。

图7-4从整体上展示了数据库的结构。protobuf中的数据是一个可以嵌套的结构，也可以随时随意地增加与删除属性。在编码时，只有一个规则需要注意，就是属性编号不能重用。假如之前的4号属性我们不再使用，可以将这个4号定义直接删除，这大大简化了我们的工作，不再需要编写数据库更新函数了。

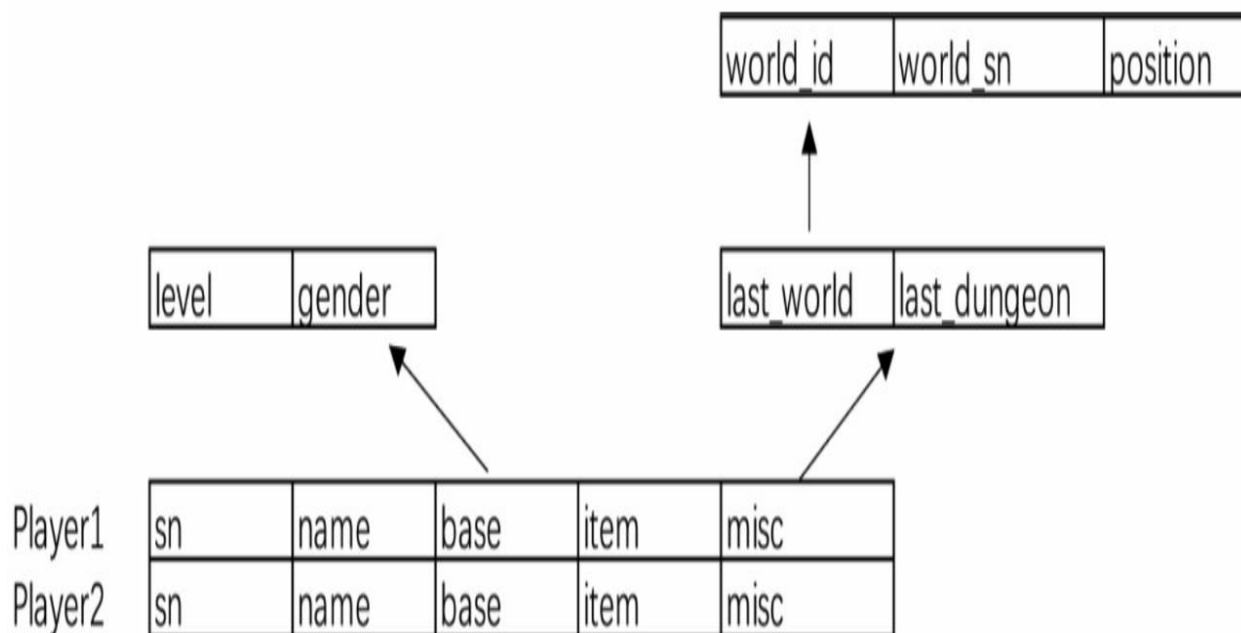


图7-4 玩家数据结构

角色查询与创建流程

数据库的操作是整个框架的基础，下面将之前的功能整合起来。在第6章的例子中，机器人已经登录到了login进程中，梳理一下登录之后的流程：在进程login中验证完成之后，需要向dgmgr进程发起查询角色的协议，如果返回数据没有角色，就由客户端发起创建角色的协议。整个流程如图7-5所示。

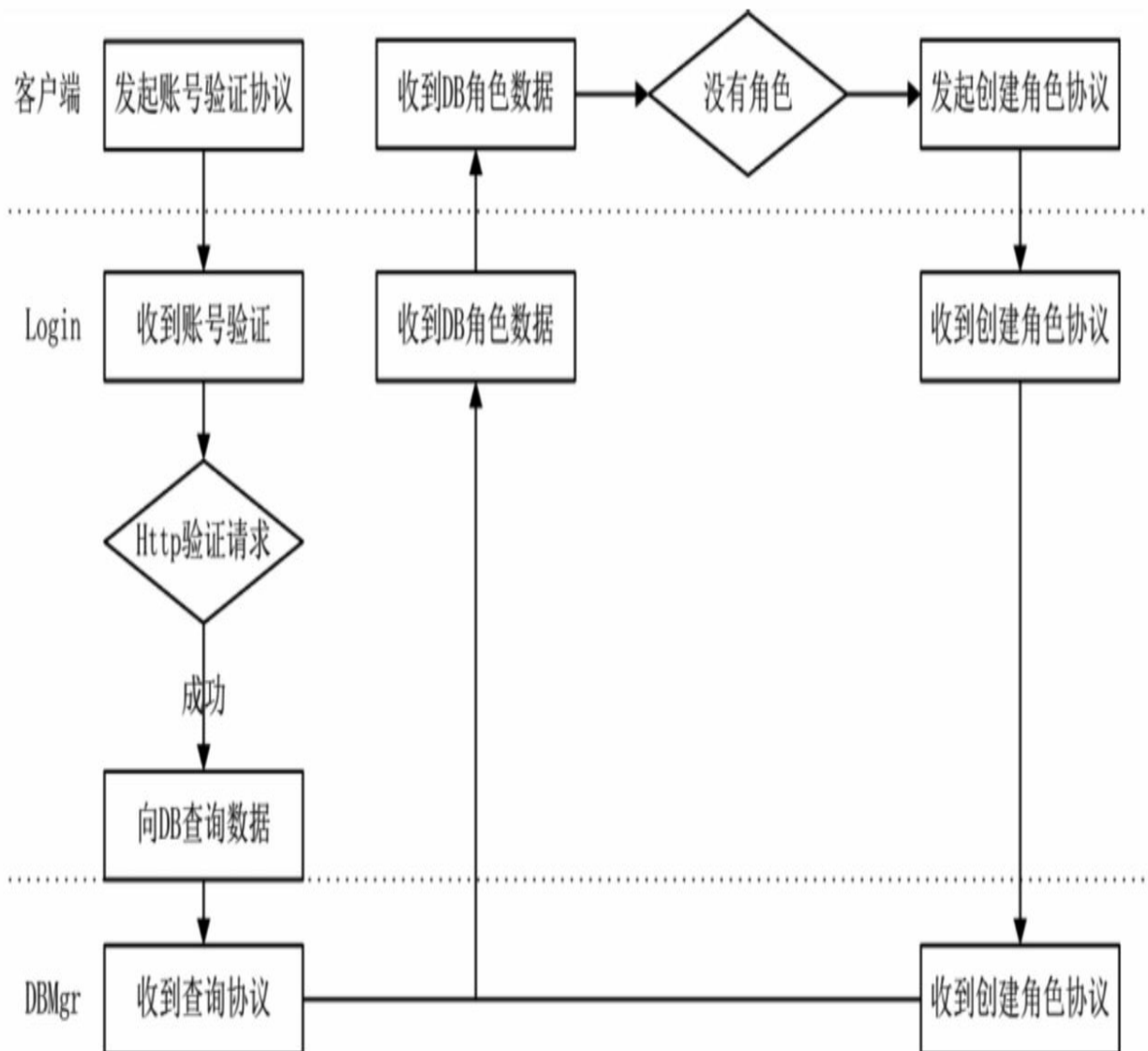


图7-5 创建、读取角色的流程

对于客户端来说，需要发起的协议有两个：一个是账号验证协议；另一个是创建角色协议。需要处理的协议是收到角色列表协议。

对于login工程来说，需要处理3个协议：验证账号、收到角色列表返回数据以及收到创建角色协议的处理。

对于dbmgr工程来说，需要处理两个协议：一个是查询角色协议；另一个是创建角色协议。

7.10.1 为login进程与dbmgr进程创建连接类

在前面的案例中，协议总是从客户端（robots进程）传来的，现在服务端是多进程了，验证账号在login进程中，而创建角色在dbmgr进程中，这两个进程是如何传递消息的呢？

首先，login和dbmgr都打开了监听端口，但login的监听端口是为了玩家连接，而dbmgr打开监听端口则是为了其他进程方便连接到它。所以，在login工程的main.cpp文件中加入了一个新的NetworkConnector组件。

```
int main(int argc, char* argv[]) {
    const APP_TYPE curAppType = APP_TYPE::APP_LOGIN;
    ServerApp app(curAppType, argc, argv);
    app.Initialize();
    ...
    auto pYaml = Yaml::GetInstance();
    auto pCommonConfig = pYaml->GetIPEndPoint(curAppType);
    pThreadMgr->CreateComponent<NetworkListen>(pCommonConfig->Ip,
    pThreadMgr->CreateComponent<NetworkConnector>((int)APP_TYPE::A
    ...
}
```

监听组件NetworkListen传入了参数监听的IP与端口，而连接组件NetworkConnector传入的参数为AppType。在NetworkConnector初始化时，我们通过AppType从YAML配置文件中读出了需要连接的AppType的IP与端口。

在上面的代码中，希望可以连接到dbmgr的端口。当NetworkConnector类从对象池中被唤醒之后，它做了如下操作：

```
void NetworkConnector::AwakeFromPool(int appType, int appId) {
    auto pYaml = Yaml::GetInstance();
    auto pComponent = pYaml->GetIPEndPoint((APP_TYPE)appType, appId);
    if (pComponent == nullptr) {
        LOG_ERROR("can't find yaml config.");
        return;
    }
}
```

```
Connect(pComponent->Ip, pComponent->Port);  
auto pNetworkLocator = ThreadMgr::GetInstance()->GetComponent  
pNetworkLocator->AddConnectorLocator(this, (APP_TYPE)appType,  
}
```

在初始化的同时，将这个网络类放到了NetworkLocator组件中，方便发送数据时的定位。这样，login进程就与dbmgr进程有了一个网络连接。

7.10.2 找到指定的NetworkConnector实例

新流程需要向dbmgr发起一个查询请求，那么login是如何找到dbmgr的连接实例的呢？继续向下走这个流程，验证账号请求结束之后，在其处理函数中有如下实现：

```
void Account::HandleAccountCheckToHttpRs(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::AccountCheckToHttpRs>();
    auto pPlayer = _playerMgr.GetPlayer(proto.account());
    if (pPlayer == nullptr) {
        std::cout << "can't find player. account:" << proto.account();
        return;
    }
    Proto::AccountCheckRs protoResult;
    protoResult.set_return_code(proto.return_code());
    SendPacket(Proto::MsgId::C2L_AccountCheckRs, pPlayer->GetSocket());
    // 验证成功，向DB发起查询
    if (proto.return_code() == Proto::AccountCheckReturnCode::ARC_
        Proto::QueryPlayerList protoQuery;
        protoQuery.set_account(pPlayer->GetAccount().c_str());
        SendPacket(Proto::MsgId::L2DB_QueryPlayerList, protoQuery,
    }
}
```

函数HandleAccountCheckToHttpRs是HTTP请求的返回处理函数。它首先从返回协议中取出账号，以验证登录玩家对象是否还存在，因为每一步都是异步操作，所以每一次消息到来时，玩家都有可能已经下线了。如果验证成功，就向dbmgr发送L2DB_QueryPlayerList协议。组织协议看上去并不复杂，只传入了账号数据，最后调用了SendPacket函数发送数据。SendPacket函数的定义在message_system.h文件中，有两个定义，代码如下：

```
void SendPacket(const Proto::MsgId msgId, const SOCKET socket, goo
void SendPacket(const Proto::MsgId msgId, google::protobuf::Messag
```

当需要向网络层发送消息时，现有两种发送方式，一种类似Robot类，创建一个Socket，主动连接到服务端，这类连接可以从Player类上

拿到Socket值；另一种是服务端进程之间的数据发送，这时我们只需要向一个指定的APP_TYPE上发送数据即可。其实现如下：

```
void IMessageSystem::SendPacket(const Proto::MsgId msgId, google::
    auto packet = CreatePacket(msgId, 0);
    packet->SerializeToBuffer(proto);
    SendPacket(packet, appType, appId);
}
void IMessageSystem::SendPacket(Packet* packet, APP_TYPE appType,
    auto pNetworkLocator = ThreadMgr::GetInstance()->GetComponent
    auto pNetwork = pNetworkLocator->GetNetworkConnector(appType,
    if (pNetwork != nullptr) {
        packet->SetSocket(pNetwork->GetSocket());
        pNetwork->SendPacket(packet);
        return;
    }
    ...
}
```

在上面的代码中，从主线程中取出NetworkLocator组件，从中取得了dbmgr的INetwork实例发送数据。初始化NetworkConnector时，已经向NetworkLocator组件注册过数据了，所以NetworkLocator可以找到连接到dbmgr的NetworkConnector实例，直接发送数据。

7.10.3 创建角色

对于数据库查询角色的功能，我们在阐述MySQL函数的时候已经讲解过了，这里不再多说。玩家数据最终会传递到客户端，也就是我们的测试robots进程中，Robot类收到一个空的玩家数据，这时它会发起一个创建角色的协议，协议编号为C2L_CreatePlayer。在这个创建角色协议里，客户端一般会带上自己设定的参数，例如玩家名、玩家性别等。login进程收到这个协议的处理函数为Account::HandleCreatePlayer，这个函数最终将数据传递到了dbmgr进程，使用的协议编号为L2DB_CreatePlayer。这里我们不再详细讲述。

7.10.4 机器人登录创建角色测试

在07_01_db目录的工程中同时也准备了robots测试工程。现在需要启动两个服务端进程login和dbmgr。在启动之前确保engine.yaml文件中的数据库地址正确，不需要手动操作数据库，程序会自动创建数据库，并更新数据表。如果先启动的是login，就会看到它一直不断地出现re connect（重新连接）打印数据。因为它一直在试图连接到dbmgr上，所以直到启动好dbmgr进程之后，login的re connect行为才会停止。

启动robots之后，输入“login-a test”可以进行test登录，在Robot类中，我们完成了自动登录和自动创建角色的功能。第一次登录，dbmgr进程上会有以下打印消息：

```
[INFO] - HandlePlayerCreate sn:55650694413156386 account:test name
[DEBUG] - player list. account:test player list size:1 socket:700
```

为test账号创建了一个名为test-2086的角色，这个名字是随机取的。登录MySQL数据库，看看我们刚才新建的玩家数据：

```
MariaDB [e_gamedata]> select sn, name, account, base from player;
+-----+-----+-----+-----+
| sn          | name      | account | base |
+-----+-----+-----+-----+
| 55650694413156386 | test-2086 | test    | NULL |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

当然，如果登录已经创建过角色的账号，那么将不会再创建角色。关于Robot类的代码实现细节可参考源代码，由于篇幅有限，这里就不再赘述。

7.11

总结

本章介绍了对数据库的使用，数据库是游戏存储的基本组件。第8章将进一步优化这个ECS框架。但不需要担心，在这些改造中，我们几乎不用变更这些基于Actor原则的类，因为它们在某种程度上来说真的是与世隔绝，与外界没有任何耦合。

第8章

深入学习组件式编程

本章将继续深入学习组件式编程，同时进一步深入学习ECS体系中的System部分。在本章中将有一个全新的EntitySystem系统以及全新的UpdateSystem、MessageSystem系统。为了方便调试新建了allinone工程，将所有进程的所有功能都放在一起。除此之外，进一步优化对象池，让它以线程为单位，去掉对象池的锁。在深入应用之后，对于定时器函数的需求越来越多，所以在本章中加入了一个时间堆的组件。本章包括以下内容：

- 全新的System系统。
- 引入allinone工程。
- IAwakeSystem接口与对象池。
- 时间堆。

新的系统管理类SystemManager

在前面的代码中编写了一个EntitySystem来管理组件和实体，除此之外，还没有深入谈到System部分的内容。System可以理解为游戏逻辑的动作部分。任何一个框架都有动作部分，我们熟悉的Update更新操作就是一个动作。在前面的框架中，驱动Update的方式是给EntitySystem增加std::list<IUpdateSystem*>队列，这个队列中的每一个组件继承IUpdateSystem接口，并实现Update函数每帧的调用。伪代码如下：

```
void EntitySystem::Update() {
    for(auto one : updataObjs){
        one->Update();
    }
}
```

下面分析这种流程带来的问题。假设除了IUpdateSystem外，还有一个IStartSystem用于组件的首次调用，那么按照之前的逻辑，就需要在EntitySystem类中再增加一个std::list<IStartSystem*>队列。

随着游戏功能的增加，要做的动作会不断增加，如果有10个System，就要建10个相关集合，为每个集合编写调用函数，这使得EntitySystem变得越来越不可维护，这不符合之前一直强调的解耦思维。

一旦System多起来，按目前这个流程处理起来就会越来越麻烦。如果我们新加了一个MoveSystem用来处理移动相关的操作，就需要建立一个IMoveSystem接口，在这个接口定义一个虚函数UpdateMove。在EntitySystem中维护IMoveSystem的列表去调用UpdateMove，所做的工作依然不少。

一个组件想要处理的功能越多，需要继承的接口就越多。它可能要继承IUpdateSystem，还要继承IMoveSystem，也许还有其他更多的接口。以此类推，显然为每个System独立维护一个列表有数据冗余。

现在，我们需要颠覆这个逻辑，丢掉面向对象的接口，完全以组件的方式来实现系统功能。简单来说，不再使用IUpdateSystem这类的接口，而是改用UpdateComponent组件来替代它，拥有这个组件就拥有这个功能，而每一个类都可以为自己创建一个UpdateComponent组件，不再需要继承特定的基类。在一些特殊的情况下，甚至可以在某些时间给某些Entity加上UpdateComponent组件。下面来看新的系统是如何设计的。

本节的源代码位于目录08_01_system中，从旧框架到新框架，需要完成如下几步：

(1) 将所有System从EntitySystem中抽取出来，让EntitySystem成为一个单纯的只管理组件与实体的类。

(2) 创建一个新的SystemManager类，用来管理所有的系统。

(3) 将Thread类和ThreadMgr类这两个类的基类替换为SystemManager类。

先来看一下新的SystemManager类，其定义在entity_system.h文件中。

```
class SystemManager : virtual public IDisposable {
public:
    SystemManager();
    virtual void Update();
    ...
protected:
    MessageSystem* _pMessageSystem;
    EntitySystem* _pEntitySystem;
    std::list<ISystem*> _systems;
    ...
};
```

从名字可以看出，SystemManager是一个管理系统的类。

SystemManager类的构造函数如下：

```
SystemManager::SystemManager() {
    _pEntitySystem = new EntitySystem(this);
    _pMessageSystem = new MessageSystem(this);
    _systems.emplace_back(new UpdateSystem());
    ...
}
```

SystemManager在构造函数中初始化了3个系统，分别是EntitySystem（实体系统）、MessageSystem（消息系统）和UpdateSystem（更新系统）。

（1）EntitySystem，负责所有组件和实体的管理，所有组件实例在这个类中都可以找到。如果是多线程，EntitySystem就只负责本线程中的组件。

（2）MessageSystem，负责处理从网络层或从别的线程中发来的Packet消息。

（3）UpdateSystem，处理需要不断更新的数据的组件。

除了EntitySystem之外，所有系统都继承自ISystem基类。前面介绍过EntitySystem类，严格来说它不算是一个系统，它没有动作，是所有系统的基础，因为它管理着所有实体与组件。

除了目前这几个系统之外，在后面还会有其他的系统。在SystemManager类中，只有MessageSystem和EntitySystem被单独处理，其他的系统类实例都放在一个队列中。对于SystemManager类来说，它并不关心每一个系统有什么作用，这是每个具体的系统自己去实现的。

SystemManager管理类比较简单，这些基于ISystem的系统之间有一些依赖关系，但并没有耦合。如果要做一个角色运动轨迹，例如MoveSystem系统，这个系统进入系统管理类，就可以很好地对某些玩家的运动做出计算，但如果没有它，整个框架不会有问题，框架还是一样运行，只是玩家不能运动而已。

这种不耦合的设计方式可以编写出非常复杂的功能，但在代码上给

人非常简洁的感觉。下面一个一个来看这些基本系统的工作原理。

8.1.1 实体系统EntitySystem

首先要讲的是EntitySystem类，作为数据管理类，它是所有系统的基础，其定义如下：

```
class EntitySystem : public IDisposable {
public:
    template <class T, typename... TArgs>
    T* AddComponent(TArgs... args);

    template <typename... TArgs>
    IComponent* AddComponentByName(std::string className, TArgs...

    template<class T>
    ComponentCollections* GetComponentCollections();

    void Update();
    ...
private:
    // 所有对象
    std::map<uint64, ComponentCollections*> _objSystems;
    ...
};
```

新的EntitySystem类中没有任何动作接口，现在它是一个纯数据类。在类中新增了一种ComponentCollections数据，用来保存实体或组件，该类的作用是将一系列相似的组件放在一起，即所有更新组件UpdateComponent实例是放在一个ComponentCollections实例中的，当需要取得更新组件时，取到对应的ComponentCollections实例即可，ComponentCollections定义在component_collections.h文件中：

```
class ComponentCollections :public IDisposable {
public:
    void Add(IComponent* pObj);
    void Remove(uint64 sn);
    ...
private:
    // uint64为其父类Entity的sn，一个sn不可能存在多个同一个类型的组件
    std::map<uint64, IComponent*> _objs;
    std::map<uint64, IComponent*> _addObjs;
```

```
std::list<uint64> _removeObjs;
};
```

ComponentCollections类的数据组织和之前提到的CacheRefresh类有相似之处，它是一个组件的集合，为了避免死锁，增加数据或删除数据时都会提前缓存，而后在下一帧处理。

框架中通过EntitySystem类来创建组件，创建组件时ComponentCollections实例被创建，其代码如下：

```
template <class T, typename ... TArgs>
T* EntitySystem::AddComponent(TArgs... args) {
    auto pComponent = DynamicObjectPool<T>::GetInstance()->MallocO
    AddComponent(pComponent);
    return pComponent;
}
template<class T>
inline void EntitySystem::AddComponent(T* pComponent) {
    const auto typeHashCode = pComponent->GetTypeHashCode();
    auto iter = _objSystems.find(typeHashCode);
    if (iter == _objSystems.end()) {
        _objSystems[typeHashCode] = new ComponentCollections(pComp
    }
    auto pEntities = _objSystems[typeHashCode];
    pEntities->Add(dynamic_cast<IComponent*>(pComponent));
}
```

还记得之前提到过的，在整个框架中有3个可以创建组件的途径。不论是哪种情况，生成组件的函数都是由EntitySystem提供的，也就是上面定义的AddComponent或AddComponentByName函数。这两个函数调用了类工厂函数，前面已经讲解过，这里不再多说。

以创建监听类为例，看看它的实例是如何生成并被放置在ComponentCollections集合中的。在进程启动时，线程管理类调用了CreateComponent函数，意图创建一个NetworkListen实例。最终，在某个线程中调用了AddComponentByName函数，NetworkListen类会被创建出来。创建出来的实例进入了一个ComponentCollections实例，EntitySystem的工作也就完成了。

EntitySystem现在的任务只是管理这些组件的实例。EntitySystem类中也有一个Update函数，但是新函数Update的意义和之前完全不一样：

```
void EntitySystem::Update() {  
    for (auto iter : _objSystems) {  
        iter.second->Swap();  
    }  
}
```

在EntitySystem::Update中，现在只做了一件事情，就是对于每一个ComponentCollections类调用了Swap函数。在ComponentCollections类定义中，可以看到它采用了CacheRefresh的方式，有3组数据：一组是当前正常运行有效的组件集合，一组是新增的组件数据，另一组是需要删除的组件。Swap函数就是将新增与删除的组件合并到有效组件集合中。之所以分成3组是为了不形成死循环。

实体和组件是一个相互循环的结构，实体可以创建无数个实体作为自己的组件，这些创建出来的实体也可以创建无数个新的实体，这个逻辑可以无限地嵌套下去。

当游戏功能足够复杂时，就会出现这样一种情况：在某一帧对所有拥有UpdateComponent组件的实体进行遍历更新操作时，其中一个实体A触发了某种特殊情况，生成一个新实体B，而生成的这个新实体B也有一个UpdateComponent组件。这时，整个更新组件集合，也就是管理UpdateComponent集合的ComponentCollections类的数据会发生改变，但这显然不是应该改变的时机，因为它还在循环遍历执行更新操作。

因此，不论是增加还是删除组件都放到下一帧去执行，这样可以有效地避免冲突。

既然EntitySystem的Update什么也没有做，那么读者一定会好奇NetworkListen的Update函数是怎么被调用的呢？这就要提到新的更新系统UpdateSystem。

8.1.2 更新系统UpdateSystem

更新系统由两个类组成：一个是组件类UpdateComponent；另一个是系统类UpdateSystem。这两个类的关系是，UpdateComponent相当于一个标记，它在某个实体上打上了一个需要更新的标记，而UpdateSystem是通过EntitySystem找到这些有标记的实体进行更新操作。

关键点1：组件UpdateComponent

还是以NetworkListen为例来看代码有什么不同。

```
class Network : public Entity<Network>, public INetwork { }  
class NetworkListen :public Network, public IAwakeFromPoolSystem<s
```

从定义可以看出，类NetworkListen变得更简洁了，去掉了IUpdateSystem接口的实现函数。那么NetworkListen是如何实现更新的呢？关键代码在它的唤醒函数中：

```
void NetworkListen::AwakeFromPool(std::string ip, int port) {  
    // update  
    auto pUpdateComponent = AddComponent<UpdateComponent>();  
    pUpdateComponent->UpdateFunction = BindFunP0(this, &NetworkLis  
    ...  
    return;  
}
```

在NetworkListen被初始化时增加了UpdateComponent组件。这个组件加入之后，更新的操作就可以转移给该组件。NetworkListen类的更新函数被绑定到了这个新创建的组件上。

按照这种思路，我们可以对任何一个类添加UpdateComponent组件，而不破坏这个类本身的数据，也不再需要过多的继承。所有的类都是扁平的，没有层次关系。

接下来，这个绑定的函数在UpdateComponent组件中是如何起作用的呢？先来看看组件的定义，它的定义在update_component.h文件中：

```
class UpdateComponent :public Component<UpdateComponent>, public I
public:
    void AwakeFromPool() override;
    void BackToPool() override;
    std::function<void()> UpdateFunction{ nullptr };
};
```

组件相对简单，有一个std::function用于回调实体的更新函数，毕竟每个实体的更新操作都是不一样的。在NetworkListen类初始化的代码中，将NetworkListen::Update函数绑定到了UpdateComponent组件的std::function变量上，这个函数会在更新系统UpdateSystem中被执行。

关键点2：系统UpdateSystem

系统类UpdateSystem定义在update_system.h文件中：

```
class UpdateSystem : virtual public ISystem {
public:
    void Update(EntitySystem* pEntities) override;
};
```

所有的System都继承自ISystem类，都必须实现Update(EntitySystem*pEntities)函数。下面是UpdateSystem的更新操作代码：

```
void UpdateSystem::Update(EntitySystem* pEntities) {
    auto pCollections = pEntities->GetComponentCollections<UpdateC
    if (pCollections == nullptr)
        return;
    pCollections->Swap();
    auto lists = pCollections->GetAll();
    for (const auto one : lists) {
        const auto pComponent = one.second;
```

```

        const auto pUpdateComponent = static_cast<UpdateComponent*>
        pUpdateComponent->UpdateFunction();
    }
}

```

在上面的代码中，先从EntitySystem中取出所有UpdateComponent组件进行遍历，执行其组件绑定的更新函数，以达到每个组件更新的目的。图8-1展示了这个流程。最外层的调用入口只有一个，也就是SystemManager::Update函数。在SystemManager中，每一帧都会调用所有系统的Update函数，而对于UpdateSystem而言，UpdateSystem::Update的作用就是遍历所有UpdateComponent组件绑定的更新函数。NetworkListen类的更新函数Update只是其中之一。

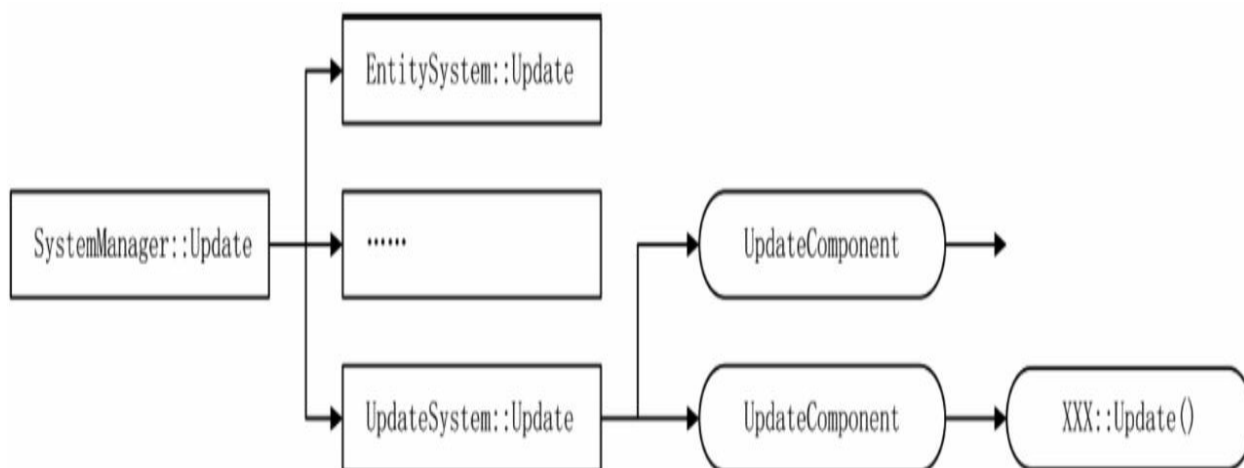


图8-1 SystemManager更新流程

虽然EntitySystem实体和UpdateComponent相互拥有对方的指针，但是从本质上来说，它们是两个相互独立的类。EntitySystem实体提供一个绑定函数，而UpdateComponent组件负责执行该绑定函数。对于每个UpdateComponent实例来说，它不了解，也不关心实EntitySystem体到底在更新函数中做了些什么，整个UpdateSystem系统只完成更新函数的调用。

8.1.3 消息系统MessageSystem

一个基本的框架除了基本的更新操作之外，还有一个重要的功能是协议的收发。在最初的框架中，将这两个功能融合到了ThreadObject基类中。随后，有了EntitySystem类之后，协议的收发变成了继承IMessageSystem接口。

现在有了消息系统，我们为协议的处理写了一个名为MessageSystem的系统。它涉及的文件比较多，有message_callback.h、message_component.h、message_system.h和message_system_help.h。

关键点1：组件MessageComponent

重点来看组件MessageComponent，其定义在message_component.h文件中，定义如下：

```
class MessageComponent : public Component<MessageComponent>, public
public:
    void AwakeFromPool(MessageCallbackFunctionInfo* pCallback) over
    void BackToPool() override;
    bool IsFollowMsgId(Packet* packet) const;
    void ProcessPacket(Packet* packet) const;
protected:
    IMessageCallbackFunction* _pCallBackFuns{ nullptr };
};
```

该组件承担了之前IMessageSystem接口的工作，主要任务是注册每个组件关心的协议并绑定处理函数。根据不同的需求，参数可以选择不同的MessageCallbackFunctionInfo实现类，具体代码在之前已经介绍过了，这里不再赘述。回顾一下之前的消息系统。

- 第一版：从网络底层读上来数据，将这些数据组织成一个Packet类，然后将这些类放到各个线程中，线程中的每个对象继承自基类ThreadObject，每个对象都要实现处理消息的基础函数。
- 第二版：在深入编码之后，我们发现并不是所有组件都需要处理

消息。然后设计了一个IMessgaeSystem接口，继承了该接口的组件可以收到Packet消息。

现在，将IMessgaeSystem接口去掉了，只要有了MessageComponent组件就可以处理消息，这将产生一些非常灵活而便捷的操作。例如，在某些条件下，实体A可以处理1、2号协议，当它的状态发生改变时，可以删除这个MessageComponent组件，增加一个新的MessageComponent组件，这时可以处理3、4号协议。对于一些复杂的情况，也可以更改1、2号协议的处理方式。

关键点2：系统MessageSystem

在之前的消息处理中，当Packet到来之后，EntitySystem收到数据并将它保存在本地。作为一个管理组件的类，EntitySystem存储Packet数据显然是不太合理的。现在，需要将这一部分数据转到MessageSystem系统中。任何时候调用MessageSystem：：AddPacketToList都可以将一个Packet缓存到MessageSystem系统中。MessageSystem类的定义如下：

```
class MessageSystem :virtual public ISystem {
public:
    void Update(EntitySystem* pEntities) override;
    void AddPacketToList(Packet* pPacket);
    ...
private:
    static void Process(Packet* pPacket, std::map<uint64, IComponen
private:
    std::mutex _packet_lock;
    CacheSwap<Packet> _cachePackets;
    ...
};
```

这些函数的具体实现和用法前面已经讲解过了，这里不再赘述。实现方式并没有多大改变，改变的是它的调用方式。

8.1.4 测试执行效率

综上所述，完成了现阶段所有功能的优化，在08_01_system目录中的工程，所有实体去掉了IUpdateSystem接口与IMessageSystem接口。现在整体框架变得更加扁平。以登录类Account为例，类的定义更加简洁，它的新定义如下：

```
class Account :public Entity<Account>, public IAwakeFromPoolSystem
```

而之前的定义如下：

```
class Account :public Component<Account>, public IMessageSystem, p
```

新定义更加简洁。在完成本小节之前，对整个框架的性能做一个测试。看一看运行1000个账号的执行效率：

```
[root@localhost bin]# ./logind
...
Login-Connecting over. time:0.354992s
Login-Connected over. time:1.14729s
Login-LoggedIn over. time:1.72766s
Login-SelectPlayer over. time:2.05091s
```

多次反复测试，从连接到login进程，经过查询角色、选中角色这一系列事件，1000个账号在2秒左右完成。

现在有一个新的问题凸显出来，当我们测试工程时变得非常麻烦，因为需要同时启动dbmgr和login。如果能够合并起来，只要启动一个进程就好了。因此，下一节要加入一个新的工程，在所有代码不改动的前提下，将dbmgr工程和login工程的所有功能合并到一个进程中，方便测试使用。

8.2

all in one工程

在以往的工程中，将一个工程合并到另一个工程是一个相当大的挑战，本书中的框架比较特别，严格遵照Actor模型的原则做了解耦的操作，因此将dbmgr和login合并成一个工程非常简单。如果能将它们合并在一起同时启动，那么可以极大地方便开发与提高测试效率。

8.2.1 新工程allinone

对于底层框架来说，login和dbmgr为什么功能不一样，完全是因为它们加载了不同的组件。鉴于此，增加一个名为allinone的工程，将login和dbmgr中所有的组件加载到一个进程中，是不是可以实现所有的功能了呢？答案是肯定的。

打开本节源代码的08_02_allinone目录，使用make-all.sh脚本进行编译，编译完成之后，在bin目录下会生成4个可执行文件：allinone、dbmgr、login和robots。现在启动服务端，可以不需要启动dbmgr和login，而是直接启动allinone，可以达到相同的效果。

为了共用代码，将login工程的组件添加整理成函数，放在login.h文件中，将dbmgr工程中的组件添加整理成函数，放在dbmgr.h文件中。

这样做的好处在于，不需要在allinone工程中再编写一遍加载login或dbmgr组件的代码。如果在allinone工程中再编写一段相似的加载代码，就容易忘记在allinone工程中加入login组件，造成组件遗漏的情况。牢记编码的一个原则，相似的代码绝不重复出现，重复的代码在后期维护困难，且易于产生Bug。下面来看login.h文件的内容：

```
inline void InitializeComponentLogin(ThreadMgr* pThreadMgr) {
    pThreadMgr->CreateComponent<RobotTest>();
    pThreadMgr->CreateComponent<Account>();
}
```

在该函数中仅加载了Account组件和一个测试用的RobotTest组件。login作为一个单独的线程启动时，增加的并不止这两个组件，它的main函数如下：

```
int main(int argc, char* argv[]) {
    ...
    auto pThreadMgr = ThreadMgr::GetInstance();
    InitializeComponentLogin(pThreadMgr);
    auto pYaml = Yaml::GetInstance();
    auto pCommonConfig = pYaml->GetIPEndPoint(curAppType);
    pThreadMgr->CreateComponent<NetworkListen>(pCommonConfig->Ip,
```

```
pThreadMgr->CreateComponent<NetworkConnector>((int)APP_TYPE::A
...
return 0;
}
```

可以看到，在进程中还有一个NetworkListen组件用于监听，另一个NetworkConnector组件用于连接DB进程。不过，这两个组件在allinone工程中是不需要的。

同理，工程dbmgr的加载组件函数在dbmgr.h文件中。目前只使用了MysqlConnector组件：

```
inline void InitializeComponentDBMgr(ThreadMgr* pThreadMgr) {
    pThreadMgr->CreateComponent<MysqlConnector>();
}
```

现在，在新工程allinone中引用这两个文件，工程allinone的main函数如下：

```
int main(int argc, char* argv[]) {
    const APP_TYPE curAppType = APP_TYPE::APP_ALLINONE;
    ServerApp app(curAppType, argc, argv);
    app.Initialize();

    auto pThreadMgr = ThreadMgr::GetInstance();

    // dbmgr
    InitializeComponentDBMgr(pThreadMgr);

    // login
    InitializeComponentLogin(pThreadMgr);

    auto pYaml = Yaml::GetInstance();
    auto pCommonConfig = pYaml->GetIPEndPoint(curAppType);
    pThreadMgr->CreateComponent<NetworkListen>(pCommonConfig->Ip,
    ...
    return 0;
}
```

在上面的代码中加入了dbmgr和login的所有组件，同时打开了一个监听端口，用于客户端登录使用，这个监听端口与login工程中的配置一致。其配置文件在engine.yaml文件中，新增了一个allinone的属性，用于allinone工程的配置。

整个allinone只有一个main.cpp文件，其他都是引用的login和dbmgr的文件。而在Linux系统下，我们需要修改CMakeLists.txt文件中对源代码目录设置的相关部分。CMakeLists.txt文件的部分编码如下：

```
aux_source_directory(. SRCS)
aux_source_directory(..../login SRCS)
list(REMOVE_ITEM SRCS "..../login/main.cpp")
aux_source_directory(..../dbmgr SRCS)
list(REMOVE_ITEM SRCS "..../dbmgr/main.cpp")
```

不论是dbmgr还是login工程，在编译执行文件时只需要将自己目录下的文件编入即可。但是allinone工程需要将login和dbmgr的源代码都纳入编译目录，不过需要剔除两个工程中的main.cpp文件，毕竟每个进程中只能有一个main函数。

8.2.2 协议是如何被转发的

合并组件之后，另一个让人疑惑的地方是发送协议。在启动login进程时会向dbmgr发起一个网络连接，但是现在它们在一个进程中，没有网络连接了，那么应该如何发送消息呢？

以login进程向dbmgr发送查询玩家的数据协议为例回顾一下这段代码：

```
void Account::HandleAccountCheckToHttpRs(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::AccountCheckToHttpRs>
    ...
    // 验证成功，向DB发起查询
    if (proto.return_code() == Proto::AccountCheckReturnCode::ARC_
        Proto::QueryPlayerList protoQuery;
        protoQuery.set_account(pPlayer->GetAccount().c_str());
        MessageSystemHelp::SendPacket(Proto::MsgId::L2DB_QueryPlay
    }
}
```

当需要将一个消息发送到指定的进程时，最终调用的是MessageSystemHelp类中的SendPacket函数。

```
void MessageSystemHelp::SendPacket(const Proto::MsgId msgId, googl
    auto packet = CreatePacket(msgId, 0);
    packet->SerializeToBuffer(proto);
    SendPacket(packet, appType, appId);
}
void MessageSystemHelp::SendPacket(Packet* packet, APP_TYPE appTyp
    if ((Global::GetInstance()->GetCurAppType() & appType) != 0) {
        // 正好在当前进程中，直接转发，例如curapptype==all的时候
        DispatchPacket(packet);
    } else {
        auto pNetworkLocator = ThreadMgr::GetInstance()->GetEntity
        GetComponent<NetworkLocator>();
        auto pNetwork = pNetworkLocator->GetNetworkConnector(appTy
        if (pNetwork != nullptr) {
            packet->SetSocket(pNetwork->GetSocket());
            pNetwork->SendPacket(packet);
        } else {
```

```
        LOG_ERROR("can't find network. appType:" << AppTypeMgr  
    }  
}  
}
```

在上面的代码中，首先创建了一个Socket默认值为0的Packet，接着判断AppType。AppType枚举很早就在使用了，但是没有认真介绍过它。它标记了一个进程的类型，其定义如下：

```
enum APP_TYPE {  
    APP_DB_MGR = 1,  
    APP_GAME_MGR = 1 << 1,  
    APP_SPACE_MGR = 1 << 2,  
    APP_LOGIN = 1 << 3,  
    APP_GAME = 1 << 4,  
    APP_SPACE = 1 << 5,  
    APP_ROBOT = 1 << 6,  
    APP_APPMGR = APP_GAME_MGR | APP_SPACE_MGR,  
    APP_ALLINONE = APP_DB_MGR | APP_GAME_MGR | APP_SPACE_MGR | APP  
};
```

这个枚举值不是单纯的数字，而是使用了位标志。使用位标志是为了可以混合类型。在上面的代码中有一句：

```
if ((Global::GetInstance()->GetCurAppType() & appType) != 0)
```

这句代码兼容了所有可能的情况。我们可以任意将多个AppType融合到一个进程中，在发送数据时，如果发现目标进程正好在本进程当前的AppType中，那么要做的事情是使用DispatchPacket发送消息即可。

以dbmgr为例，需要向dbmgr发送数据时，当前的AppType为APP_ALLINONE，说明当前进程中有dbmgr的所有组件，这时只需要广播消息，dbmgr的组件就可以收到该消息。

使用allinone进程启动服务时，所有的组件都在一个进程中，这时GetCurAppType() & appType必然不会是0，这就形成了内部的转发包。

以上是发送的流程，那么接收时呢？当dbmgr收到查询玩家数据的Packet时，需要返回一个消息，它是如何返回的呢？还是以查询角色为例，MysqlConnector类收到查询角色协议的处理函数如下：

```
void MysqlConnector::HandleQueryPlayerList(Packet* pPacket) {
    auto protoQuery = pPacket->ParseToProto<Proto::QueryPlayerList>
        QueryPlayerList(protoQuery.account(), pPacket->GetSocket());
}
void MysqlConnector::QueryPlayerList(std::string account, SOCKET s
    ...
    MessageSystemHelp::SendPacket(Proto::MsgId::L2DB_QueryPlayerLi
}
```

在创建回应包时传入了pPacket->GetSocket()的值，假如dbmgr和login之间有真正的网络连接，那么这里一定是真正的Socket文件符，但是在allinone工程中，这里只可能为0。发送这个协议的函数如下：

```
void MessageSystemHelp::SendPacket(const Proto::MsgId msgId, const
    const auto pPacket = CreatePacket(msgId, socket);
    pPacket->SerializeToBuffer(proto); // 序列化结构
    SendPacket(pPacket);
}
void MessageSystemHelp::SendPacket(Packet* pPacket) {
    // 找不到Network，就向所有线程发送协议
    if (pPacket->GetSocket() == 0) {
        DispatchPacket(pPacket);
        return;
    }
    auto pNetworkLocator = ThreadMgr::GetInstance()->GetEntitySyst
    auto pNetwork = pNetworkLocator->GetNetworkConnector(pPacket->
    if (pNetwork != nullptr) {
        pNetwork->SendPacket(pPacket);
        return;
    }
    // 最后试着向listen发送数据
    pNetwork = pNetworkLocator->GetListen(NetworkTcpListen);
    pNetwork->SendPacket(pPacket);
}
```

在发送包的时候，若Socket等于0，则表示该包需要向所有线程进

行转发。完成这两个功能点，我们的allinone工程就完成了。

8.2.3 查看线程中的所有对象

编译08_02_allinone目录，运行一下，看看效果。

```
[root@localhost 08_02_allinone]# ./make-all.sh
[root@localhost 08_02_allinone]# make
```

最后在bin目录下多生成了一个allinone可执行文件。allinone已经具备了dbmgr和login的所有功能，可以启动robots进程进行登录测试。

现在整个框架有了多线程、多进程的配置，当我们不需要多进程时，启动allinone就变成了单进程；当我们不再需要多线程时，在engine.yaml文件中可以配置线程的个数，配置为0时只有主线程。

启动allinone进程之后，输入“thread-entity”查看线程中对象的分布情况，如表8-1所示。

使用单线程，所有的实体都在主线程中。在配置两个线程时，实际上再加上主线程就有了3个线程。Console类是放在主线程上的，所以表的右侧最后一个线程应该就是主线程，主线程上除了Console外，还有NetworkLocator类，而其他两个线程的组件是login和dbmgr所有的组件，基本是均匀分布的。

表8-1 单线程与多线程的内存对象

单 线 程	多 线 程
thread id:139630582573184 Console count:1 Account count:1 RobotTest count:1 NetworkListen count:1 MysqlConnector count:1 NetworkLocator count:1 total count:6	thread id:140470474872576 Account count:1 MysqlConnector count:1 total count:2 thread id:140470466479872 RobotTest count:1 NetworkListen count:1 total count:2 thread id:140470628587648 Console count:1 NetworkLocator count:1 total count:2

指令thread的功能由ConsoleCmdThread和ConsoleThreadComponent组件实现。ConsoleCmdThread发送指令，每个线程初始化时都加载了一个ConsoleThreadComponent组件，该组件的任务是处理ConsoleCmdThread发送的协议，收到协议时就会将当前线程中的数据打印出来。读者可以在工程中查看这两个组件的实现源代码。

8.2.4 测试执行效率

修改engine.yaml文件中allinone的线程配置，将其改为5个。也就是说，将login和dbmgr中的组件均匀地放到了5个线程中，robots线程也修改为5个。现在，运行1000个账号的执行效率如下：

```
[root@localhost bin]# ./allinoned
...
Login-Connecting over. time:0.450946s
Login-Connected over. time:0.454s
Login-Logged over. time:1.68313s
Login-SelectPlayer over. time:1.71794s
```

1000个账号从登录到选择角色花费了不到2秒，因为有多线程，即使将所有的组件都放进一个进程中，其效率也不太会受到影响。

线程分类

在8.2节中，从打印出来的线程数据中会发现NetworkListen组件被随机放置到了某个线程中，但这并不是我们想要的，我们希望NetworkListen类可以独占一个线程。除此之外，我们可能还希望能启动两个或多个MysqlConnector组件，毕竟1000个账号同时登录时，用几个线程来读取数据库的数据肯定快过用一个线程来读取数据库的数据。

在dbmgr进程中，创建MysqlConnector组件时调用的函数为CreateComponent。将随机挑选一个进程生成对象实例。该功能现在已经无法满足我们的需要，为了保证性能，希望每个MysqlConnector可以独占进程，这又引出了一个新的问题。

假设有两个线程，每个线程中都有MysqlConnector实例，那么查询玩家的协议是否会被执行两次？毕竟现在所有协议是分发给所有线程的。

为了解决这两个问题，有必要对线程类型进行一个规划，有些线程对于协议的处理是互斥的。所谓互斥，就是像MysqlConnector组件一样，只需要在多个线程中挑选一个线程处理即可，而另一些线程是完全平等的，如逻辑线程。因此，定义了线程枚举ThreadType。

```
enum ThreadType {
    MainThread = 1 << 0,
    ListenThread = 1 << 1,    // 监听线程
    ConnectThread = 1 << 2,
    LogicThread = 1 << 3,    // 逻辑线程
    MysqlThread = 1 << 4,    // 数据库线程
    AllThreadType = MainThread | LogicThread | ListenThread | Conn
};
```

本例中的代码参考08_03_thread目录。在线程管理类ThreadMgr中，维护的不再是一个又一个线程实例对象，而是一个又一个线程集合。这些线程集合中存放着某一类线程的所有实例，编写着一些属于这些线程的规则。

```
class ThreadMgr :public Singleton<ThreadMgr>, public SystemManager
...
private:
    std::map<ThreadType, ThreadCollector*> _threads;
};
```

从ThreadMgr类中的变量定义可以看出，某种类型的线程放在一个ThreadCollector集合中，例如所有处理逻辑的线程都放在一个ThreadCollector集合中，而所有处理数据库的线程都放在另一个ThreadCollector集合中。ThreadCollector类在thread_collector.h文件中，其定义如下：

```
class ThreadCollector :public IDisposable {
public:
    ...
    virtual void HandlerMessage(Packet* pPacket);
    virtual void HandlerCreateMessage(Packet* pPacket);
protected:
    ThreadType _threadType;
    CacheRefresh<Thread> _threads;
    size_t _index{ 0 };
};
```

ThreadCollector类维护了某种类型线程实例的列表。除了ThreadCollector类之外，还有一个它的扩展类ThreadCollectorExclusive。

```
class ThreadCollectorExclusive :public ThreadCollector {
public:
    ...
    virtual void HandlerMessage(Packet* pPacket) override;
    virtual void HandlerCreateMessage(Packet* pPacket) override;
```

```
private:
    size_t _index;
};
```

ThreadCollectorExclusive类处理的是互斥的情况，可用于MySQL线程上。逻辑线程和数据库线程最大的区别在于对于协议的处理。当一个协议到来时，数据库线程只需要挑选一个线程实例处理协议即可，而逻辑线程需要将协议发送到所有线程实例中。以下是两个类对于协议的处理方式：

```
void ThreadCollector::HandlerMessage(Packet* pPacket) {
    auto pList = _threads.GetReaderCache();
    for (auto iter = pList->begin(); iter != pList->end(); ++iter)
        (*iter)->GetMessageSystem()->AddPacketToList(pPacket);
}

void ThreadCollectorExclusive::HandlerMessage(Packet* pPacket) {
    auto vectors = *(_threads.GetReaderCache());
    vectors[_index]->GetMessageSystem()->AddPacketToList(pPacket);
    _index++;
    _index = _index >= vectors.size() ? 0 : _index;
}
```

在逻辑线程集合中收到一个协议，一定是广播出去的。例如，收到网络断开协议，因为不知道在整个线程中有哪些组件关心这些协议，所以一定是对逻辑进程集合整个广播。而对于一个存储协议来说，处理流程则不相同，为了让数据库操作分散，每一个数据库线程都是完全一样的，它们有同样的组件。这时ThreadCollectorExclusive类的执行方式是采用轮询的方式，以达到均衡的目的。

打开本例中的res目录，修改engine.yqml配置。在allinone下，将逻辑线程数量改为1，而数据库线程thread_mysql改为2。运行allinone，使用指令“thread-entity”查看内存的线程及其中的所有组件。打印信息如下：

```
[DEBUG] - *****
thread id:140593884665984
thread type:MainThread
7Console count:1
```

```

    14NetworkLocator count:1
total count:2
[DEBUG] - *****
thread id:140593722488576
thread type:MysqlThread
    14MysqlConnector count:1
total count:1
[DEBUG] - *****
thread id:140593714095872
thread type:MysqlThread
    14MysqlConnector count:1
total count:1
[DEBUG] - *****
thread id:140593730881280
thread type:LogicThread
    7Account count:1
    9RobotTest count:1
total count:2
[DEBUG] - *****
thread id:140593705703168
thread type:ListenThread
    13NetworkListen count:1
total count:1

```

进程中一共有5个线程，包括主线程、监听线程、两个数据库线程和一个逻辑线程。将NetworkListen监听放到了一个专门的进程中。逻辑线程LogicThread中有Account类以及RobotTest类，而MysqlThread类型的两个线程是完全一模一样的。现在分类之后，各种线程的功能更加明确了。

为了进一步测试数据库线程是否按我们的要求在工作。打开mysql_msg.cpp文件，在选择角色函数中加入一行打印数据，在打印数据中加上线程Id。启动robots进程并登录账号，就会看到每次登录时选择的是不同的数据库线程，有兴趣的读者可以自行测试。

l AwakeSystem接口与对象池

在前面的示例代码中，采用的是全局对象池，即一种类型的对象池只有一个实例。在多线程中使用对象池，操作已使用、未使用的集合时进行了加锁操作。

本节继续优化框架，将全局对象池变更为线程对象池。一般来说，线程中创建的对象都在线程内使用，这些对象是不需要加锁的。

本节的源代码位于08_04_pool目录中，在这个示例中将消灭所有以创建方式创建组件的操作，所有组件将通过线程中的对象池来生成，同时我们还会写一个指令用于随时查看各个线程中的所有组件。

8.4.1 DynamicObjectPoolCollector对象池集合

在前面的工程中，需要生成一个组件时要取得该组件的对象池实例，其代码如下：

```
DynamicObjectPool<Class>::GetInstance()
```

如果有多个不同的类对象池，就有多个DynamicObjectPool实例，为了便于管理，引入一个新类DynamicObjectPoolCollector，其作用是维护DynamicObjectPool集合。

这个新类由SystemManager来管理，相当于每个ECS体系中都有一个对象池管理类。DynamicObjectPoolCollector类的定义文件为object_pool_collector.h，来看看它的定义：

```
class DynamicObjectPoolCollector : public IDisposable {
public:
    DynamicObjectPoolCollector(SystemManager* pSys);
    void Dispose();
    template<class T>
    IDynamicObjectPool* GetPool();
    void Update();
private:
    std::map<uint64, IDynamicObjectPool*> _pools;
    SystemManager* _pSystemManager{ nullptr };
};
```

该类中提供了一个GetPool模板函数用于提取适合的对象池，实现代码如下：

```
template<class T>
IDynamicObjectPool* DynamicObjectPoolCollector::GetPool() {
    const auto typeHashCode = typeid(T).hash_code();
    auto iter = _pools.find(typeHashCode);
    if (iter != _pools.end()) {
        return iter->second;
    }
}
```

```
    auto pPool = new DynamicObjectPool<T>();  
    pPool->SetSystemManager(_pSystemManager);  
    _pools.insert(std::make_pair(typeHashCode, pPool));  
    return pPool;  
}
```

从上面的代码可以看出，当需要某种类型的对象池时，如果DynamicObjectPoolCollector类中还没有该对象池，就无条件地创建一个。

DynamicObjectPoolCollector类实例是在SystemManager类创建时产生的，其代码如下：

```
SystemManager::SystemManager() {  
    _pEntitySystem = new EntitySystem(this);  
    _pMessageSystem = new MessageSystem(this);  
    ...  
    _pPoolCollector = new DynamicObjectPoolCollector(this);  
}
```

在SystemManager创建时，同时创建了一个DynamicObjectPoolCollector类来管理当前整个体系中的所有对象。因为每一个线程都是一个ECS体系结构，也就是说每个线程有一个自己的DynamicObjectPoolCollector对象池管理实体。

从全局的对象池管理到SystemManager中的对象池管理，在内存上并没有任何优化，将对象池作为全局对象时，一个关于ConnectObj类的对象池一旦生成，将生成50个缓存。当这个对象池放在线程上时，同样也是50个缓存，它不会变少，也不会增多，但是在代码上去掉了锁。

一个对象池实例不会对所有线程共用，它一定属于某个特定的线程，每个线程有自己的对象池实例。当我们对进程和线程进行合并时，合并到最后，整个变成单进程、单线程，此时全局只有一个SystemManager，而对象池管理类也只有一个。

按之前的逻辑，对象池管理类是一个全局对象时，即使是单进程，加锁的代码依然需要运行。而现在优化之后，对象池变成了ECS框架的一部分，变得更容易理解了。

在旧框架中，当我们需要创建一个对象时，往往是这样调用的：

```
DynamicObjectPool<T>::GetInstance()->MallocObject()
```

函数MallocObject必定是加锁的。现在不论在哪一个组件中创建对象，我们都可以如下调用：

```
auto pCollector = pSysMgr->GetPoolCollector();  
auto pPool = (DynamicObjectPool<T>*)pCollector->GetPool<T>();  
T* pComponent = pPool->MallocObject();
```

生成组件时，从本线程中的SystemManager类中取出本线程的对象池管理类，再由对象池管理类找到该类型的对象池，最后生成对象。调用函数MallocObject不再加锁。

8.4.2 全局单例对象

在本节的示例中，除了ThreadMgr类之外，几乎去掉了所有全局单例对象。主要原因是单例太难管理，单例使用前要调用生成函数，退出程序时也需要调用销毁函数。在实际编码中，在生成单例类时，要么忘记调用Instance函数来生成它，要么忘记编写DestroyInstance函数来销毁它。

现在，有了对象池，将所有类对象的生成都直接放到对象池中，也包括这些单例类，例如Yaml、ResPath类等。先看看这些全局类的生成方式：

```
void ThreadMgr::InitializeGloablComponent (APP_TYPE ppType, int app
// 全局Component
GetEntitySystem()->AddComponent<ResPath>();
GetEntitySystem()->AddComponent<Log4>(ppType);
GetEntitySystem()->AddComponent<Yaml>();
GetEntitySystem()->AddComponent<NetworkLocator>();
...
}
```

现在从上面的代码中可以看出，Log4、Yaml这些单例类都变成了一个个组件，对于整个框架来说更容易管理。但这引出了一个新的问题，Log4、Yaml都是只需要生成一个对象，而组件生成时使用的是对象池，在对象池中生成多个缓冲不是一种浪费吗？

为了从根本上解决这个问题，我们引入了两个IAwakeSystem接口。

理论上来说，IAwakeSystem并不是典型的ECS中的System。顾名思义，AwakeSystem这是一个唤醒操作。对于接触过Unity的读者来说，对Awake函数应该并不陌生，在Unity中创建一个组件，在Start、Update函数调用之前必定会调用Awake函数。这与我们的IAwakeSystem系统有异曲同工之效。

在整个框架中，所有的组件都继承自IAwakeSystem或IAwakeFromPoolSystem接口。打开system.h文件，其定义如下：

```

template <typename... TArgs>
class IAwakeSystem : virtual public ISystem {
public:
    IAwakeSystem() = default;
    virtual ~IAwakeSystem() = default;
    virtual void Awake(TArgs... args) = 0;
    static bool IsSingle() { return true; }
};
template <typename... TArgs>
class IAwakeFromPoolSystem : virtual public ISystem {
public:
    IAwakeFromPoolSystem() = default;
    virtual ~IAwakeFromPoolSystem() = default;
    virtual void Awake(TArgs... args) = 0;
    static bool IsSingle() { return false; }
};

```

上面两个接口中都有一个类的静态函数IsSingle，即当我们拿到要生成的类时，就知道它是不是一个单例。下面来看一下Yaml类的定义：

```

class Yaml : public Component<Yaml>, public IAwakeSystem<> { }

```

对比一下网络连接类：

```

class ConnectObj : public Entity<ConnectObj>, public IAwakeFromPool

```

Yaml类读取配置文件全局只需要一个，所以它只是一个单例，而ConnectObj可以生成多个实例，所以需要为它多准备一些实例备用。在对象池生成对象时，调用了IsSingle函数，调用方式如下：

```

template <typename T>
template <typename ... Targs>
T* DynamicObjectPool<T>::MallocObject(Targs... args) {
    if (_free.size() == 0) {
        if (T::IsSingle()) {
            T* pObj = new T();
            pObj->SetPool(this);
            _free.push(pObj);
        } else {

```

```

        for (int index = 0; index < 50; index++) {
            T* pObj = new T();
            pObj->SetPool(this);
            _free.push(pObj);
        }
    }
    ...
}

```

如果空闲队表中没有对象了，又不是单例，就一次初始化50个对象。如果是单例，那么只需要创建一个对象。当然，即使是单例，从上面的代码可以看出，如果还想创建的话，依然可以继续创建第二个实例。IsSingle这个静态函数的任务是区分一次创建一个或多个备用实例。区分一个对象是不是单例，有时还需要从线程的角度出发，例如NetworkListen的定义如下：

```

class NetworkListen :public Network, public IAwakeSystem<std::stri

```

我们将它定义为一个单例，从逻辑上来说，每一个NetworkListen都是独占线程的，所以备用实例没有什么作用。类似的还有login工程中的Account类，这个类负责登录时的账号管理，本质上来说它也不需要备用实例。

8.4.3 查看线程中的所有对象

现在，对象都是由对象池中创建出来的。数据易于管理，便于查看。为了监控这些数据，工程中新写了一个查看当前内存中所有对象的指令“thread-pool”。

编译08_04_pool目录下的所有工程，执行bin目录下的allinone。启动之前检查一下engine.yaml中数据库的IP与端口是否与本地的配置一致。为了便于查看数据，将thread_logic和thread_mysql都设为1。启动成功之后，再启动robots进程，批量登录1000个账号，大约需要2秒钟，关闭robots进程，回到allinone的终端，输入“thread-pool”指令。如果不了解当前线程有哪些可以使用的指令，可以输入help查看。使用help指令可以列出当前进程中可以使用的所有指令。

执行了“thread-pool”指令之后，可以看到当前进程中所有的对象，指令的实现代码不做详细解释，有兴趣的读者可以看看源代码。

下面分析一下这个打印数据。现在进程中有4个线程，包括主线程、逻辑线程、数据库线程和监听线程。在配置多线程时，一定会多出一个监听线程，而在单线程中，thread_logic和thread_mysql的值都为0，监听组件是放在主线程中的，不会另启线程。

先来看看主线程中的所有组件：

```
- thread type:MainThread
- total: 1    free: 0    use: 1    call: 1    16CreateCompone
- total: 50   free: 49   use: 1    call: 1    15UpdateCompone
- total: 1    free: 0    use: 1    call: 1    7Console
- total: 50   free: 48   use: 2    call: 2    16MessageCompon
- total: 1    free: 0    use: 1    call: 1    22ConsoleThread
- total: 1    free: 0    use: 1    call: 1    4Yaml
- total: 1    free: 0    use: 1    call: 1    4Log4
- total: 1    free: 0    use: 1    call: 1    14NetworkLocato
- total: 1    free: 0    use: 1    call: 1    7ResPath
```

第一列total表示在对象池中该对象一共有多少个实例，包括已使用的和还没有使用的。第二列为空闲可以使用的对象个数，第三列为正

在使用中的对象个数，第四列call表示之前对象被调用了多少次，最后一列是类名。

在主线程中放置了全局组件Console、Yaml、Log4、NetworkLocator和ResPath。类名前面的数字是Linux系统底层类名函数的显示，但不影响我们辨识。从上面的数据可以看出，正在使用的UpdateComponent组件有一个，MessageComponent组件有两个。UpdateComponent组件来自于Console组件，而两个MessageComponent组件分别来自CreateComponentC和ConsoleThreadComponent组件。CreateComponentC和ConsoleThreadComponent这两个组件是每个线程必备的组件，CreateComponentC组件是用于跨线程创建组件的组件，而ConsoleThreadComponent组件则是辅助查看内存数据的组件，它是一个测试阶段的工具组件。

再来看一下数据库线程中组件的统计情况，除了基础的组件之外，还多了一个数据库专有的MysqlConnector组件：

```
- thread type:MysqlThread
- total: 1 free: 0 use: 1 call: 1 16CreateCompone
- total: 50 free: 47 use: 3 call: 3 16MessageCompon
- total: 1 free: 0 use: 1 call: 1 22ConsoleThread
- total: 1 free: 0 use: 1 call: 1 14MysqlConnecto
```

在数据库线程中，特有的组件为MysqlConnector，在对象池中，我们只为它创建了一个实例，也没有准备备用实例，因为不需要。

监听线程的特有组件是NetworkListen和ConnectObj，监听线程的对象分配如下：

```
- thread type:ListenThread
- total: 1 free: 0 use: 1 call: 1 16CreateCompo
- total: 1 free: 0 use: 1 call: 1 13NetworkList
- total: 50 free: 49 use: 1 call: 1 15UpdateCompo
- total: 50 free: 47 use: 3 call: 3 16MessageComp
- total: 1 free: 0 use: 1 call: 1 22ConsoleThre
- total:1050 free: 1050 use: 0 call: 1001 10ConnectObj
```

在上面的数据中，空闲的ConnectObj有1050个，正在使用的是0个，使用过1001次。这个数据是在登录了1000个账号并关闭之后的数据。1000个Robot会启动1000个ConnectObj。而RobotMgr类也会有一个网络连接，所以会多一个ConnectObj。连接关闭之后，这些ConnectObj被回收，放置到了空闲队列中。批量生成时是按50个批量生成的，所以现有总对象是1050个。

最后来看逻辑线程的数据：

```
- thread type:LogicThread
- total: 1 free: 0 use: 1 call: 1 16CreateCompone
- total:1000 free: 0 use:1000 call:1000 15UpdateCompone
- total:1000 free: 0 use:1000 call:1000 18HttpRequestAc
- total: 1 free: 0 use: 1 call: 1 7Account
- total: 1 free: 0 use: 1 call: 1 9RobotTest
- total: 50 free: 46 use: 4 call: 4 16MessageCompon
- total: 1 free: 0 use: 1 call: 1 22ConsoleThread
```

在逻辑线程中，有两个主要的组件Account和RobotTest。Account用于账号登录验证，而RobotTest用于机器人登录情况的时长统计。除了基础组件之外，在逻辑线程中还有一个HttpRequestAccount组件被调用了1000次，这是用于账号HTTP请求的组件，现在正在使用的有1000个。显然这是一个Bug，因为数据已经取到了，但是对象却没有释放。同时每个HttpRequestAccount都有一个更新组件，有1000个UpdateComponent组件也没有被释放。

从上面的分析可以看出，对象池不仅提高了执行效率，它更是一个对内存对象进行管理的工具。对象池的数据分析可以使我们更容易对所有对象进行监控。

除了几条线程中的对象之外，还有一个特殊的组件对象Packet，打印信息中包含它所在对象池中的数据：

```
- total:8050 free: 40 use:8010 call:8010 6Packet
```

我们没有释放过Packet，每次都在创建，一共产生了近8000多个对象。Packet的释放是比较复杂的，因为一个Packet会进入一个或几个线

程中，线程的处理具有不确定性，不知道什么时候对这个Packet的处理就完成了。所以要高效重用Packet类需要有一套新的规则。

综上，现在有两个问题急需解决：一个问题是HttpRequestAccount组件没有被释放；另一个问题是Packet数据没有被释放。我们将这两个问题都归为销毁对象的问题。

主动销毁对象

本节要谈论的是在对象池中的对象应该如何主动销毁呢？本例中的参考源代码在08_05_packet目录中。销毁组件或实例有两种需求：一种是在本线程中销毁，适合一般的对象；另一种是像Packet类这类跨线程实例。下面我们分别介绍这两种对象的销毁方式。

8.5.1 一般组件销毁

当不再需要一个组件时，有两种可以销毁的方式：

(1) 如果组件是通过实体AddComponent这个途径增加的，那么调用Entity::RemoveComponent函数即可销毁。

(2) 如果是没有实体的组件（例如HttpRequest），就可以直接调用该线程中的EntitySystem::RemoveComponent函数销毁。以HttpRequest为例看一下它是如何销毁自己的，代码实现如下：

```
void HttpRequest::Update() {
    switch (_state) {
        ...
        case HttpResquestState::HRS_Over: {
            ProcessOver();
            _state = HttpResquestState::HRS_NoActive;
            GetSystemManager()->GetEntitySystem()->RemoveComponent(this);
        }
        break;
        case HttpResquestState::HRS_Timeout: {
            ProcessTimeout();
            _state = HttpResquestState::HRS_NoActive;
            GetSystemManager()->GetEntitySystem()->RemoveComponent(this);
        }
        ...
    }
}
```

在HttpRequest组件中，功能正常完成或者超时都会调用EntitySystem::RemoveComponent将自己移除。具体看一下在EntitySystem类中移除组件的代码：

```
void EntitySystem::RemoveComponent(IComponent* pObj) {
    const auto entitySn = pObj->GetSN();
    const auto typeHashCode = pObj->GetTypeHashCode();
    auto iterObj = _objSystems.find(typeHashCode);
    if (iterObj == _objSystems.end()) {
        return;
    }
}
```

```
        ComponentCollections* pCollector = iterObj->second;
        pCollector->Remove(entitySn);
    }
    void ComponentCollections::Remove(uint64 sn) {
        _removeObjs.emplace_back(sn);
    }
}
```

在每个线程中，EntitySystem都拥有所有的对象实体，这些实体是按类型不同而放到ComponentCollections中的。当我们要销毁一个组件时，将该组件从ComponentCollections中移除，重新放回对象池即可。需要说明的是，HttpRequest::Update是在UpdateSystem中被调用的，当HttpRequest被销毁时，它的所有组件也将被删除。也就是说，HttpRequest类的UpdateComponent组件在自己的Update函数中意图销毁自己。

运行代码的时候，我们并没有发现异常，也没有引起冲突，是因为ComponentCollections将需要删除的对象进行了缓冲，放到了删除列表中，在下一帧才会真正删除。

8.5.2 引用计数销毁对象

Packet对象的销毁比较复杂，因为它会穿越多个线程。Packet对象的销毁采用了引用计数销毁的方式。流程图如图8-2所示，展示了整个过程。

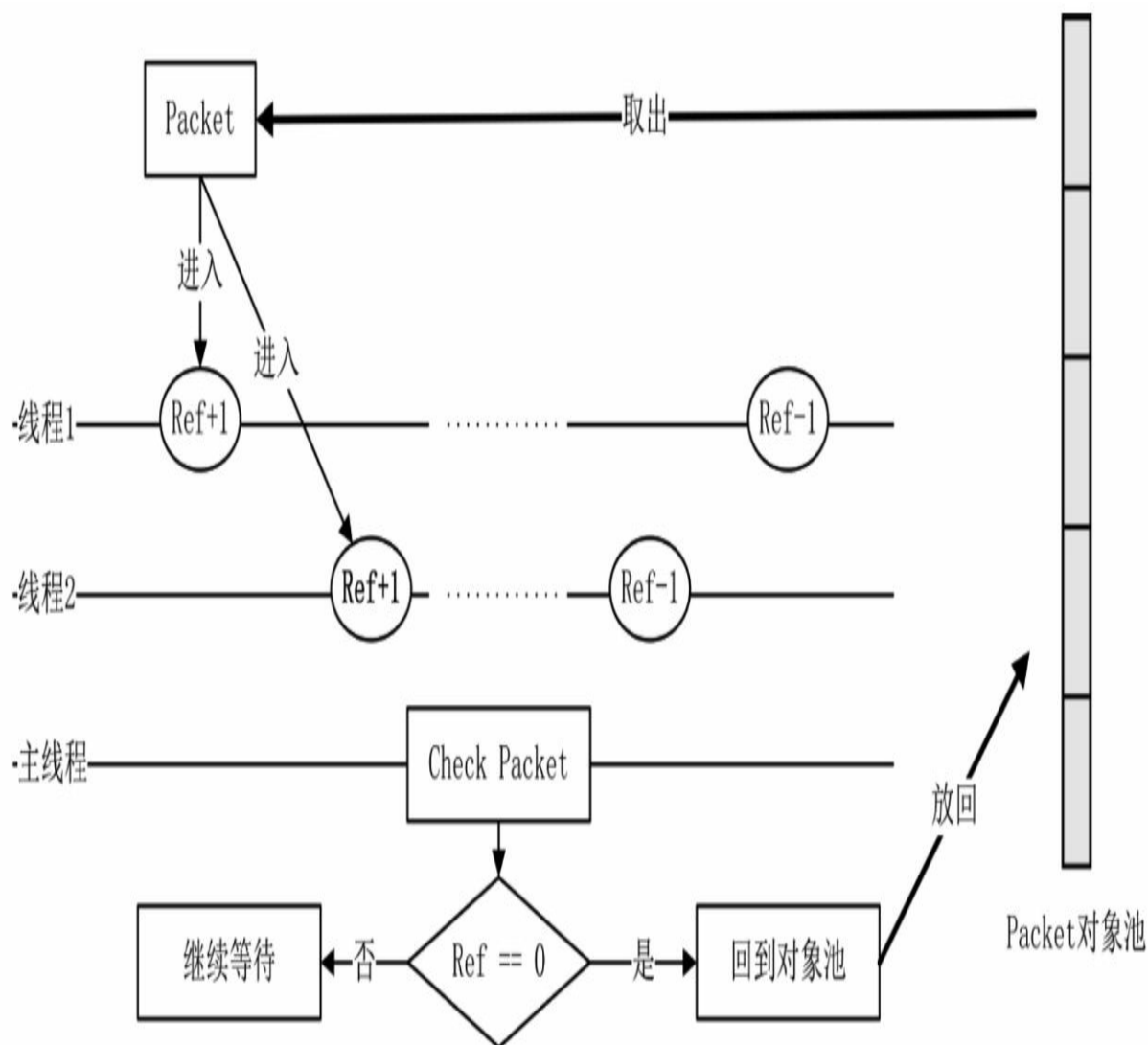


图8-2 Packet从生成到销毁流程

当一个协议到达时，它被封装成了Packet类，这个Packet类的指针将穿越所有线程，查看是否有某个组件对自己感兴趣。当它进入线程时计数加1，退出线程时计数减1。而在主线程中则不断地检查计数，归0

时，该对象将被回收。下面来看看具体的实现，在Packet类中定义了计数，代码如下：

```
class Packet : public Entity<Packet>, public Buffer, public IAwake
public:
    // ref
    void AddRef();
    void RemoveRef();
    void OpenRef();
    bool CanBack2Pool();
    ...
private:
    std::atomic<int> _ref{ 0 };
    bool _isRefOpen{ false };
};
```

对Packet类增加了AddRef、RemoveRef函数，用来进行计数。计数是在多线程中进行的，所以变量采用了std::atomic类型来处理。std::atomic是一个原子操作，底层已经加锁，不需要额外加锁。函数AddRef对引用计数加1，而函数RemoveRef对引用计数减1，当我们使用对象时，它的引用计数加1；当引用计数重新变为0时，表示所有使用已结束，可以销毁了。

对于每一个线程来说必然有一个MessageSystem系统处理协议，Packet进入MessageSystem类处理队列，则计数加1，处理完成之后计数减1。下面来看一下主要代码：

```
void MessageSystem::AddPacketToList(Packet* pPacket) {
    std::lock_guard<std::mutex> guard(_packet_lock);
    _cachePackets.GetWriterCache()->emplace_back(pPacket);
    pPacket->AddRef(); // 进入时 Ref 加1
}

void MessageSystem::Update(EntitySystem* pEntities) {
    ...
    auto lists = pCollections->GetAll();
    auto packetLists = _cachePackets.GetReaderCache();
    for (auto iter = packetLists->begin(); iter != packetLists->end(); iter++) {
        auto pPacket = (*iter);
        Process(pPacket, lists);
        pPacket->RemoveRef(); // 离开时 Ref 减1
    }
}
```

```
    _cachePackets.GetReaderCache()->clear();  
}
```

下面是分发协议的代码，这是Packet进入各线程中的必经函数：

```
void ThreadMgr::UpdateDispatchPacket() {  
    ...  
    auto pList = _packets.GetReaderCache();  
    for (auto iter = pList->begin(); iter != pList->end(); ++iter)  
        auto pPacket = (*iter);  
    ...  
    for (auto iter = _threads.begin(); iter != _threads.end();  
        iter->second->HandlerMessage(pPacket);  
    }  
    pPacket->OpenRef();  
}  
pList->clear();  
}  
void ThreadCollector::HandlerMessage(Packet* pPacket) {  
    auto pList = _threads.GetReaderCache();  
    for (auto iter = pList->begin(); iter != pList->end(); ++iter)  
        iter->second->GetMessageSystem()->AddPacketToList(pPacket)  
    }  
}
```

处理Packet时将Packet通知到各个线程集合，而线程集合再将Packet加入各个线程中。该加入操作完成之后调用了OpenRef函数，这个函数开始了Packet的检查。在Packet生成到加入线程之前，引用计数都是0，显然这时检查计数是不合适的。只有当OpenRef这个开关打开之后，也就是说Packet已经放置到线程中才是检查的时机。

```
void Packet::OpenRef() {  
    _isRefOpen = true;  
}  
bool Packet::CanBack2Pool() {  
    if (!_isRefOpen)  
        return false;  
    if (_ref == 0)  
        return true;  
    return false;  
}
```

如果CanBack2Pool函数返回true，这个Packet就认为被用完了，恢复到对象池中，等下次使用。

以上是Packet类的销毁流程，现在还是以1000个账号来测试内存数据的情况。进入08_05_packet目录，编译后启动allinone进程，再启动robots进程，批量登录1000个账号，登录完成之后，按Ctrl+C组合键退出robots进程，在allinone进程中输入“thread-pool”，这回打印数据如下：

```
- total:1550   free:1550   use:    0   call:8011   6Packet
```

从登录1000个账号到退出一共产生了8000多个协议。当前正在使用的为0个，对象池中有1550个，也就是巅峰状态时，在内存中一共存在1500多个Packet类。使用完成之后，它们都被正常释放了。再使用valgrind工具来检查一下内存：

```
[root@localhost bin]# valgrind --tool=memcheck --leak-check=full .
```

依然是登录1000个账号，登录成功后，按Ctrl+C组合键退出robots进程，让所有对象全部回到对象池。在allinone进程中输入“-exit”或按Ctrl+C组合键来安全退出。如果没有完全释放内存，就会报一些内存错误。在08_05_packet工程中，我们已经修改了内存的释放问题，所以没有内存方面的这类报错。

再来测试一下将Yaml::BackToPool函数内容屏蔽掉，编译运行，退出时就会有几个报错，都是关于Yaml的：

```
==4115== 544 (80 direct, 464 indirect) bytes in 1 blocks are defin
==4115==      at 0x4C2A462: operator new(unsigned long) (vg_replace_
==4115==      by 0x61AA34: Yaml::LoadConfig(APP_TYPE, YAML::Node&) (
```

因为屏蔽掉Yaml::BackToPool函数，在Yaml::LoadConfig创建了新的对象，但是没有销毁它。valgrind工具立刻敏锐地发现了该问

题。在编码时可以随时调用`valgrind`来查看内存数据。

时间堆

至此，整个框架已经可以良好运行，也解决了内存问题，但是还欠缺一个基础功能。考虑这种情况，当调用一个HttpRequest向外请求一个HTTP时，由于某些原因请求没有回应，登录就会一直卡在这里，客户端发送了C2L_AccountCheck协议，但是一直没有得到回应，它只有等待下去。login进程中的Account类也很无奈，HttpRequest没有给它反馈，Account类自然也没有办法给客户端反馈数据。分析一下产生这个问题的原因，是因为没有对HttpRequest定时检查，如果开始时就设置一个10秒期限，在10秒之后还没有反馈，就认为请求失败了，即使后面请求来了，也认为是失败的，这个问题就迎刃而解了。

本节要讨论的是对于服务端来说必不可少的定时器。除了HttpRequest需要定时器外，我们可以看看RobotMgr这个类，为了查看所有robot对象的状态，写了一个定时打印，放在Update函数中：

```
void RobotMgr::Update() {  
    ...  
    auto pGlobal = Global::GetInstance();  
    if (_nextShowInfoTime > pGlobal->TimeTick)  
        return;  
    _nextShowInfoTime = timeutil::AddSeconds(pGlobal->TimeTick, 2)  
    ShowInfo();  
}
```

定时打印采用的逻辑是记录下一次的打印时间，如果时间到来，就调用ShowInfo函数打印信息。同时，在当前时间的基础上增加两秒，设为下一次的打印时间。

我们知道Update函数是每帧调用的，每一帧的循环中都会判断一次是否到了触发时间。假如进程每秒调用100次，也就是说在两秒内有200次if判断。如果在整个框架中存在大量这种时间判断，其实也在消耗性能，而且这段代码看上去也不够优雅。所以，我们需要采用一种新的机制，以最少的判断来执行时间函数调用，也就是下面要讲到的时间堆。要了解时间堆是什么，需要先了解一下二叉树。

以图8-3为例来看一下二叉树结构。所谓二叉树，是一种重要的非线性数据结构，直观地看，它很像自然界中的树，二叉树的一个节点上有两个分支。为什么定时器中使用二叉树呢？假如有3个时间调用的函数，调用时间分别是11点、9点和10点，如图8-4为例，在这个二叉树中，首先建立了11点的节点，当9点的节点插入二叉树时，顶点变成9点。当10点的节点插入到这个二叉树时，其值插入到了9点节点的右侧，顶点依然还是9点。

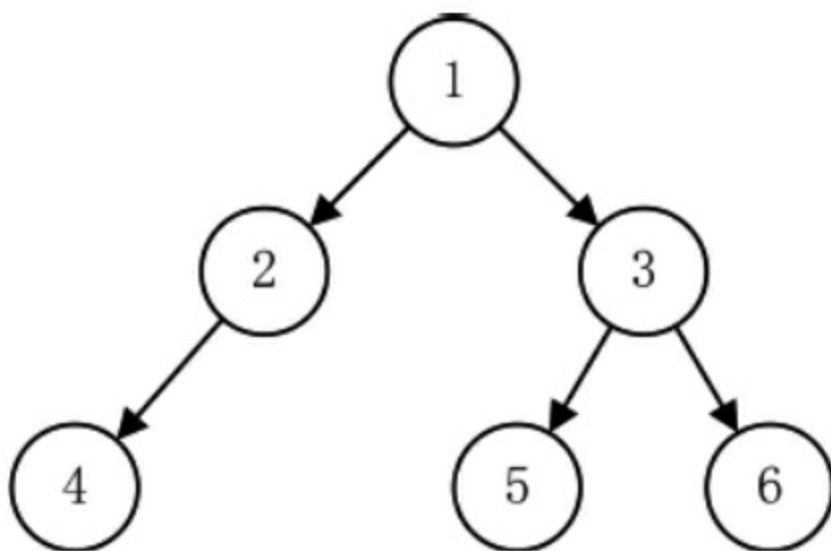


图8-3 二叉树

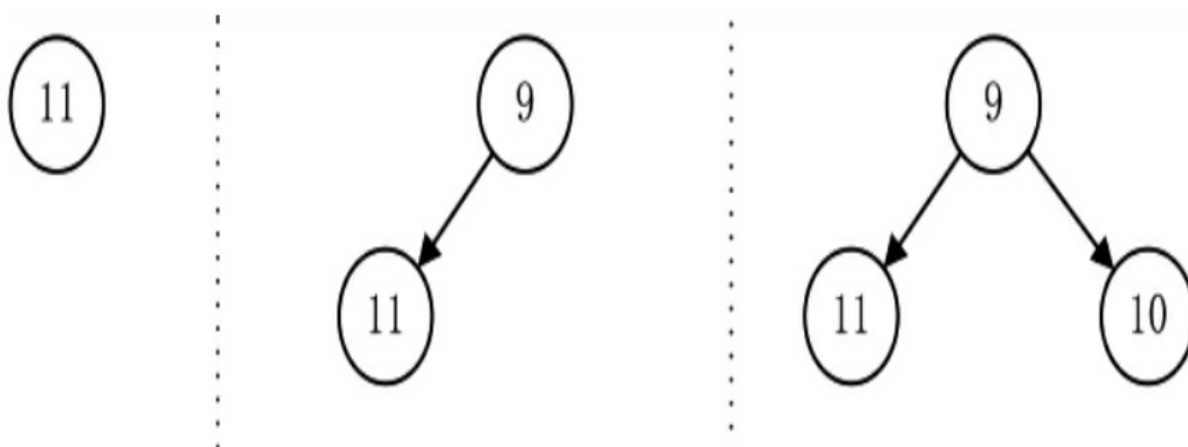


图8-4 二叉树形成

这样一棵树，它的最小顶点一定是最小的时间节点。如果有一个新的需求，需要把10:30的值插入树中，取出顶点来判断需要插入的10:30是否大于当前顶点。如果大于当前顶点，就取出顶点的子节点继续比较，直到找到一个适合自己的位置。如果小于当前顶点，那么它一定是新的顶点。这也是我们制作时间定时器的原理。

讲完了二叉树的逻辑，还需要了解二叉树的存储。二叉树的存储有非常多的方式，常见的有数组存储和链表存储。

二叉树的存储正好满足2的N次方规律，即顶层为1，一层为2，二层为4，三层为8，以此类推。因为每个节点都有两个子节点，所以 $\text{array}[2^{\text{Level}} - 1]$ 即为Level层的开始节点，如图8-5所示。

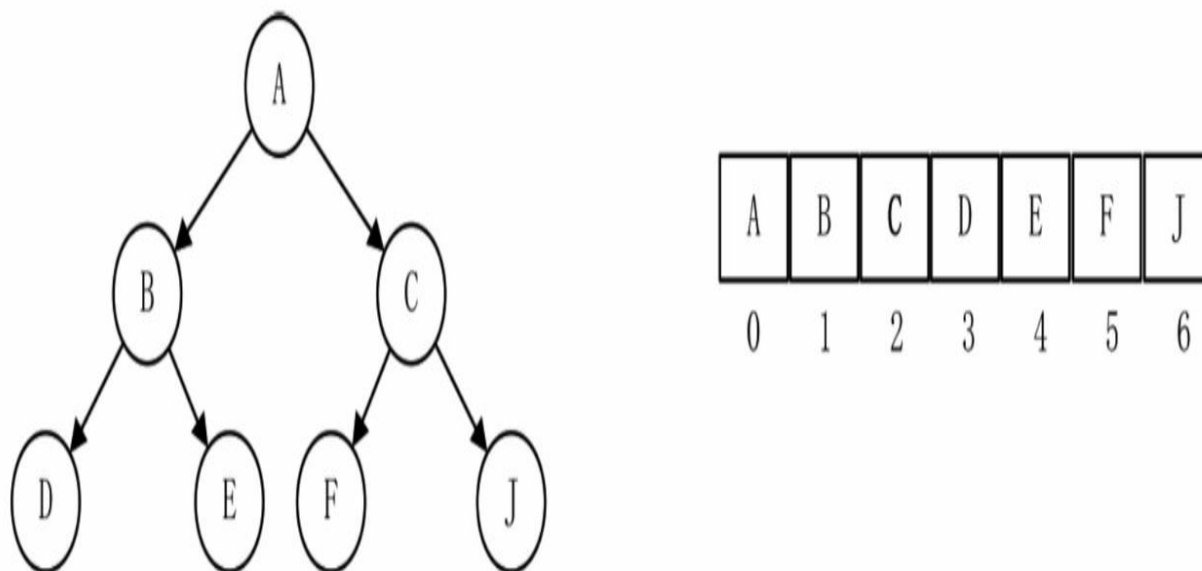


图8-5 二叉树内存存储结构

除了使用数组存储之外，还可以使用链表。链表存储的结构中，每个节点前后各有一个指针指向下一个节点。

数组存储和链表存储各有优劣，数组存储的访问速度更快，可以一步算出节点所在的位置，但是数组存储有空节点存在的情况。当树非常大时，数组存储比链表存储占用的内存更多，因为需要为节点占位。而链表存储虽然速度不及数组方式，却是按需存取节点的，不存在空节点的问题。

我们要制作的定时器就是基于二叉树的原理，名为时间堆。时间堆包括最大时间堆和最小时间堆。将整个二叉树进行排序，顶点是最小时间间隔，为最小时间堆；最下层的叶子节点为最大时间间隔，为最大时间堆。前面我们列举了最小时间堆的例子，顶点是距离现在时间最近的一个时间节点，当节点9被取出来之后，整个时间堆再次进行排序，从剩余的节点中找到最小的一个时间节点，并将这个节点设为顶点。

在一个线程中有这样一个时间堆，那么每一帧只需要有限的判断，判断当前时间与顶点时间的大小，即可得知是否要触发这个定时器的调用。如果最近的一个调用时间都没有到来，那么这个堆中的其他时间节点肯定不会触发。假设有100个定时器，按照之前的代码逻辑，需要编写100个触发判断函数，而现在被压缩在了顶点上，只需要一次判断。

8.6.1 堆实现代码

要实现这样一个二叉树，还要实现时间堆的排序，我们需要编写算法。所幸，在C++标准中已经为我们提供了一个堆操作，std标准库中的堆是基于集合的方式实现的。我们先来看一个堆的例子，源代码位于08_06_heap目录中。本例很简单，读者可以编译执行后查看结果。

```
int main() {
    std::vector<int> data{ 9, 1, 6, 3, 8, 9 };
    Show(data);

    std::cout << "执行make_heap." << std::endl;
    make_heap(data.begin(), data.end(), std::greater<int>());
    Show(data);

    PopData(data);
    PopData(data);

    PushData(data, 5);
    PushData(data, 1);

    PopData(data);
    return 0;
}
```

在main函数中定义了一个vector，随机给了几个值，然后调用make_heap函数让它变成一个堆数据。随后对数据进行了两次PopData弹出操作，两次PushData添加数据的压入操作，分别增加了5和1，最后执行了PopData弹出操作。弹出函数PopData的定义如下：

```
void PopData(std::vector<int>& data) {
    // 弹出heap顶元素，将其放置于区间末尾
    pop_heap(data.begin(), data.end(), std::greater<int>());
    std::cout << "弹出数据: " << data.back() << std::endl;
    data.pop_back();
    Show(data);
}
```

在弹出数据时调用了pop_heap函数，该函数会重新调整整个堆空间，将顶层数据取出来，放在vector数据的最后以便使用。添加数据的PushData函数的定义如下：

```
void PushData(std::vector<int>& data, const int value) {
    std::cout << "往堆中添加元素: " << value << std::endl;
    data.push_back(value);
    push_heap(data.begin(), data.end(), std::greater<int>());
    Show(data);
}
```

在堆数据中增加了新节点后，调用push_heap函数对整个heap数据进行数据整理排序。执行后得到如表8-2所示的结果。

表8-2 时间堆结果列表

代码调用序列	结 果
data{ 9, 1, 6, 3, 8, 9 };	原始数据: 9 1 6 3 8 9, 顶端: 9
调用了 make_heap 之后	调整后的数据列表: 1 3 6 9 8 9, 顶端: 1
PopData(data);	弹出数据 1 之后的数据列表: 3 8 6 9 9, 顶端: 3
PopData(data);	弹出数据 3 之后的数据列表: 6 8 9 9, 顶端: 6
PushData(data, 5);	添加元素 5 之后的数据列表: 5 6 9 9 8, 顶端: 5
PushData(data, 1);	添加元素 1 之后的数据列表: 1 6 5 9 8 9, 顶端: 1
PopData(data);	弹出数据 1 之后的数据列表: 5 6 9 9 8, 顶端: 5

在本例中主要使用了3个重要的函数：

- (1) make_heap函数的作用是重新排列给定范围内的元素，使它们

形成堆。

(2) `pop_heap`函数的作用是弹出堆顶元素，将堆顶元素移动到集合的最后，并重新排列剩下的元素。值得注意的是，之前的顶元素被放在了最后，并从堆数据中删除，相当于堆数据减1。

(3) `push_heap`函数的作用是对最后一个元素进行插入，插入堆中的适当位置。也就是说，在调用`push_heap`之前，插入的新元素一定是在数据的最尾端。

在std标准库中，对于堆有两种已实现的计算，默认是最大堆`less<T>`和最小堆`greater<T>`，当然也可以自己定义排序算法，std支持自定义算法的定义。也就是说，我们可以对堆中的数据给出一个自定义的排序算法。

准备工作已经完成了，如果我们的时间调用函数采用时间堆的方式，就需要在这个堆中注册许多待执行的函数，而每一次只需要访问一下堆顶的函数是否到了执行时间，这样就节省了判断时间，代码又不会过于烦琐。下面通过std这个堆算法来实现定时器功能。

8.6.2 时间堆组件

本小节要使用最小堆的概念写一个定时器，源代码位于08_07_timer目录。在文件timer_component.h中先看一下定时器的定义：

```
using TimerHandleFunction = std::function<void(void)>;
struct Timer {
    timeutil::Time NextTime;           // 下次调用时间
    TimerHandleFunction Handler;       // 调用函数
    int DelaySecond;                   // 首次执行时延迟秒
    int DurationSecond;                // 间隔时间(秒)
    int CallCountTotal;                 // 总调用次数(0为无限)
    int CallCountCur;                  // 当前调用次数
    uint64 SN;                          // 方便删除数据时找到
};
```

在8.6.1小节对堆进行测试时，测试代码中只保存了一个int，但是实际使用时，在时间堆中每个节点都是Timer结构，这种结构包括总调用次数、调用时间间隔、当前调用次数、调用函数和下次调用时间等。如果总调用次数为0，这个节点就是一个无限次数的循环定时器。属性DelaySecond是首次执行时延时，这个属性和时间间隔有一定的区别。打个比方，HTTP请求的超时检查不需要马上执行，所以需要设置首次执行时的延时。但在RobotMgr类输出信息时，定时器需要马上执行。

除了定义定时器的结构外，还需要有一个定时器的组件TimerComponent，定义如下：

```
class TimerComponent :public Entity<TimerComponent>, public IAwake
public:
    void Awake() override;
    uint64 Add(int total, int durations, bool immediateDo, int imm
    void Remove(std::list<uint64>& timers);
    bool CheckTime();
    Timer PopTimeHeap();
    void Update();
    ...
private:
    std::vector<Timer> _heap;
```

```
};
```

从组件TimerComponent的定义中可以看出，它是一个单例，对于一个线程来说，只需要有一个TimerComponent组件。在初始化时为该组件增加UpdateComponent组件，以方便每一帧检查是否有到了时间的函数需要执行。TimerComponent组件的初始化与更新函数如下：

```
void TimerComponent::Awake() {
    auto pUpdateComponent = AddComponent<UpdateComponent>();
    pUpdateComponent->UpdateFunction = BindFunP0(this, &TimerCompo
}
void TimerComponent::Update() {
    while (CheckTime()) {
        Timer data = PopTimeHeap();
        data.Handler();
        if (data.CallCountTotal != 0)
            data.CallCountCur++;
        if (data.CallCountTotal != 0 && data.CallCountCur >= data.
            //delete pNode; 取出之后，不再加入堆中
        } else {
            // 重新加入堆中
            data.NextTime = timeutil::AddSeconds(Global::GetInstan
            Add(data);
        }
    }
}
```

在更新函数中，首先检查是否有需要执行的节点，判断函数非常简洁：

```
bool TimerComponent::CheckTime() {
    if (_heap.empty())
        return false;
    const auto data = _heap.front();
    return data.NextTime <= Global::GetInstance()->TimeTick;
}
```

对比一下节点的时间与当前时间，如果节点的时间大于当前时间，就说明还没有到调用时间，如果节点的时间小于或等于当前时间，就说

明定时器时间已过，需要马上执行。调用PopTimeHeap函数将顶点弹出堆，执行后再次判断它是否需要进入堆中再次执行。值得注意的是，在调用CheckTime函数时用了while，原因就是时间堆中可能存在多个满足条件的节点。

在每个线程中都有一个TimerComponent组件实例，每个EntitySystem中都有一个独立的TimerComponent组件负责该进程所有需要按时间调用的事件。在线程启动时，该组件就被创建出来了。

每个组件都有可能需要定时器，为了方便组件调用定时器，修改了IComponent组件，增加了一个关于定时器的函数。

```
class IComponent : virtual public SnObject {
protected:
    void AddTimer(const int total, const int durations, const bool
        std::list<uint64> _timers;
    ...
};
void IComponent::AddTimer(const int total, const int durations, co
    auto obj = GetSystemManager()->GetEntitySystem()->GetComponent
    const auto timer = obj->Add(total, durations, immediateDo, imm
        _timers.push_back(timer);
}
```

传入的参数依次为最大调用次数、调用间隔、是否马上执行、马上执行时间间隔。将一个定时器加入TimerComponent组件之后，在Component中留下了它的唯一标识码SN，这是为了删除使用的。每个组件都可以创建自己的定时器，而且可以创建多个，这些定时器的标识被存放在组件的times列表中。当组件被销毁时，这些定时器也会被销毁。

```
void IComponent::ComponentBackToPool() {
    BackToPool();
    if (!_timers.empty()) {
        auto pTimer = _pSystemManager->GetEntitySystem()->GetCompo
        if (pTimer != nullptr)
            pTimer->Remove(_timers);
        _timers.clear();
    }
    ...
}
```

```
}
```

下面来看时定器的应用。以RobotMgr为例，RobotMgr类在每一段时间需要显示一下当前登录的机器人的状态，现在使用定时器后，它的代码如下：

```
void RobotMgr::Awake() {  
    ...  
    AddTimer(0, 2, false, 0, BindFunP0(this, &RobotMgr::ShowInfo))  
}
```

在旧的代码中，需要维护一个变量记录下次执行的时间点，现在不用了。现在这些数据都在TimerComponent组件中，不需要做额外的工作。在初始化RobotMgr时调用定时器注册函数AddTimer即可。AddTimer函数最终调用的是TimerComponent::Add，该函数的实现如下：

```
uint64 TimerComponent::Add(const int total, const int durations, c  
    Timer data;  
    data.SN = Global::GetInstance()->GenerateSN();  
    data.CallCountCur = 0;  
    data.CallCountTotal = total;  
    data.DurationSecond = durations;  
    data.Handler = std::move(handler);  
    data.NextTime = timeutil::AddSeconds(Global::GetInstance()->Ti  
    if (immediateDo) {  
        data.NextTime = timeutil::AddSeconds(Global::GetInstance()  
    }  
    Add(data);  
    return data.SN;  
}  
void TimerComponent::Add(Timer& data) {  
    _heap.emplace_back(data);  
    if (_heap.size() == 1) {  
        make_heap(_heap.begin(), _heap.end(), CompareTimer());  
    } else {  
        push_heap(_heap.begin(), _heap.end(), CompareTimer());  
    }  
}
```

从RobotMgr初始化代码中不难看出在RobotMgr中注册的这个定时器是一个无限执行的函数，它的间隔时间为2秒，不需要立刻执行。当这个定时器被加入组件TimerComponent中的std::vector<Timer>之后，调用了标准库中的push_heap函数进行堆计算，让这个时间堆中最近一个当前时间点的节点位于这个堆的最顶端，方便TimerComponent组件在下一帧时调用CheckTime函数时使用。

将Timer结构放入或取出堆时，自定义了一个排序类CompareTimer。

```
struct CompareTimer {
    constexpr bool operator()(const Timer& _Left, const Timer& _Right) {
        return (_Left.NextTime > _Right.NextTime);
    }
};
```

对每个节点的下一次调用时间做了对比。

在08_07_timer目录的工程中，对HttpRequestAccount类的超时使用了定时器，这里不再进行代码讲解，原理与前面一致。

本节引入了定时器，组件中只需要调用一个AddTimer函数就可以生成一个定时器。除了时间堆之外，还有一种定时器的模式叫时间轮。时间轮比时间堆效率更高，因为插入和取出时，时间轮都没有时间成本，但时间轮的数据结构更为复杂。本书不再对时间轮进行讲述，如果读者感兴趣，可以上网搜索该算法。

总结

本章完成了ECS框架中关于系统部分的改造，已经彻底从继承类的编程思路中跳转出来，变成了面向组件编程。本章是对基础框架的一个补充，在第7章的基础框架之上增加了线程的类型，不同的线程处理Packet有不同的策略。本章还优化了对象池，方便进一步监控内存对象的生命周期。同时引入了时间堆的概念，并加入了定时器功能。定时器是一个非常基础的功能，在逻辑代码中会经常使用。

至此，这个分布式系统搭建的框架部分基本完成了，后面的章节将重点放在上层应用上。

第9章

服务器管理进程与HTTP

在最初的构想中，我们的框架有多个login进程，它是服务器的第一道屏障，玩家只有登录成功才可能进入游戏。当有多个login的时候就涉及一个问题，客户端到底该从哪个login登录呢？这是本章要解决的问题。本章包括以下内容：

- 提出一个多进程登录方案。
- HTTP分析。
- 新的网络标识。

启动多个login进程

到目前为止，框架还只有一个login进程，单login极易形成瓶颈，所以本节需要为整个架构设置多个login进程。要架设多个login，首先需要进行多个login的配置，打开engine.yaml文件，重新配置如下：

```
login:
  thread_num: 2
  url_login: HTTP://192.168.1.120/member_login_t.php
  url_method: "get"
  apps:
    - id: 101
      ip: 127.0.0.1
      port: 5401
    - id: 102
      ip: 127.0.0.1
      port: 5402
```

在login属性之下配置了一个apps，这是一个列表类，YAML语法用到了“-”。现在配置了两个login信息，其中属性id（libserver库中ServerApp类中的AppId属性）是为了区分到底启动的是哪一个login。现在，启动login时需要带上参数，例如“./login.exe-sid=101”是启动AppId为101的login。

本节的源代码位于目录09_01_appmgr中，在ServerApp初始化时，ServerApp::Initialize函数对命令行参数进行了分析。分析出来的AppId保存到全局的Global类中，以方便读取。现在有了多个login，那么问题来了，对于客户端来说，登录游戏时应该使用哪一个login呢？

解决这个问题有两种方案：一种方案是从运维的角度来解决，利用

Nginx的功能，这种方式在后面讨论；另一种方案是通过游戏的逻辑数据来解决，做一个简单的登录策略，每个玩家登录时，可以通过某种方式得知玩家在线最少的login的IP和端口。

现在登录问题进一步演变为该如何得到最小负载的login。我们向哪个服务器发送这个请求可以得到答案呢？为了解决这个问题，需要新建一个工程，用来维护所有的login，将其称为appmgr。除了当前的login进程之外，appmgr将来还会维护game进程。

appmgr 进程

在之前的例子中，robots进程登录服务器时是通过一个具体的IP和端口来进行网络连接的，但是在新的流程中，因为有无数个login进程，所以并没有一个具体的IP和端口，而且随着游戏的进展，这些IP和端口还可能动态地增减。在此基础上，需要有一个appmgr进程，如图9-1所示，这个进程的作用是动态收集login的信息，再以某种方式通知客户端。

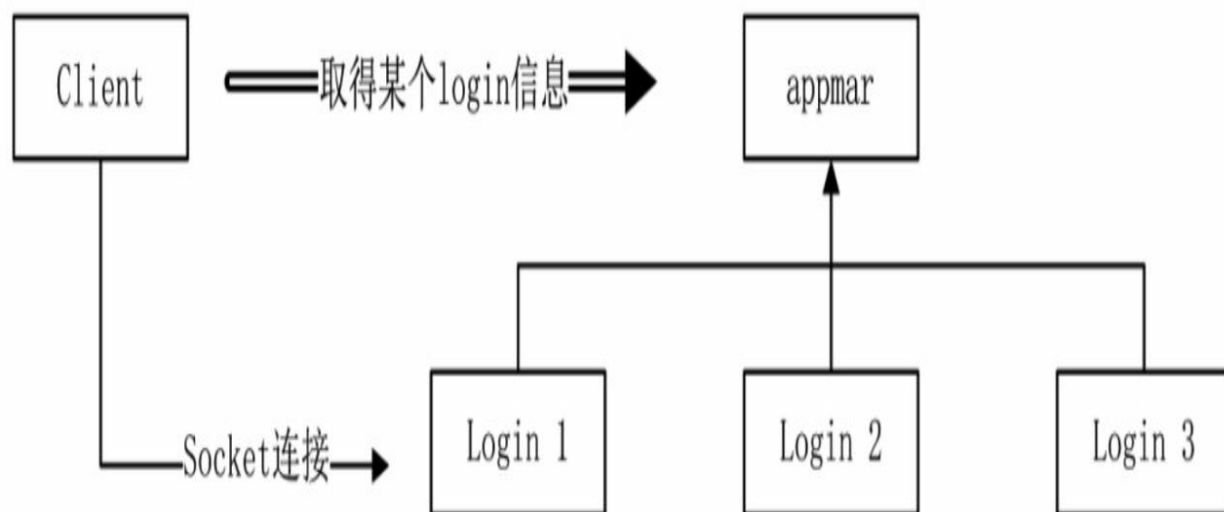


图9-1 appmgr登录数据

login进程启动之后会定期向appmgr发送自己当前的状态，这样appmgr就被动地收集到了所有login的数据。而客户端登录时，只需要向appmgr询问即可得到最小负载的login信息。

本节源代码位于09_01_appmgr目录中，打开appmgr工程的login_sync_component.h文件，其中定义了收集组件，名为LoginSyncComponent。该组件的主要作用是收集所有login进程的同步信息，其类定义和基类定义如下：

```
class LoginSyncComponent :public SyncComponent, public IAwakeSystem
public:
    void Awake() override;
    void BackToPool() override;
};
class SyncComponent :public Entity<SyncComponent> {
public:
    void AppInfoSyncHandle(Packet* pPacket);
    bool GetOneApp(APP_TYPE appType, AppInfo& info);
    ...
protected:
    std::map<int, AppInfo> _apps;
};
```

基类组件SyncComponent将收集到的数据保存在字典std::map<int, AppInfo>中，Key值为AppId，Value值为结构AppInfo，AppInfo的定义如下：

```
struct AppInfo {
    APP_TYPE AppType;
    int AppId;
    std::string Ip;
    int Port;
    int Online;
    SOCKET Socket;
};
```

目前功能比较简单，这个结构也非常简单，AppInfo结构中只有一个Online属性，用于记录在这个AppId对应的进程中有多少玩家在线。

LoginSyncComponent::Awake初始化函数中注册了login同步消息MI_AppInfoSync的处理函数。对于LoginSyncComponent组件来说，它并不关心谁发送了数据，只关心MI_AppInfoSync协议本身的内容，谁发送来的并不重要。当LoginSyncComponent收到MI_AppInfoSync协议时，交

给它的基类处理。处理该协议的函数实现如下：

```
void SyncComponent::AppInfoSyncHandle(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::AppInfoSync>();
    const auto iter = _apps.find(proto.app_id());
    if (iter == _apps.end()) {
        AppInfo syncAppInfo;
        syncAppInfo.Parse(proto);
        syncAppInfo.Socket = pPacket->GetSocket();
        _apps[syncAppInfo.AppId] = syncAppInfo;
    } else {
        const int appId = proto.app_id();
        _apps[appId].Online = proto.online();
        _apps[appId].Socket = pPacket->GetSocket();
    }
}
```

该协议中会收集AppType、AppId以及当前的在线人数，LoginSyncComponent将这些数据缓存起来，以备需要时使用。

MI_AppInfoSync协议是由login进程发出的，在login工程的Account类中。Account类增加了一个定时器，每隔几秒就将自己的信息发送出去。

```
void Account::Awake() {
    AddTimer(0, 10, true, 2, BindFunP0(this, &Account::SyncAppInfo
    ...
}
void Account::SyncAppInfoToAppMgr() {
    Proto::AppInfoSync protoSync;
    protoSync.set_app_id(Global::GetInstance()->GetCurAppId());
    protoSync.set_app_type((int)Global::GetInstance()->GetCurAppTy
    protoSync.set_online(_playerMgr.Count());
    MessageSystemHelp::SendPacket(Proto::MsgId::MI_AppInfoSync, pr
}
```

在这个定时器中，每10秒调用一次SyncAppInfoToAppMgr函数。每隔10秒，Account类根据当前玩家的数据向appmgr进程发送在线情况。协议数据最终被送到了LoginSyncComponent组件中。

打开09_01_appmgr目录，编译后运行allinone工程，来看看收到了些什么数据。在工程中新写了一个app指令，利用app-info查看当前进程中app的情况。

```
app -info
[DEBUG] -  appId:    0 type:allinone online:0
```

当我们使用robots进程登录一个玩家之后，在allinone终端输入“app-info”，情况发生了一些变化。

```
[DEBUG] -  appId:    0 type:allinone online:1
```

可以拆分进程再测试一次。启动dbmgr、appmgr和两个login进程，用robots登录一个玩家到其中一个进程上。注意启动login时需要参数“./login-sid=101”。在appmgr终端输入“app-info”可以看到如下信息：

```
[DEBUG] -  appId: 101 type:login online:1
[DEBUG] -  appId: 102 type:login online:0
```

启动了两个login，登录一个账号之后，AppId为101的login进程上就会有一个玩家在线。玩家进入id为101的login之后，数据会被通知到appmgr进程上。

现在有了数据，但还是没有解决多服务器登录的问题，因为robots总是连接到AppId为101的进程上。

前面完成了数据的收集工作，接下来要解决的问题是：客户端通过什么方式从appmgr中知道login的这些状态呢？鉴于客户端与appmgr进程只有一次性的通信，这里我们采用HTTP弱连接方式。具体方法是，appmgr进程提供一个HTTP端口——例如appmgr的IP为192.168.0.100，开放一个HTTP服务——端口为8081，提供一个192.168.0.100:8081/login请求，该请求返回一个JSON数据，类似{"ip": "192.168.0.100", "port": 5002}，让调用者知道当前在

192.168.0.100的5002端口上有一个可以进行连接的login进程。

总之，在登录时，robots进程开始并不知道它能够连接哪一个login，它会先向appmgr发起一个HTTP请求，appmgr根据当前所有login的动态信息给它一个合适的用于登录的IP和端口，robots进程收到之后再再进行网络连接。

鉴于此，现在需要在框架中加入HTTP的处理，提供HTTP服务。

这样做有两个好处：第一，HTTP是短连接，用完就释放，比TCP快，同时HTTP本身有自己的协议体，不需要再创建协议结构；第二，除了用在登录上外，提供HTTP接口还可以用于查看现有服务器的状态。之前我们提供了一些控制台指令来查看服务器的当前状态，但在真正运营的时候不能每一次都用指令来查看当前数据，所以必须有一种方式可以将这些数据提供给外部的第三方显示，这里第三方工具可能是一个后台的Web工具，使用HTTP就可以方便地展示在网页上，而不需要登录后台输入指令。

要提供HTTP服务，就不得不从HTTP开始分析。这个协议是我们日常使用的，其实非常简单。

HTTP

首先需要了解HTTP是怎么回事。我们每天的生活都离不开HTTP，如网上浏览、网上购物都在使用HTTP。不要觉得这个协议很神秘，以我们现在的知识储备量完全可以解决它。在第1章介绍过网络协议，那么HTTP和我们自定义的协议有什么区别呢？游戏逻辑中的协议是由我们自己定义格式的，而HTTP是一种全球都需要遵循的规范。

本书中自定义协议的格式是由一个长度字段加上PacketHead结构和一个二进制流三部分组成的。HTTP的格式大同小异。它没有长度字段，由协议头加上协议体所组成。HTTP的头部长度的比我们自定义的头部结构更长。在游戏逻辑中，协议都是采用二进制的方式传输的，但HTTP的内容采用的是文本格式。下面用实例来说明一个常规的HTTP请求协议的头部结构。步骤如下：

(1) 打开09_02_http目录，修改log4_help.h文件，打开LOG_HTTP_OPEN宏。

(2) 使用根目录下的make-all.sh脚本编译工程。

(3) 进入bin目录，执行“./allinoned”。

(4) 新开一个终端，使用curl命令查看<http://127.0.0.1:7071/login>请求。其结果如下：

```
[root@localhost ~]# curl http://127.0.0.1:7071/login
{
  "ip" : "127.0.0.1",
  "port" : 5401,
  "returncode" : 0
}
```

```
}
```

在本例中，我们打开了一个新的端口7071，用来接收HTTP相关的请求。在另一个终端使用curl命令向服务端发起一个请求。服务端收到请求后，打印出来的头部数据如下：

```
GET /login http/1.1
User-Agent: curl/7.29.0
Host: 127.0.0.1:7071
Accept: */*
(空行\r\n)
```

众所周知，HTTP是一个文本格式的协议，上面的打印结果就是证明。不需要任何转换，只需要把收到的数据打印出来，我们就可以看到HTTP的数据结构。它就是一个文本。如果用浏览器访问7071端口上的login请求，头部数据的打印输出就不一样了：

```
GET /login http/1.1
Host: 127.0.0.1:7071
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:69.0) Gecko/20100101 Firefox/69.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
(空行\r\n)
```

多了很多头部属性数据，但是总的来说，其格式并没有改变。HTTP的格式如下：

第一行：1.请求方式(GET) 2.空格 3.方法名(/login) 4.空格 5.HTTP版本 6.回车
第二行：1.头部属性名 2.冒号 3.头部属性值 4.回车符\r\n
...
第N行：1.头部属性名 2.冒号 3.头部属性值 4.回车符\r\n
头结束：回车符\r\n

从第二行开始，可以叠加多个属性，头部的结束行必须是回车符。也就是说，在分析协议时，发现有连续两个\r\n，头部数据就结束了。

常用的HTTP请求方式有两种：一种是GET，另一种是POST。当我们在浏览器中输入网页地址或单击一个链接的时候，一般使用的是GET方法，旨在从服务器获取一些数据。而当我们登录或者填写表单的时候，一般使用的是POST方法，旨在将这些数据推送到服务器上。

读者一定很好奇，请求被发送到了HTTP端口上，那么浏览器是如何显示数据的呢？

在框架中，当我们向服务端发送了验证账号协议之后，服务器会返回给客户端一个验证账号结果协议。HTTP请求是一样的道理，服务端收到HTTP请求之后，向连接的Socket（套接字）写入了数据，这些写入的数据被显示到了浏览器上。

这里就涉及HTTP返回数据协议格式，在服务端可以看到输出信息：

```
http/1.1 200 OK
Connection: close
Content-Type: application/json; charset=utf-8
Content-Length:58
（空行\r\n）
{
    "ip" : "127.0.0.1",
    "port" : 5401,
    "returncode" : 0
}
```

HTTP返回协议的头部格式如下：

第一行：1.HTTP版本 2.空格 3.状态码 4.状态码描述 5.回车符\r\n
第二行：1.头部属性名 2.冒号 3.头部属性值 4.回车符\r\n
...
第N行：1.头部属性名 2.冒号 3.头部属性值 4.回车符\r\n
头结束：回车符\r\n

返回协议的第一行与请求协议略有不同，接下来几行和请求协议一

样，都是属性名：属性值的结构，直到两个\r\n结束。如果将头部数据打印出来，那么这两个\r\n表现为一个空行。随后进入协议体。协议体就是真正在浏览器中显示的数据。

在浏览器中输入“<http://192.168.0.120:7071/login>”，该请求在浏览器上显示为：

```
{ "ip" : "192.168.0.120", "port" : 5401, "returncode" : 0 }
```

从这个显示可见，真正的数据部分也就是除开头之外的部分。在返回协议中，属性Connection: close表示主动要求断开连接。Content-Length的属性值得我们注意，它的值就是整个协议体的长度。

在某些情况下，Content-Length属性不存在，这时的返回数据可能是按块传递的。当返回数据非常大，不便于计算长度时，就可以按块来传递数据。假设有100KB的网页数据，这100KB的数据可能是几次传递到浏览器上的，但头部数据只传递一次，后面收到的数据格式为：长度+协议体。直到收到两个\r\n结束协议体，表示整个数据接收完成。

简而言之，按块传递时，发送数据是分成几次发送的，但是头部数据只有一次。后面我们会详细讲解这一部分的内容。

在返回协议中，另一个属性Content-Type表示这个协议体中的格式是什么，它的值非常多，如果填上text/plain就表示纯文本，text/html是常规网页的显示数据，表示协议体中是一个HTML格式的文本，本例中返回的是application/json，表示内容是JSON格式的数据。简单了解了HTTP的格式后，下面在框架中实现一个HTTP服务。

使用Mongoose分析HTTP格式

由于已经有非常多HTTP的第三方开源工程，本书不会再重新编写一个库。框架中采用的第三方库为Mongoose，其源代码地址为<https://github.com/cesanta/mongoose>。该库非常简单，但如果全部采用Mongoose库的逻辑，与框架逻辑结合起来就会比较混乱，所以没有完整地使用这个库，只使用了两个文件：mongoose.h和mongoose.c。这两个文件的作用是分析HTTP请求协议的格式。

将这两个文件直接引入libserver中，框架并不使用Mongoose服务器，只是根据现有的框架，使用它的函数来减少HTTP的分析工作，毕竟Mongoose库就是用于HTTP服务的，在分析方面肯定做得比我们更好，考虑得更全面。

本节和后面关于HTTP的章节，示例都在09_02_http目录中。既然要开放HTTP服务，必然需要一个监听类，依然还是使用NetworkListen类。为了区分游戏逻辑层和HTTP监听的不同，在network_type.h文件中新定义了一个枚举：

```
enum class NetworkType {  
    None = 0,  
    TcpListen = 1 << 0,  
    TcpConnector = 1 << 1,  
    HttpListen = 1 << 2,  
    HttpConnector = 1 << 3,  
};
```

下面是NetworkListen类的定义：

```
class NetworkListen : public IAwakeSystem<std::string, int>, public  
public:  
    void Awake(std::string ip, int port) override;  
    void Awake(int appType, int appId) override;  
    ...  
}
```

现在，NetworkListen类有两种创建方式：当以<int, int>参数创建时，表示输入两个int类型的数据，分析的是AppType与AppId，创建出来的NetworkListen实例是TCP监听，即游戏逻辑需要使用的监听。

当以<string, int>参数创建时，表示输入的是IP与端口，创建出来的NetworkListen实例用于HTTP的监听。下面来详细分析NetworkListen如何实现对HTTP的处理。

9.4.1 HTTP类型

不论是游戏的端口监听还是HTTP的端口监听，要接收数据都需要创建一个Socket，其调用代码如下：

```
SOCKET Network::CreateSocket() const {
    _sock_init();
    SOCKET socket;
    if (_networkType == NetworkType::HttpListen || _networkType ==
        socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
    else
        socket = ::socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    ...
    return socket;
}
```

在创建Socket的时候，TCP监听使用了：`socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)`，注意最后一个参数`IPPROTO_TCP`。也就是说，要用TCP的方式来创建一个Socket。HTTP监听时使用的是`IPPROTO_IP`，是一种很简单的传输方式，也是一种无状态的传输方式。

使用`IPPROTO_TCP`时，TCP底层有一套机制保证传递到网络上的数据是有序的且不遗漏，但`IPPROTO_IP`不是这样，`IPPROTO_IP`没有TCP的3次握手，但这种方式更快。

假设在这里HTTP监听或HTTP连接也采用`IPPROTO_TCP`，可不可以？当然可以。HTTP协议体与它的传输方式没有任何关系，完全可以使用`IPPROTO_TCP`来传输它，只是作为一个一次性连接没有必要而已。HTTP在网络传输中丢掉就丢掉，丢掉超时可以再发，所以使用更为简单的`IPPROTO_IP`。

如果读者曾经看过一些Web开发相关的图书，会提到Session、Cookie之类的内容，这并不是HTTP底层特有的，而是我们使用的框架逻辑附加给它的。本书中的示例并没有Session和Cookie的概念，我们实现的是比较简单的HTTP服务，而Session和Cookie是业界约定俗成的，导致大部分源代码都以Session作为Web连接的处理类。

9.4.2 接收HTTP数据

创建好监听之后，当收到HTTP请求时，首先会在网络层产生一个Socket描述符，这和TCP监听没有什么两样。当底层有数据需要读取，封装为Packet时，分析数据写了一个GetHttpPacket的分支：

```
Packet* RecvNetworkBuffer::GetPacket() {
    auto pNetwork = _pConnectObj->GetParent<Network>();
    auto iType = pNetwork->GetNetworkType();
    if (iType == NetworkType::HttpConnector || iType == NetworkType::
        return GetHttpPacket();

    return GetTcpPacket();
}
```

GetTcpPacket函数用于TCP监听模式下，取出来的数据是自定义的协议。函数GetHttpPacket则是对于HTTP的分析，将网络层数据转换成内存中使用的Packet类。下面来看看具体实现：

```
Packet* RecvNetworkBuffer::GetHttpPacket() {
    if (_endIndex < _beginIndex) {
        // 有异常，关闭网络
        _pConnectObj->Close();
        LOG_ERROR("http recv invalid.");
        return nullptr;
    }
    const unsigned int recvBufLength = _endIndex - _beginIndex;
    const auto pNetwork = _pConnectObj->GetParent<Network>();
    const auto iType = pNetwork->GetNetworkType();
    const bool isConnector = iType == NetworkType::HttpConnector;

    // 分析出头部的数据
    http_message hm;
    const unsigned int headerLen = mg_parse_http(_buffer + _beginIndex,
    if (headerLen <= 0)
        return nullptr;

    unsigned int bodyLen = 0;
    const auto mgBody = mg_get_http_header(&hm, "Content-Length");
    if (mgBody != nullptr) {
```

```

        bodyLen = atoi(mgBody->p);
        // 整个包的长度不够，再等一等
        if (bodyLen > 0 && (recvBufLength < (bodyLen + headerLen))
            return nullptr;
    }

    bool isChunked = false;
    const auto mgTransferEncoding = mg_get_http_header(&hm, "Trans
    if (mgTransferEncoding != nullptr && mg_vcasecmp(mgTransferEnc
        isChunked = true;

    // 后面的数据还没有到达
    if (recvBufLength == headerLen)
        return nullptr;

    bodyLen = mg_http_get_request_len(_buffer + _beginIndex +
    if (bodyLen <= 0)
        return nullptr;

    bodyLen = _endIndex - _beginIndex - headerLen;
}
// 数据接收完毕之后创建一个Packet类
Packet* pPacket = MessageSystemHelp::ParseHttp(_pConnectObj, _b

// 从网络缓存中删除已读取的数据
RemoveData(bodyLen + headerLen);
return pPacket;
}

```

上面的代码看上去比较枯燥，涉及缓存中字节的读取，使用的是第三方库函数，分别说明如下：

- (1) `mg_parse_http`函数，从一个buffer中提出HTTP的头部信息，结构为`http_message`。
- (2) `mg_get_http_header`函数，从头部信息中取得某个属性的值。
- (3) `mg_http_get_request_len`函数，从传入的字符串中找到两个`\r\n`，返回长度。

取得HTTP头部信息之后，如果有长度属性`Content-Length`，就按长度读出数据；如果没有长度属性，就需要检查数据是不是按块发送的。

按块发送时，头部属性中可以读出Transfer-Encoding属性。如果该属性值为chunked，就表示协议体是按块发送的，需要收集所有块数据，再将这些数据组织到Packet中，并发送给其他组件。当数据接收完成之后，调用MessageSystemHelp::ParseHttp函数生成供框架使用的Packet类。下面是它的实现：

```
Packet* MessageSystemHelp::ParseHttp(NetworkIdentify* pIdentify, c
Proto::HTTP proto;
if (bodyLen > 0) {
    if (isChunked) {
        const auto end = s + bodyLen;
        mg_str tmp;
        while (true) {
            const int len = std::atoi(s);
            if (len == 0)
                break;
            s = mg_skip(s, end, "\r\n", &tmp);
            s = mg_skip(s, end, "\r\n", &tmp);
            proto.mutable_body()->append(tmp.p, tmp.len);
        }
    } else {
        proto.mutable_body()->append(s, bodyLen);
    }
}
Proto::MsgId msgId = Proto::MsgId::MI_HttpRequestBad;
if (hm->method.len > 0) {
    // 请求
    do {
        if (mg_vcasecmp(&hm->method, "GET") == 0) {
            if (mg_vcasecmp(&hm->uri, "/login") == 0) {
                msgId = Proto::MsgId::MI_HttpRequestLogin;
                break;
            }
        }
    } while (false);
} else {
    proto.set_status_code(hm->resp_code);
    msgId = Proto::MsgId::MI_HttpOuterResponse;
}
auto pPacket = CreatePacket(msgId, pIdentify);
pPacket->SerializeToBuffer(proto);
return pPacket;
}
```

在分析请求时，当前仅处理了login请求，其他所有请求都是非法的。在框架中，为了使HTTP与自定义协议一致，定义了一个Proto：：Http结构来暂存数据。定义如下：

```
message Http {
    string body = 1;
    int32 status_code = 2;
}
```

综上所述，收到数据时，使用Mongoose库提供的函数分析http_message结构。该结构中的数据有两种情况，可能是一个请求数据头，也可能是一个响应返回数据头。

当收到的数据是一个外部请求时，其http_message：：method：：len值一定大于0。即使在浏览器中输入的是127.0.0.1：7071，没有带任何后续方法，这个值也是大于0的，因为method是以“/”开头的，空方法method的值为“/”，method：：len等于1。

当收到的数据是响应返回数据时，说明是框架主动向外部的HTTP服务器发起的一个请求返回的数据，例如验证账号时，我们需要向Nginx发送一个请求，收到返回时，http_message：：resp_code中就有当前返回的状态值，这个状态值是有标准的，404是大家比较熟悉的一个值，表示找不到这个请求的处理方式，无法处理。如果一切数据正常，状态值就是200。这些值是HTTP的常规值，是约定俗成、不可改变的。网络层只负责将这些数据分析出来，至于数据是请求还是响应返回，返回的是成功还是失败，作为网络层来说并不关心，只需要将它发送到框架中，再由框架中的其他组件去处理。

回到RecvNetworkBuffer：：GetPacket函数中，无论它分析出来的Packet数据是用于HTTP还是逻辑层面的协议，最终这个Packet会进入ThreadMgr：：DispatchPacket函数对协议进行分发。

9.4.3 处理HTTP数据的协议号

既然将HTTP数据打包成了Packet数据，那么这些HTTP的Packet必然有协议号。打开proto_id.proto的定义文件：

```
// HTTP listen的请求（外部请求）
MI_HttpBegin          = 10000;
MI_HttpInnerResponse  = 10001;           // 响应数据
MI_HttpRequestBad     = 10002;
MI_HttpRequestLogin   = 10003;
MI_HttpEnd            = 10499;
// HTTP connector 的消息（内部请求，外部返回）
MI_HttpOuterRequest   = 10500;           // 内部向外请求
MI_HttpOuterResponse  = 10501;           // 外部响应数据
```

工程中将协议号分成了两种：

- 情况1：从框架外部向框架发起请求，例如收到127.0.0.1:7071/login消息时，它对应的协议号为MI_HttpRequestLogin，除此之外，没有处理的都是MI_HttpRequestBad消息，返回404编码。当有请求被框架捕捉到时，处理完成之后会向网络层发送一个返回数据，这时返回包的Packet协议号为MI_HttpInnerResponse，所有的请求返回的消息协议号都是MI_HttpInnerResponse。

- 情况2：框架向外部HTTP服务发起请求，例如Nginx上有一个验证用户的请求，这时使用MI_HttpOuterRequest和MI_HttpOuterResponse分别作为请求与返回协议编号。只有一个协议号，一旦我们多个请求同时发送，它们会冲突吗？答案是不会。即使只有一个协议号，但是Socket却不同，所以处理的ConnectObj也是不同的。

下面来澄清“请求”这个概念，如果我们的服务器端口为8081，外部访问127.0.0.1:8081/login就是一个请求。在源代码中只实现了login请求，其他的请求都被认为是一个Bad请求。如果访问一个网站时输入一个随机的地址，例如www.baidu.com/fdfsdf，百度会提示我们访问的页面不存在，这就是一个Bad请求，因为百度服务器上根本没有处理随机输入的fdfsdf请求。

总之，通过NetworkType的类型可以区分出逻辑协议与HTTP，且为此生成了不同的Packet类，这些Packet类会广播到所有线程中，关心协议的组件会去处理。

下面我们通过实例进一步理解这个流程。

9.4.4 收到HTTP请求是如何响应的

当收到外部发起的login请求时，这个请求被封装成Packet，其MsgId为MI_HttpRequestLogin。处理组件类是LoginSyncComponent，存放在appmgr工程的login_sync_component.h文件中。

```
void LoginSyncComponent::Awake() {
    ...
    // http
    pMsgCallBack->RegisterFuntion(Proto::MsgId::MI_HttpRequestLogi
}
```

为什么选择LoginSyncComponent来处理login请求呢？很显然，因为login请求涉及的数据在LoginSyncComponent类中。该类中包含当前全部有效的login进程数据。在上面的代码中，当类从对象池中被唤醒时，在Awake函数中设置了MI_HttpRequestLogin的处理函数为HandleHttpRequestLogin，实现代码如下：

```
void LoginSyncComponent::HandleHttpRequestLogin(Packet* pPacket) {
    Json::Value responseObj;
    AppInfo info;
    if (!GetOneApp(APP_LOGIN, info)) {
        responseObj["returncode"] = 4;
        responseObj["ip"] = "";
        responseObj["port"] = 0;
    } else {
        responseObj["returncode"] = 0;
        responseObj["ip"] = info.Ip;
        responseObj["port"] = info.Port;
    }
    std::stringstream jsonStream;
    _jsonWriter->write(responseObj, &jsonStream);
    MessageSystemHelp::SendHttpResponse(pPacket, jsonStream.str()).
}

bool SyncComponent::GetOneApp(APP_TYPE appType, AppInfo& info) {
    if (_apps.size() == 0) {
        LOG_ERROR("GetApp failed. no more. appType:" << GetAppName
        return false;
    }
    // 找到第一个同类型数据
```

```

        auto iter = std::find_if(_apps.begin(), _apps.end(), [&appType
            return (pair.second.AppType & appType) != 0;
        ]);
        if (iter == _apps.end()) {
            LOG_ERROR("GetApp failed. no more. appType:" << appType);
            return false;
        }
        // 遍历后面的数据，找到最小值
        auto min = iter->second.Online;
        int appId = iter->first;
        for (;iter != _apps.end();++iter) {
            if (min == 0)
                break;
            if ((iter->second.AppType & appType) == 0)
                continue;
            if (iter->second.Online < min) {
                min = iter->second.Online;
                appId = iter->first;
            }
        }
        // 数据加1，以避免瞬间落在同一个App上，下次同步数据会将其覆盖为真实值
        _apps[appId].Online += 1;
        info = _apps[appId];
        return true;
    }
}

```

从基类的GetOneApp函数中取到一个真实的login进程的数据，将数据压入Json：：Value结构中，封装为JSON并转换为串，最后调用MessageSystemHelp：：SendHttpResponse函数向网络层发送数据。其实现代码如下：

```

void MessageSystemHelp::SendHttpResponse(NetworkIdentify* pIdentif
    SendHttpResponseBase(pIdentify, 200, content, size);
}
void MessageSystemHelp::SendHttpResponseBase(NetworkIdentify* pIde
    auto pNetworkLocator = ThreadMgr::GetInstance()->GetEntitySyst
    auto pNetwork = pNetworkLocator->GetListen(NetworkType::HttpLi
    if (pNetwork == nullptr) {
        LOG_ERROR("can't find network. http send failed.");
        return;
    }

    Packet* pPacket = CreatePacket(Proto::MsgId::MI_HttpInnerRespo
    std::stringstream buffer;

```

```

buffer << "http/1.1 " << status_code << " " << mg_status_messa
buffer << "Connection: close\r\n";
buffer << "Content-Type: application/json; charset=utf-8\r\n";
buffer << "Content-Length:" << size << "\r\n\r\n";
if (size > 0) {
    buffer.write(content, size);
}
pPacket->SerializeToBuffer(buffer.str().c_str(), buffer.tellp(
pNetwork->SendPacket(pPacket);
}

```

发送数据的时候，先找到NetworkType：：HttpListen的网络实例，再组织一个返回格式，然后将数据写入。在写入数据时，可以看到完全是一个文本的写入规则。本例的返回数据比较简单，采用Content-Length指定长度的方式。

9.4.5 发送HTTP返回数据流程

在发送返回数据时，依然是将返回数据写入一个Packet中，其MsgId为MI_HttpInnerResponse，但这个Packet中的Buffer数据与之前的所有Buffer数据不同，以往向其中写入的是一个protobuf的结构，但在HTTP返回中写的是一个字符串。Packet最终会发送到网络底层的缓冲区中。下面是底层写入HTTP结构的Packet和逻辑Packet的代码：

```
void SendNetworkBuffer::AddPacket(Packet* pPacket) {
    const auto dataLength = pPacket->GetDataLength();
    TotalSizeType totalSize = dataLength + sizeof(PacketHead) + si
    // 长度不够，扩容
    while (GetEmptySize() < totalSize) {
        ReAllocBuffer();
        //std::cout << "send buffer::Realloc. _bufferSize:" << _bu
    }
    // 对于HTTP来说没有自定义头
    const auto msgId = pPacket->GetMsgId();
    if (!NetworkHelp::IsHttpMsg(msgId)) {
        // 1.整体长度
        MemcpyToBuffer(reinterpret_cast<char*>(&totalSize), sizeof
        // 2.头部
        PacketHead head;
        head.MsgId = pPacket->GetMsgId();
        MemcpyToBuffer(reinterpret_cast<char*>(&head), sizeof(Pack
    }
    // 3.数据
    MemcpyToBuffer(pPacket->GetBuffer(), pPacket->GetDataLength())
}
```

当协议是HTTP时，上面的代码做了分支的处理，它没有自定义的头部数据了。HTTP的头在Packet类的Buffer中已经写好了，所以直接发送即可，这样它就会发送到浏览器上。整个处理请求数据的流程如图9-2所示。

图9-2清楚地呈现了HTTP数据在框架中的走向。简单来说，是先读取数据生成请求Packet，Packet到达处理函数之后，再将返回数据组织成一个返回Packet，返回Packet传递回网络层发送给对方。这样就形成了一个完整的HTTP响应。

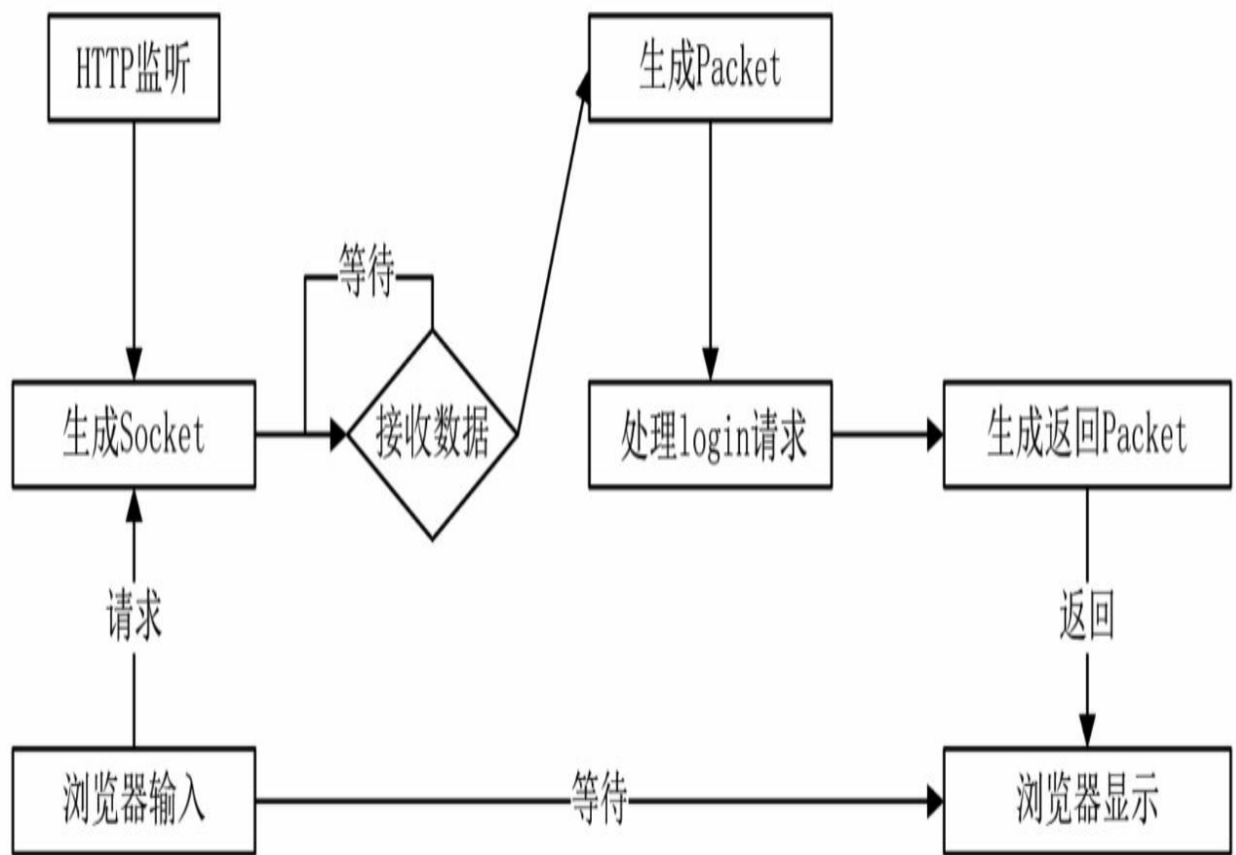


图9-2 HTTP请求流程

为Packet定义新的网络标识

在本例中，除了增加HTTP部分外，还有一个网络层的变化。在前面的示例中，为了让读者更容易理解网络层，我们仅采用Socket值来标识一个网络，简化了网络层的一些数据。

使用Socket单一值来标识一个网络通道其实是有不确定性的。既然给了一个通行牌，每人一个，为什么会有不确定性呢？它的不确定性主要归于异步。当我们使用多线程一瞬间密集向一个端口发起连接请求时，Socket值可能重用。说得更具体一点，当上千个连接同时发送某个网络端口时，创建Socket会有一个报错，Windows系统下是10035，Linux系统下是35，对于这个错误，大部分情况下认为不是错误，原因是对端口处理不过来了，对于发来的请求没有回应，但这个通道并没有关闭，是一个等待状态。当对端处理完成时，它会向连接端发起一个写指令。这时，这个通道就打开了。

在我们的框架中，当大量机器人密集向服务器发起Socket连接，又是多线程连接时，在Windows下非常容易出现Socket同值的情况，因为Windows下的Socket值是随机的。也就是说，两个Robot类可能共用了一个Socket值，听上去似乎不可能，可以做一个实验，用2000个线程同时连接一个端口，就会发现概率创建相同值的Socket。

假设有一个Socket值为1001，被两个Robot类分配到了。对于网络层来说，当第一个对象连接时，服务器处理不过来，返回了35错误，这个请求在服务端可能已经被抛弃了，但Robot A认为可以等待，并没有关闭这个Socket，第二个Robot B刚好又分配到了1001，它向服务器发起了一个连接请求，这时服务器同意了，这样Robot A和Robot B都拿到了这个号码牌，它们都认为自己已经连接成功。A和B轮询了同一个Socket值，注意通道只有一个。这是一种逻辑上的错误，这种情况在

Linux和Windows上都可能出现。

即使没有出现上述情况，还有另一种情况。假设Robot A使用值为1001的Socket进行登录，在这个过程中，某些原因导致底层网络中断，这时底层网络会向逻辑层发送一个断开消息，在这个消息还在消息队列中等待下一帧处理的时候，逻辑层在同一帧对1001发送了一条数据，这条数据放在了发送队列中待发送。好巧不巧，这时有一个新来的玩家Robot B登录了，正好重用了1001。这将产生什么样的后果呢？很可能在下一帧，这个新上线的Robot B将莫名其妙地收到一条数据，这条数据原本是想发给Robot A的。发送Packet包中，标记的Socket为1001，发送给Robot B并没有错。因为值为1001的Socket已经重新被分配给Robot B了，但实际上我们想要的效果并不是这样的。

综上所述，单纯依靠Socket来标识一个玩家是行不通的，需要使用一种新的网络身份法则。在这个新的网络身份中，我们不再使用单一的Socket值来定义一个网络通道。框架中定义了一个新的NetworkIdentify结构，该定义在09_02_http工程的socket_object.h文件中。在这个结构中有两个关键值：SocketKey和ObjectKey。相当于绳子的两边，一边绑定Socket数据，另一边绑定逻辑层的对象数据。一个Packet到达网络层需要对这两个值进行判断。

针对上面的情况，假如Robot A已经断开，新来Robot B，即使SocketKey与Packet中的SocketKey完全一致，但ObjectKey必然与旧的ObjectKey不一致，只要有不一致，Packet就需要被丢弃。结构NetworkIdentify定义如下：

```
struct NetworkIdentify {
public:
    NetworkIdentify() = default;
    NetworkIdentify(SocketKey socketKey, ObjectKey objKey);

    SocketKey GetSocketKey() const { return _socketKey; }
    ObjectKey GetObjectKey() const { return _objKey; }

protected:
    SocketKey _socketKey{ INVALID_SOCKET, NetworkType::None };
    ObjectKey _objKey{ ObjectKeyType::None , {0, ""} };
};
```

框架中至少有4种不同类型的NetworkType，SocketKey结构标识出NetworkType，方便找到目标。SocketKey结构和ObjectKey结构定义如下：

```
struct SocketKey {
    SOCKET Socket;
    NetworkType NetType;
    ...
};
struct ObjectKey {
    ObjectKeyType KeyType{ ObjectKeyType::None };
    ObjectKeyValue KeyValue{ 0, "" };
    ...
}
```

ObjectKey结构的使用是多样的，其枚举如下：

```
enum class ObjectKeyType {
    None = Proto::NetworkObjectKeyType::ObjectKeyTypeNone,
    Account = Proto::NetworkObjectKeyType::ObjectKeyTypeAccount,
    App = Proto::NetworkObjectKeyType::ObjectKeyTypeApp,
};
```

在使用枚举时，采用了在protobuf中定义的枚举，有3个值：None、Account和App。如果是Robot类或客户端与服务端的连接，就为ObjectKeyType::Account。除了玩家之外，需要进行网络连接的还有可能是App之间的连接。下面用实例代码来说明新机制。

9.5.1 使用网络标识创建一个连接

在前面的工程中，在线程中创建Robot对象时，每个Robot对象都是一个NetworkConnector，每个NetworkConnector都有一个Update事件需要处理，毕竟每帧都需要查看数据。如果有1000个Robot登录，就有1000个对象，这1000个对象要执行1000次：`select`或：`epoll`操作。

再来看一看NetworkListen对象是如何处理的，因为所有的对象都来自监听端口，所以所在的连接在监听上的对象都在一个NetworkListen中，即使有1000个连接，每一帧也仅执行一次：`select`或：`epoll`函数。

鉴于此，在09_02_http目录中重构了NetworkConnector类，希望它能够像NetworkListen对象一样，即使有1000个对象，每一帧也只执行一次：`select`或：`epoll`函数。将NetworkConnector类改为一个容器，而不再是一个NetworkConnector类实例对应一个网络通道。在这个容器中，每个ConnectObj都是一个连接通道，进程内所有的对外连接都由NetworkConnector类维护。

按照之前的流程，在创建NetworkConnector时就会将需要连接的IP和端口传递给类实例，现在它变成了容纳多个连接的容器。一个指定的连接是如何产生的呢？

连接一般都是主动的，不像监听产生的通道是在监听类中被动产生的。在工程中有两种主动连接：

- 一种是login进程需要主动连接到appmgr进程上。
- 另一种是像Robot类，要求连接到某个指定的login进程上。

这两种连接方式不同，采用了不同的处理流程。login进程到appmgr之间的连接是可控的，是预先已经十分明确的，所以采用的是初始化创建。打开login工程中的main.cpp文件，创建NetworkConnector类的代码如下：

```
pThreadMgr->CreateComponent<NetworkConnector>(ConnectThread, false
```

在创建NetworkType：：TcpConnector类型的NetworkConnector类时，参数已经说明了NetworkConnector要优先创建两个App连接，分别指向appmgr和dbmgr两个进程。

再来看看Robot的主动连接，采用的是协议方式。当Robot需要进行一个连接时，无论是对login还是HTTP的连接都需要发起一个协议MI_NetworkConnect。对应的协议体结构为Proto：：NetworkConnect，结构定义如下：

```
message NetworkConnect {
    int32 network_type = 1;
    NetworkObjectKey key = 2;
    string ip = 3;
    int32 port = 4;
}
```

该结构的第一个属性network_type指定连接的类型是进行TCP连接还是进行HTTP连接，属性key给出连接的ObjectKey。对于Robot来说，它需要给出一个类似{ObjectKeyType：：Account，{0，Account}}这样的键（Key）。最后在这个协议中还要给出连接对方的地址与端口，毕竟没有网络地址怎么实现网络连接呢。

处理协议MI_NetworkConnect的是NetworkConnector类。来看看它对协议的处理实现：

```
void NetworkConnector::HandleNetworkConnect(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::NetworkConnect>();
    if (proto.network_type() != (int)_networkType)
        return;
    ObjectKey key;
    key.ParseFromProto(proto.key());
    ConnectDetail* pDetail = new ConnectDetail(key, proto.ip(), pr
        _connecting.AddObj(pDetail);
}
```

NetworkConnector类在login工程中有两个实例：一个用来处理HTTP；另一个用来处理游戏逻辑协议。处理HTTP的NetworkConnector实

例，主要任务是向外部（本书中是Nginx）发送HTTP请求；而处理游戏逻辑TCP协议的NetworkConnector实例，主要任务是处理login主动向外发起的TCP连接，例如连接到dbmgr和连接到appmgr的网络连接。

因此，开始处理函数时就对NetworkType进行了判断，与自己不符合的协议不进行处理。随后，框架中增加了一个ConnectDetail结构用于尝试连接，每一个想要创建连接请求，其数据都会存在于一个ConnectDetail结构中，等待下一帧处理。下一帧，ConnectDetail结构被删除，一个新的ConnectObj生成。尝试连接的代码在每帧中处理如下：

```
void NetworkConnector::Update() {
    // 有新的连接请求
    if (_connecting.CanSwap())
        _connecting.Swap(nullptr);
    // 创建新的连接
    if (!_connecting.GetReaderCache()->empty()) {
        auto pReader = _connecting.GetReaderCache();
        for (auto iter = pReader->begin(); iter != pReader->end();
            if (Connect(iter->second)) {
                _connecting.RemoveObj(iter->first);
            }
        }
    }
    Epoll();
    OnNetworkUpdate();
}
```

在这段代码中没有加锁，因为协议处理与Update操作是串行的，所以没有必要加锁。当NetworkConnector在执行Update函数时，发现有了新的连接数据，ConnectDetail实例就会被取出来，调用Connect函数尝试连接，给它创建一个ConnectObj。

关于NetworkConnector::Connect(ConnectDetail*pDetail)这个函数的具体实现，这里就不再详细讲解了，可参见源代码。总之，它让一个请求转换成一个处于Connecting状态的ConnectObj。

需要注意的是，在之前的逻辑中，认为一个ConnectObj一定是一个可用的网络连接，但现在它不是了。它有了一个状态：

```
enum class ConnectionStateType {  
    None,  
    Connecting,  
    Connected,  
};
```

对于NetworkConnector类来说，它维护的ConnectObj数据大部分先是处于Connecting（连接中）状态，随后才是Connected（已连接）状态。而对于NetworkListen实例来说，它维护的ConnectObj数据全部是Connected状态，因为一个通过：`accept`函数得到的通道一般都是已经完成了3次握手的连接。因此，在NetworkListen类中的ConnectObj对象不存在Connecting这个状态。

不论是NetworkListen还是NetworkConnector，如果产生了断线，这个ConnectObj就被删除了。一个ConnectObj被销毁之后，断线协议一定会发送到逻辑上层。如果它是一个玩家，就有相应的下线处理。如果这个连接的ObjectType表明它是一个服务端进程之间的连接，那么一个重连接协议将被发起，协议号依然是MI_NetworkConnect，具体的代码可参见NetworkLocator组件。

9.5.2 使用网络标识发送数据

为了适应新的网络身份机制，需要改造ConnectObj连接对象。首先它一定是一个NetworkIdentify，拥有网络标识，一头绑定了网络，另一头绑定了一个逻辑层的对象。在创建ConnectObj时需要输入NetworkIdentify相关数据。ConnectObj类的唤醒函数定义如下：

```
class ConnectObj : public Entity<ConnectObj>, public NetworkIdentify
{
public:
    void Awake(SOCKET socket, NetworkType networkType, ObjectKey k
    ...
}
```

同时，还为ConnectObj准备了一个状态。这一点不难理解，连接是一个异步过程，只有确认已经连接成功了，才认为这个通道是可用的。当网络层传来可读或可写时，这个通道就成功创建了，这时会调用函数ChangeStateToConnected来改变ConnectObj的状态。

```
void ConnectObj::ChangeStateToConnected() {
    _state = ConnectStateType::Connected;
    if (GetObjectKey().KeyType == ObjectKeyType::App) {
        auto pLocator = ThreadMgr::GetInstance()->GetEntitySystem(
        pLocator->AddNetworkIdentify(GetObjectKey().KeyValue.KeyIn
    } else {
        // 通知逻辑层连接成功了
        MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_Network
    }
}
```

一旦成功连接，需要向逻辑层发送一个MI_NetworkConnected协议，通知之前给我们ObjectKey的对象连接成功了，同时将当前的SocketKey赋值给ObjectKey的对象。还是以Robot为例，看看MI_NetworkConnected协议的处理。Robot类中对于该协议的处理函数为HandleNetworkConnected，在Robot类从对象池中被唤醒时，进行了协议号的处理函数的注册，其注册与实现代码如下：

```

void Robot::Awake(std::string account) {
    // message
    auto pMsgCallBack = new MessageCallBackFunctionFilterObj<Robot
    AddComponent<MessageComponent>(pMsgCallBack);
    pMsgCallBack->GetPacketObject = [this](NetworkIdentify* pIdent
        if (_objKey == pIdentify->GetObjectKey())
            return this;
        return static_cast<Robot*>(nullptr);
    };
    pMsgCallBack->RegisterFunctionWithObj(Proto::MsgId::MI_Network
    ...
}
void Robot::HandleNetworkConnected(Robot* pRobot, Packet* pPacket)
    if (_socketKey.NetType != NetworkType::None && _socketKey.Sock
        // 有新的连接后, 旧的连接需要关闭
        MessageSystemHelp::DispatchPacket(Proto::MsgId:: MI_Networ
    }
    _socketKey = pPacket->GetSocketKey();
}

```

Robot在生成时注册了MI_NetworkConnected协议的处理函数。当收到连接成功之后, 会将生成的SocketKey赋值给自己本地的SocketKey。在注册这个函数时, 注意使用的是RegisterFunctionWithObj方法。这是一个过滤流程, 一般情况下, 协议是无差别发送给所有线程的, 但是RegisterFunctionWithObj需要调用GetPacketObject函数来过滤对象, 以避免收到不需要的数据。上面的代码中有对于GetPacketObject的实现, 当ObjectKey一致时, 就是我们需要接收Packet的对象了。

对于Robot的状态类来说, 一开始没有连接成功之前, 它都是处于正在连接RobotStateHttpConnecting状态, 这是等待HTTP连接的状态。Robot类模拟了客户端的操作, 首先会发起一个HTTP请求, 得到一个可以登录的login进程信息, 连接成功之后, 再发起真正的登录操作。在等待HTTP连接的状态下, 每一帧都在监听是否连接成功, 其处理函数在状态类中, 实现如下:

```

RobotStateType RobotStateHttpConnecting::OnUpdate() {
    auto socketKey = _pParentObj->GetSocketKey();
    if (socketKey.Socket != INVALID_SOCKET) {
        if (socketKey.NetType != NetworkType::HttpConnector) {
            LOG_ERROR("error.");
        }
    }
}

```

```
        MessageSystemHelp::SendHttpRequest(_pParentObj, _ip, _port  
        return RobotStateType::HTTP_Connecttd;  
    }  
    return GetState();  
}
```

当Robot: : HandleNetworkConnected收到网络层的消息,更改了自己的SocketKey时, RobotStateHttpConnectting状态马上就会反应过来,发现Socket已经不是一个非法值了。这时,此状态认为已经连接成功,会向网络通道发起一个HTTP请求。调用的是函数 MessageSystemHelp: : SendHttpRequest, 在这个请求中没有参数,最后一个值为nullptr。同时, Robot的状态变成了RobotStateType: : http_Connecttd。

9.5.3 向外部请求HTTP数据

前面我们讲了如何响应从外部发起的HTTP请求，现在讨论一下向外部的HTTP服务发送一个请求的流程。这个流程在验证账号和Robot类向appmgr请求login进程信息时都会用到。

以Robot类为例向appmgr请求数据时，调用代码如下：

```
MessageSystemHelp::SendHttpRequest(_pParentObj, _ip, _port, _metho
```

该函数的实现如下：

```
void MessageSystemHelp::SendHttpRequest(NetworkIdentify* pIdentify
    Packet* pPacket = CreatePacket(Proto::MsgId::MI_HttpOuterReque
    std::stringstream buffer;
    buffer << "GET " << method;
    if (pParams != nullptr) {
        buffer << "?";
        for (auto iter = pParams->begin(); iter != pParams->end();
            buffer << iter->first << "=" << iter->second << "&";
        }
    }

    buffer << " HTTP/1.1" << "\r\n";
    buffer << "Host: " << ip.c_str() << ":" << port << "\r\n";
    buffer << "Content-Type: application/json; charset=utf-8" << "
    buffer << "Accept: */*,text/*,text/html" << "\r\n";
    buffer << "\r\n";

    pPacket->SerializeToBuffer(buffer.str().c_str(), buffer.tellp(
    SendPacket(pPacket);
}
```

在该函数中写了一个HTTP格式的请求协议，并将数据发送出去。当ConnectObj收到返回协议时，数据封装成MI_HttpOuterResponse协议号的Packet，这个Packet在ConnectObj中生成，附加上了ConnectObj类的NetworkIdentify信息，该信息只会与唯一一个Robot对应。在Robot中对于这个协议的处理如下：

```

void Robot::HandleHttpOuterResponse(Robot* pRobot, Packet* pPacket)
{
    auto protoHttp = pPacket->ParseToProto<Proto::HTTP>();
    const int code = protoHttp.status_code();
    if (code == 200) {
        const auto response = protoHttp.body();
        Json::CharReaderBuilder readerBuilder;
        Json::CharReader* jsonReader = readerBuilder.newCharReader();
        Json::Value value;
        jsonCPP_STRING errs;
        const bool ok = jsonReader->parse(response.data(), response.data() + response.data().size(), &value, &errs);
        if (ok && errs.empty()) {
            const auto code = value["returncode"].asInt();
            if (code == Proto::LoginHttpReturnCode::LHRC_OK) {
                _loginIp = value["ip"].asString();
                _loginPort = value["port"].asInt();
                ChangeState(RobotStateType::Login_Connectting);
            }
        } else {
            LOG_ERROR("json parse failed. " << response.c_str());
        }
        delete jsonReader;
    } else {
        LOG_ERROR("http response error:" << code << "\r\n" << protoHttp.c_str());
    }
}

```

收到正确的数据后，Robot会得到正确的login进程的IP地址，此时调用了ChangeState（RobotStateType：：Login_Connectting），Robot类的状态从HTTP阶段修改为与login正在连接的状态。之后的流程就不再介绍了，回到了之前熟悉的工程流程上。流程图如图9-3所示，总结了这些步骤。

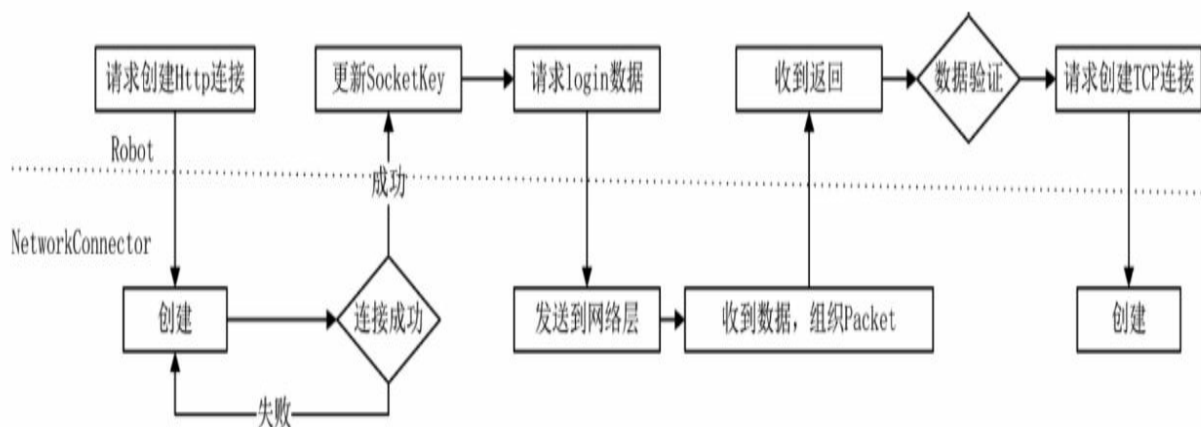


图9-3 Robot登录流程

现在，启动robots进程，批量登录1000个账号，这1000个账号的网络处理都会集中到一个NetworkConnector中处理。

除了Robot类之外，login进程在验证账号时向Nginx发送了一个PHP请求，用于模拟平台账号验证。之前是使用curl第三方库实现的。现在框架实现了自己的HTTP处理，可以不再需要curl库了。其实现原理和Robot类向服务端发送HTTP请求一致。简而言之，当Account类收到C2L_AccountCheck账号验证协议后，发起一个MI_NetworkConnect协议想要创建一个连接，在协议的ObjectKey标志中带上自己的标识。当这个请求成功之后，会收到来自底层的MI_NetworkConnected协议，表示连接成功，随后向这个网络标识发送一个HTTP的验证请求。请求发送之后会收到MI_HttpOuterResponse协议，得到验证的结果。具体功能参见源代码，这里不再阐述。

HTTP分块

在前面的章节中阐述了HTTP的协议格式与框架中的HTTP数据的收发流程。本节我们回到HTTP格式上，再来说明一下HTTP的块数据传递是怎么回事。

单从代码来看很难厘清它的格式。为了测试数据分块，在robots工程中写了一个测试类TestHttpLogin，本节的源代码依然在09_02_http目录中。

启动robots进程，启动完成之后输入“http-check test test”，这个指令发起了一个验证账号的HTTP请求。这个HTTP请求是Nginx下的一个PHP接口。修改engine.yaml文件，确保配置中的IP正确。其调用之后的请求数据如下：

```
http -check test test
connected. socket:6 networkType:HttpConnector connect type:Account
GET /member_login.php?account=test&password=test& http/1.1
Host: 192.168.0.120:80
Content-Type: application/json; charset=utf-8
Accept: */*,text/*,text/html
```

返回数据如下：

```
http/1.1 200 OK
Server: nginx/1.12.2
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
```

```
X-Powered-By: PHP/5.4.16
(回车空行)
10(长度)
{"returncode":2}
0(长度)
(回车空行)
```

从打印输出的数据来看，在请求返回数据中并没有Content-Length属性，没有长度也就不知道请求返回数据有多长，但有Transfer-Encoding: chunked属性说明数据是按块传递的。要理解这个原理，可以举一个例子，我们刷新网页时会先弹出一部分内容，过一会儿再弹出全部内容。

基于用户体验，在HTTP中有一个机制，不依赖头部的长度也可以快速知道传输的内容。也就是说，HTTP头结束之后，即在/r/n/r/n之后的数据是一个数据串，这个串的格式是：长度+/r/n+数据+/r/n。长度值不包括它结尾的结束符（\r\n），也不包括分块数据结尾的结束符。最后一个分块长度值必须为0，对应的分块数据没有内容，表示协议结束。

如果一个网页有100KB，发送请求之后返回了头部，这时头部就有属性Transfer-Encoding: chunked。表9-1整理了每一步的数据内容，以帮助理解这个过程。

表9-1 HTTP分块数据

发送数据	说 明
第一次收到数据	头部\r\n
第二次收到数据	20(长度)\r\n 内容\r\n
第三次收到数据	10(长度)\r\n 内容\r\n
第N次收到数据	...
结束	0\r\n(或者 \r\n\r\n)

当然，在真正的网络传输过程中，并不是真的一次一次这样规整地收到数据，有可能第二段数据与第三段数据一起到达网络缓冲区，也有可能到达网络缓冲区的只有第二段一半的数据。鉴于此，回到代码中，在RecvNetworkBuffer::GetHttpPacket函数分析数据时有一个等待长度的操作：

```
bool isChunked = false;
const auto mgTransferEncoding = mg_get_http_header(&hm, "Transfer-
if (mgTransferEncoding != nullptr && mg_vcasecmp(mgTransferEncodin
    isChunked = true;
    // 后面的数据还没有到达
    if (recvBufLength == headerLen)
        return nullptr;

    bodyLen = mg_http_get_request_len(_buffer + _beginIndex + head
    if (bodyLen <= 0)
        return nullptr;

    bodyLen = _endIndex - _beginIndex - headerLen;
}
```

在这一段代码中，mg_http_get_request_len函数本来是用来判断request头部分长度的，但是我们用这个函数来判断块数据是否接收完成。它们有一个共同的标志就是以双/r/n或0\r\n结束。一定要等到结束符到来，这个块的返回结果才是完整的，只有所有块都接收完成了，生成的Packet才是完整的。

机器人测试批量登录

在本章中解决了两个问题：

- (1) HTTP的监听数据如何处理外部对服务端的请求并正确返回。
- (2) HTTP的请求数据如何处理向外部HTTP服务发送请求并正确返回。

关于HTTP的内容就写到这里，鉴于之前关于网络的内容已经有很多了，这部分尽量减少网络相关内容，以避免枯燥，但是09_02_http目录中的项目与之前的项目在网络机制上非常不同，如果没有理解，不妨调试一下源代码。不同之处主要表现在两个方向：

- (1) 网络识别不再是Socket，而是NetworkIdentify结构。
- (2) 在网络内存组织上的不同。旧工程中的每个连接都要生成一个NetworkConnector，而最新工程是放在一个NetworkConnector中的。这两者之间的比较可以对网络模型有一个更加深刻的认识。一个进程中的两个线程，如果数据要共享的话，就必须加锁，但是对于网络通道来说，只要Socket是给定的，那么它在任何一个线程都可以读出数据。即使是监听Socket，也与其监听对象的通道是独立的，即一方的关闭与异常不会影响另一方。

为了证明这个观点，我们可以做一个测试，修改一下监听的NetworkListen::Update，在头部增加几行代码：

```
void NetworkListen::Update() {  
    if (!_connects.empty() && _masterSocket != INVALID_SOCKET) {
```

```

        DeleteEvent(_epfd, _masterSocket);
        _sock_close(_masterSocket);
        _masterSocket = INVALID_SOCKET;
    }
    _mainSocketEventIndex = -1;
    epoll();
    ...
}

```

在头部的if中，我们写了一个条件判断，如果connects集合不为空，就将监听端口从epoll的队列中移除，并将它关闭。也就是说，只要有一个网络通道打开，我们不再关心监听端口的数据。即使不关心监听端口，但已打开的通道却并没有受限，还是可以正常地收发数据的。这时，再启动一个robots进程来进行登录，却无法登录成功了，因为监听端口已经关闭，数据没有在逻辑层处理了。

这个例子说明，无论是：`epoll`函数还是：`select`函数中传入的每一个Socket值都是独立的，它们之间没有因果关系。即使将一个监听的Socket和一个主动连接生成的Socket放在一起调用：`epoll`函数或：`select`函数，也是完全没有问题的。可能网络通道生成的方式不同，但是一旦生成，在读取或写入数据时就是完全相同并且独立的。

按照这个思路，是不是可以将连接类和监听类合并起来呢？我们将这个问题留给读者自己思考，可以在09_02_http的项目中动手试一试。希望通过本章的介绍，读者能对网络底层收发有更加深刻的认识。

下面对Robot类的登录步骤进行简短的总结，以梳理分布式登录的流程。在之前的工程中，login进程只有一个，Robot类直接连接到login进程上。现在login有了多个进程，登录流程需要做出调整，首先向appmgr发起一个HTTP请求，得到login进程的登录地址。着重看一下新增部分的代码，首先需要重新定义Robot类的状态，Robot类增加了Http_Connecting和Http_Connected两个新的状态。在Http_Connecting状态下，Robot需要向服务端发送一个HTTP连接，如果连接成功就进入Http_Connected状态，发起HTTP请求得到一个JSON数据，分析出登录的login进程的IP地址，再发起TCP连接。Robot的状态在libserver工程中的robot_state_type.h文件中：

```
enum class RobotStateType {
```

```

None = -1,
HTTP_Connecting = 0,           // HTTP请求
HTTP_Connected,
Login_Connecting,              // 正在连接 Login
Login_Connected,               // 连接成功
Login_Logined,                 // 登录成功
...
};

```

一旦Robot实例生成，它的初始状态是一个Http_Connecting。状态的进入函数事件实现如下：

```

void RobotStateHttpConnecting::OnEnterState() {
    Proto::NetworkConnect protoConn;
    protoConn.set_network_type((int)NetworkType::HttpConnector);
    ObjectKey key{ ObjectKeyType::Account, {0, _pParentObj->GetAcc
key.SerializeToProto(protoConn.mutable_key());

    const auto pYaml = ComponentHelp::GetYaml();
    const auto pCommonConfig = dynamic_cast<CommonConfig*>(pYaml->
protoConn.set_ip(pCommonConfig->Ip.c_str());
    protoConn.set_port(pCommonConfig->HttpPort);

    _ip = pCommonConfig->Ip;
    _port = pCommonConfig->HttpPort;
    _method = "/login";

    MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_NetworkConn
}

```

状态类从engine.yaml配置文件中取读了appmgr的HTTP地址与端口，同时创建一个新的HTTP连接。engine.yaml这个配置本来是在服务器上的，因为Robot和服务端的进程是放在一起的，所以直接读取了。在真实游戏部署的时候，一般采用域名的方式来获取登录进程数据。

发送连接请求之后，Robot就只有等待了。当连接成功时，NetworkConnector类会发送一个MI_NetworkConnected协议。Robot处理这个协议时，将重置自己的SocketKey。一旦状态检查到SocketKey发生了改变，就会马上做出反应，将状态改为Http_Connected，且将请求数据发送出去。HTTP返回数据是JSON类型的，如果解析成功，Robot就会进入Login_Connecting状态。

关闭之前打开的LOG_HTTP_OPEN宏，重新编译一下工程，来看看Robot进程测试1000个账号登录的效率。

```
[root@localhost bin]# ./robotsd
...
Http-Connecting over. time:0.153479s
Http-Connected over. time:0.919222s
Login-Connecting over. time:1.40633s
Login-Connected over. time:2.04906s
Login-SelectPlayer over. time:3.07127s
```

经过几次反复的测试，登录1000个账号的时间在3秒之内，上面的数据给出了每一个阶段消耗的时间。数据是在Linux下测试的，比起之前的登录流程增加了1000个HTTP数据的请求。

总结

本章中，我们为框架增加了HTTP的格式分析，提供了HTTP服务。在分布式登录中，启动了多个login进程，这些进程会定时将自己的当前状态发送到appmgr进程中，appmgr提供了HTTP服务，客户端可以通过HTTP请求得到一个用于登录的login进程的信息，以实现登录。

现在，我们已经从多个login中选择一个进程登录，第10章将进入game游戏逻辑进程。

第10章

分布式登录与Redis内存数据库

本章将建立最终的框架需要的所有进程，即增加game和space进程。同时，引导玩家进入game进程中，在这个过程中使用了Redis内存数据库。本章包括以下内容：

- 增加game和space工程。
- Redis内存数据库的使用。
- 完善登录流程。

10.1

game与space的定位

本节将建立最终的框架需要的所有进程，即增加game和space进程。图10-1展示了game和space进程之间的关系。

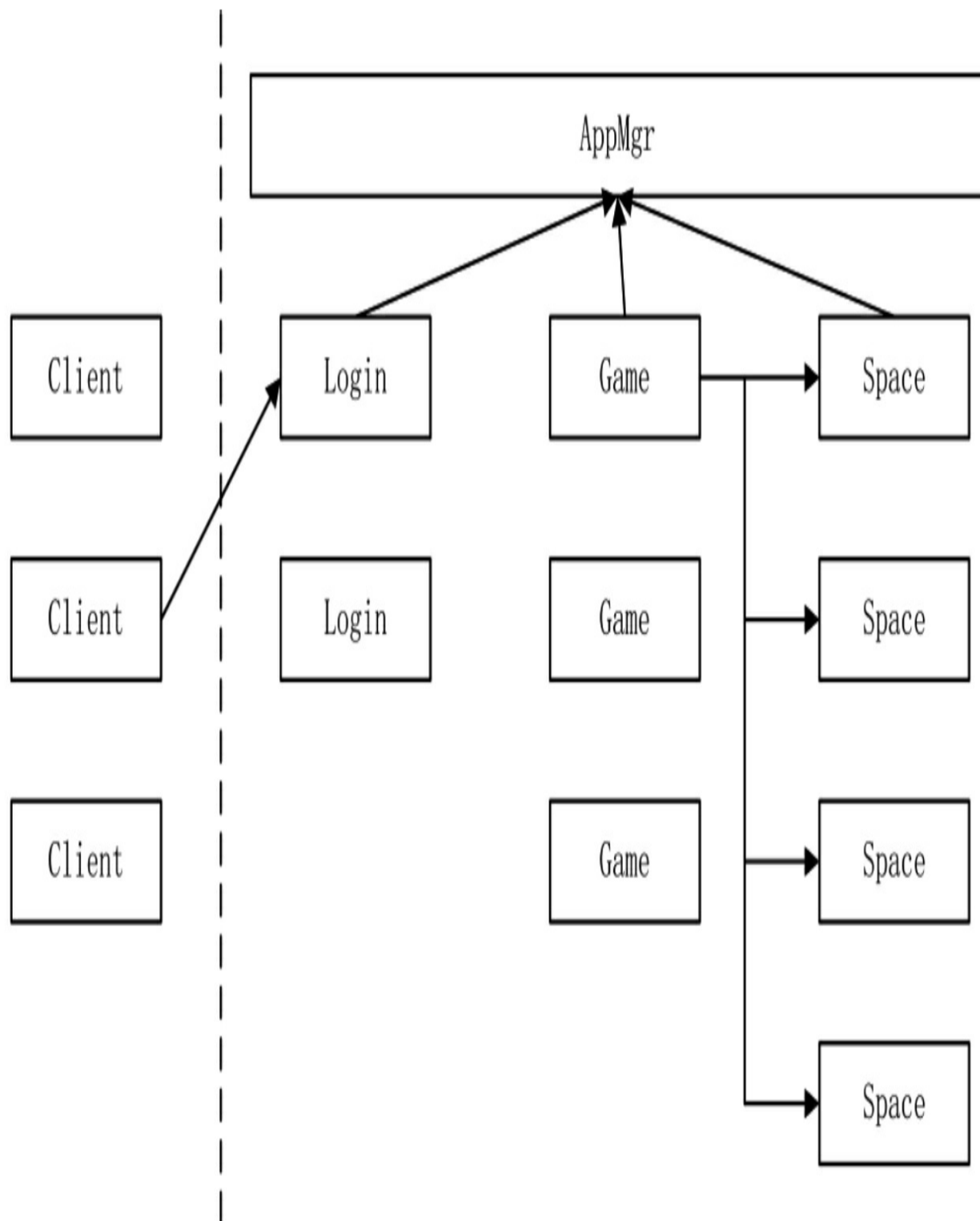


图10-1 进程分布

客户端通过HTTP请求到一个合适的login进程进行账号验证，验证成功之后，login会分配一个合适的game进程给客户端进行连接，game进程提供玩家角色加载，如果有多角色进行选角操作，space进程就为其分配最终的地图实例。

以MOBA游戏的流程为例，当我们登录游戏进行玩家对战匹配之后会进入副本，这个副本的真实数据就运行在space进程上。space进程对玩家是不可见的，game进程会进行代理。

为什么要把game和space分开，放在一起不是更容易开发吗？

当然是为了容纳更多的玩家，假如game进程中可以连接一万个玩家，这一万个玩家可能在10个space进程中玩游戏，从而分散了计算压力。space承载了副本功能，但除了这个功能之外，还可以有其他功能。例如，我们可以创建一个特别的聊天组件，在这个组件中处理所有玩家的聊天信息，并将它们发送到目的地。

space进程很重要的作用是实现聚合功能。例如，我们可以将游戏中的帮派系统放在space上，在这个系统中处理所有帮派相关的事宜。假设有两个玩家分别在game1和game2进程中，他们属于同一帮派，两个玩家登录时，帮派系统会收到他们登录的信息，如果在game1上的玩家推送了一个帮派聊天协议，最终这个协议会到达帮派系统在space的实例上，帮派系统会将这个协议发送到本帮派所有玩家的客户端上。正是因为有了帮派系统的存在，所以不需要关心对方是否存在于本进程中。

总之，space进程是为了解决聚合问题，运行在space进程上的每一个实体都应当是一个空间。这里的空间可以是地域上的空间，也可以是逻辑上的空间，空间中的对象需要相互交互。

现在，框架中有6个进程，但是这些进程可以根据具体需求调整。可能在某些需求下，需要将login和game进程合并，这个过程非常容易。因为所有login的组件都是Actor模式的，放在任何进程中都可以正常工作，这里不再详述，可参见allinone工程。

接下来需要明确一个问题，这些进程两两之间如何连接必须要有一个规范。设计原则如下：

1.space进程

- 1.1 所有space进程都开放一个监听端口。
 - 1.2 所有space进程都连接所有game进程，向其发送本进程的状态，也就是说，所有game进程
 - 2.1 所有game进程都自动连接所有space端口。
 - 2.2 所有game进程都开放一个监听端口，用于客户端连接。
 - 2.3 所有game进程都连接appmgr进程，向其发送本进程的状态。appmgr会将这些状态
 - 2.4 所有game进程都连接dbmgr，用于数据读取。
 - 3.login进程
 - 3.1 所有login进程都开放一个监听端口，用于客户端连接。
 - 3.2 所有login进程都连接appmgr，向其发送本进程的状态，以便HTTP请求时，appmgr
 - 4.appmgr进程
 - 4.1 appmgr只有一个唯一实例，提供HTTP端口，用于客户端HTTP请求。
 - 5.dbmgr进程
 - 5.1 dbmgr只有一个唯一实例，提供DB存储。
-

剔除一些细节，从头梳理一下，一个完整的登录流程是这样的：

(1) 客户端通过HTTP连接得到一个可用于登录的login进程的IP和端口。

(2) 连接到login进程上进行账号验证，成功登录之后，创建角色和选择角色，目前大部分游戏都是单一角色，可能没有选择角色这一功能。选定角色之后，login进程会分析所有game进程的数据，选择一个合适的发送给客户端，并同时生成token。

(3) 客户端通过token连接到game进程上，正式开始游戏，进入某个地图。

(4) 收到进入地图的协议，game进程首先向appmgr发起请求，判断是否有该地图的实例，如果没有，就马上在一个合适的space中创建一个地图实例。同时，在game进程中生成WorldProxy实例用于代理地图数据，WorldProxy一旦生成，所有数据均会转发至space进程，game进程则作为一个中转进程，类似网关进行数据转发。地图建立成功之后，向客户端发送进入地图协议，加载地图与玩家模型，玩家正式进入地图。

如果读者第一次接触游戏，最后一步一定让读者有一种头晕目眩的感觉。先来考虑前3步，如何登录game进程，虽然登录看上去只有短短几步，但是需要考虑的东西却不少。以验证账号为例，如果有玩家使用同一个账号同时在login1和login2进程登录，该如何处理呢？我们先将

这个问题放在这里，在后面的章节中再来解决，本节先为所有进程建立两两连接的网络，把框架建立起来。

在有些架构中，在网络连接部分，在需要时才建立网络连接。例如game和space进程，开始可能没有建立game与任何space进程的网络通信，需要时才建立一个，但是这种连接有一定的风险性，可能会丢失数据。创建一个网络需要时间，当我们发送数据时才创建网络，这就涉及Packet的缓存问题，Packet需要缓存到它依赖的网络创建好了再发送。

鉴于此，本书的框架中选择在所有进程初始化时就连接所有需要连接的目的地。这样，在玩家需要跳转到space2上时，如果space2出现异常宕机，在game进程上就会抛出一个断线协议。当玩家发送跳转协议时，可以检测到该网络连接已经找不到了，从而向客户端发送异常提示。

本节讨论的多进程项目已经在10_01_game目录中创建出来了，增加了game和space两个新的工程。目前space没有任何组件，只是一个空工程。在10_01_game目录的项目中，我们加入了登录game的操作。先来运行一下看看结果，再分析具体的代码。先启动allinone，再启动robots，登录一个账号进行测试：

```
login -a test
test begin
Http-Connecting over. time:4e-06s
Http-Connected over. time:0.00738s
Login-Connecting over. time:0.017818s
Login-Connected over. time:0.021281s
Login-SelectPlayer over. time:0.048624s
Game-Connecting over. time:0.059159s
Game-Connected over. time:0.063525s
```

从Robot类的状态来看，一个账号从登录到最后完成了7个状态的功能，最终这个Robot成功登录game进程。登录成功的标志是Robot类向game进程发送C2G_LoginByToken协议，使用token登录，服务端返回C2G_LoginByTokenRs协议，如果成功，Robot的状态就变为Game-Connected。

测试10_01_game工程时有两种启动方式，第一种是allinone，这种

方式很简单；第二种是以多进程启动。现在我们至少需要启动5个进程，它们分别是：

```
[root@localhost bin]# ./dbmgrd
[root@localhost bin]# ./appmgrd
[root@localhost bin]# ./logind -sid=101
[root@localhost bin]# ./gamed -sid=201
[root@localhost bin]# ./spaced -sid=301
```

进程的参数101和201是在engine.yaml文件中配置的，根据配置可以读出相关的监听地址。下面来看login启动后的打印：

```
[root@localhost bin]# ./logind -sid=101
...
connected appType:appmgr appId:0 socket:9 networkType:TcpConnecto
connected appType:dbmgr appId:0 socket:10 networkType:TcpConnecto
```

进程login连接上了appmgr和dbmgr。这部分代码在NetworkConnector初始化时就加上了，前面已经讲过。下面来看game进程启动后的打印输出：

```
[root@localhost bin]# ./gamed -sid=201
...
connected appType:dbmgr appId:0 socket:488 networkType:TcpConnect
connected appType:appmgr appId:0 socket:492 networkType:TcpConnec
connected appType:space appId:301 socket:508 networkType:TcpConne
```

game进程连接了id为301的space进程。如果这时关闭301进程，game进程就会敏锐地察觉到并打印以下信息：

```
[DEBUG] - remove appType:space appId:301
```

这时打开301进程，game会马上重新连接上去。在整个框架中，当我们用多进程的方式启动时，不需要启动顺序，所有的进程都会对自己需要的连接进行自动重连。总之，我们已经将所有需要的进程进行了两

两连接，在后面使用的过程中只需要配置好数据即可。可以在 `engine.yaml` 的 `space` 下配置多个需要开启的 `space` 进程。关于自动重连网络的部分，本节不再详细介绍。下面从代码分析的角度看看 `Robot` 是如何从 `login` 进程跳转到 `game` 进程的，它是根据什么来选择跳转以及跳转的，之后如何验证正误。

10.1.1 选择合适的game进程

进程login如何知道在众多的game进程中哪一个是合适的呢？有无数种方法可以达到这个目的，但这里我们只讨论两种方案：

第一种方案是让game进程向第三方的appmgr进程定时发送自己的状态信息，例如有多少人在线，当login进程需要时，向appmgr请求数据，从而获得一个负载最小的game进程。这个方案是一个异步方案，而且每次向appmgr请求数据时必然要经过一个串行处理。appmgr上需要编写一个组件来处理这个消息，当协议到达这个消息时就进行串行的处理，从而会降低效率。

图10-2展示了这两种方案的流程。左边是第一种方案，右边是第二种方案。再来看看第二种方案。

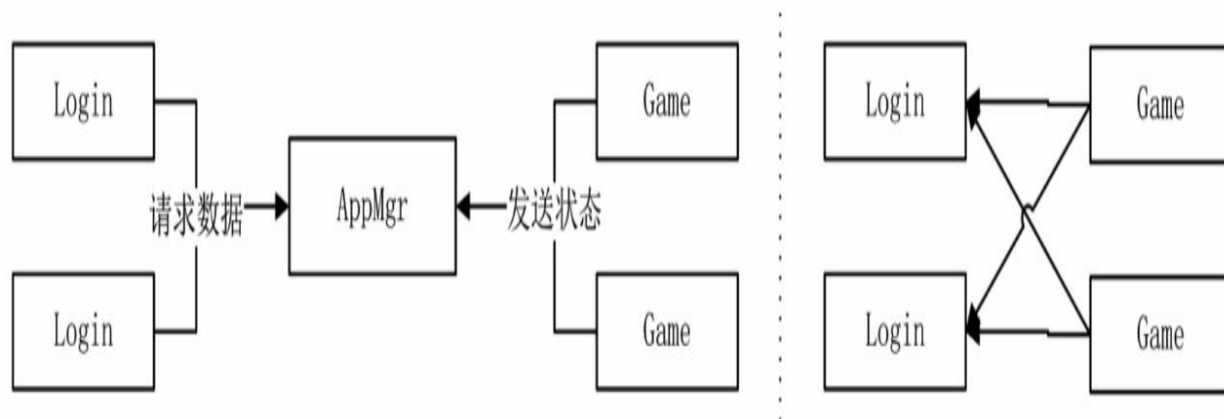


图10-2 选择Game进程

在第二种方案中，game与login两两通信，game定时向所有login进程发送自己的状态信息。当login进程需要时，不需要异步过程，直接就知道哪个game进程负载最小。但是这个方案有一个问题，如果有两个人分别处于login1和login2进程上，在一瞬间两个login进程分析出来的数据得到的最小负载的game进程是同一个进程，这意味着从短时间来看，负载可能不是精确均衡的，从长期来看，这个问题会得以解决。

除此之外，第二种方案还有一个问题，如果我们有10个线程，也就

是说每个game都必须连接10个login，连接这10个login的原因只是为了发送状态数据，显然有点不值。为此，合并一下两种方案。login进程不要直接和game进程产生网络连接，game进程向appmgr进程发送自己的状态，但是每隔两秒，appmgr进程将收集到的数据推送给所有login。这样既减少了网络连接，在login进程上分配game进程时又不需要使用异步数据。

下面来看具体实现的步骤。本节源代码位于10_01_game目录中。先来梳理一下思路。如果要均衡玩家在各个game进程中，是否需要知道每个进程中到底有多少玩家？一个game进程中可能有无数地图，这些地图上有无数玩家，而这些地图是分散在各个线程中的。因此，需要放置一个收集组件，让每个地图定时向这个收集组件发送当前地图的人数。同时，这个收集组件定时向appmgr进程发送它的统计数据。

在game进程中增加了新组件WorldProxyGather，就是用来做这件事情的。它的主要代码如下：

```
void WorldProxyGather::Awake() {
    AddTimer(0, 10, true, 2, BindFunP0(this, &WorldProxyGather::SyncGameInfo));
    ...
    pMsgCallback->RegisterFunction(Proto::MsgId::MI_WorldProxySync);
}

void WorldProxyGather::SyncGameInfo() {
    Proto::AppInfoSync proto;
    const int online = std::accumulate(_maps.begin(), _maps.end(), 0,
        [&](int value, const MapInfo& pair) {
            return value + pair.second.Online;
        });
    proto.set_app_id(Global::GetInstance()->GetCurAppId());
    proto.set_app_type((int)Global::GetInstance()->GetCurAppType());
    proto.set_online(online);
    MessageSystemHelp::SendPacket(Proto::MsgId::MI_AppInfoSync, proto);
}
```

在WorldProxyGather组件从对象池中被唤醒时，为它增加了一个每10秒调用的定时器，该定时器调用SyncGameInfo函数向appmgr进程发送数据。

在这个组件中，在线玩家的数据是保存在一个字典中的，这是由其他类通知WorldProxyGather而形成的，在WorldProxyGather组件中，它

关心一个名为MI_WorldProxySyncToGather的协议。在工程里，全局搜索这个协议，会发现它在WorldComponentGather组件中定时发送。

登录game进程，还没有进入具体地图的玩家汇聚在大厅中，在代码中即Lobby类。Lobby类是一个类似地图的类，它是无形的。这个类生成时被挂上了一个组件WorldComponentGather。该组件可以挂在任何一个需要发送在线数据的World类中，定时发送的协议如下：

```
void WorldComponentGather::SyncWorldInfoToGather() const {
    Proto::WorldProxySyncToGather proto;
    proto.set_world_sn(GetSN());
    proto.set_world_proxy_sn(GetSN());
    proto.set_world_id(dynamic_cast<IWorld*>(_parent)->GetWorldId(
    const int online = _parent->GetComponent<PlayerCollectorCompon
    proto.set_online(online);
    MessageSystemHelp::DispatchPacket(Proto::MsgId:: MI_WorldProxy
}
```

WorldComponentGather组件定时从其父类的PlayerCollectorComponent组件取出当前玩家的数量，发送MI_WorldInfoSyncToGather协议，并不关心谁会收到它，这个组件的责任就是组织这些数据并定时发送，任务简单明了。

无论我们有多少个game进程，每个进程上有多少个地图，只要地图实例上挂上了WorldComponentGather组件，都会将数据传递给本进程中的WorldProxyGather组件。而WorldProxyGather组件定时将这些数据发送到appmgr进程中，appmgr又定时将这些数据广播到所有login进程中，这样login就有了game进程的数据。图10-3展示了这一系列流程。无论我们登录时落在哪一个login进程上，都可以得到所有game进程的当前状况，从而选择一个最优的进程登录。

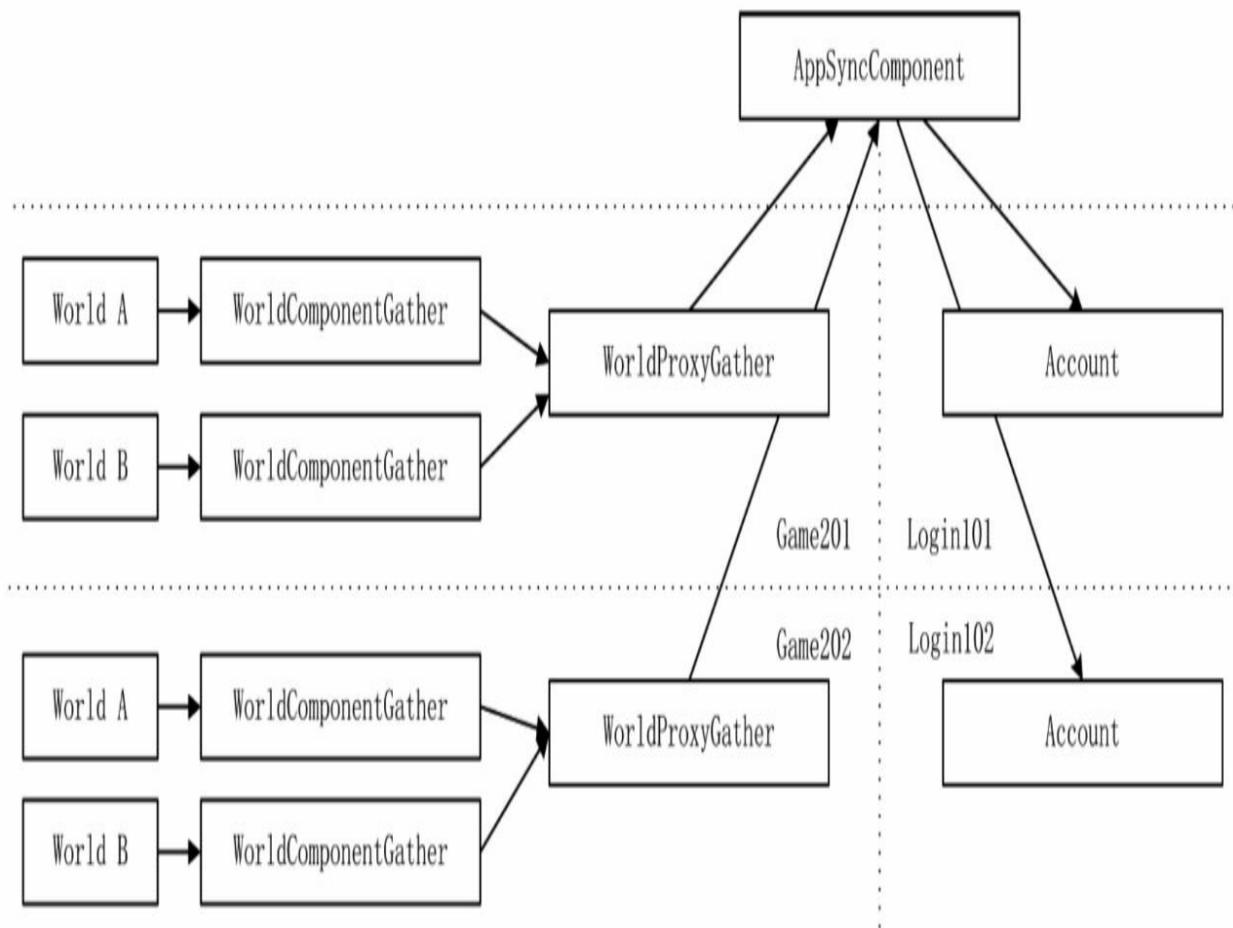


图10-3 采集数据流程

10.1.2 使用token登录game进程

当login进程中的Account类选择好可以登录的game进程时，它会发送一个L2C_GameToken协议到客户端。发送的代码如下：

```
void Account::HandleSelectPlayer(Packet* pPacket) {
    ...
    auto pPlayer = pPlayerMgr->GetPlayerBySocket (pPacket->GetSock
    ...
    Proto::GameToken protoToken;
    AppInfo info;
    if (!GetOneApp(APP_GAME, &info)) {
        protoToken.set_return_code(Proto::GameToken_ReturnCode_ Ga
    } else {
        protoToken.set_return_code(Proto::GameToken_ReturnCode_ Ga
        protoToken.set_ip(info.Ip.c_str());
        protoToken.set_port(info.Port);
        protoToken.set_token("");
    }
    MessageSystemHelp::SendPacket(Proto::MsgId::L2C_GameToken, pPl
}
```

这里编写了一行特别的代码，将token设置为空。有关token的生成将在10.2节讲解。当Robot收到协议之后，从协议中取出可以登录的地址，这里它会断开与login的通信，转而登录game进程。Robot会带上login进程发给它的token，登录game进程的协议为C2G_LoginByToken。而game进程的Lobby类中处理了该协议。

```
void Lobby::HandleLoginByToken(Packet* pPacket) {
    auto pPlayerCollector = GetComponent<PlayerCollectorComponent>
    auto proto = pPacket->ParseToProto<Proto::LoginByToken>();
    auto pPlayer = pPlayerCollector->AddPlayer(pPacket, proto.acco
    if (pPlayer == nullptr) {
        MessageSystemHelp::DispatchPacket(Proto::MsgId:: MI_Networ
        return;
    }
    LOG_DEBUG("enter game. account:" << proto.account().c_str());
    Proto::LoginByTokenRs protoLoginGameRs;
    protoLoginGameRs.set_return_code(Proto::LoginByTokenRs::LGRC_O
    MessageSystemHelp::SendPacket(Proto::MsgId::C2G_LoginByTokenRs
```

```
}
```

作为一个测试登录的工程，game进程收到这个协议之后进行了打印输出，没有实际做出正误判断，同时发出一个返回协议C2G_LoginByTokenRs，通知客户端已登录成功。关于Robot的状态以及状态的切换，这里我们就不再多说了，感兴趣的读者可以看一下源代码。

10.1.3 Player组件

在10_01_game目录的工程中新增了一个libplayer库，它的目的是将逻辑与底层框架分开。在这个libplayer库中，我们新建了几种数据便于以后使用，涉及地图的是IWorld类，涉及玩家的类有Player类以及玩家管理组件PlayerCollectorComponent。关于玩家的类都是Entity类，也就是说这些类可以加载无数个组件。以往在编写逻辑代码时，如果Player中有一个邮件系统信息，那么写法可能为“class Player {Mail_mail; }”，在Player中生成一个Mail实例或采用继承的关系“class Player: public Mail”来定义邮件数据。

现在要颠覆这个观点了，因为Entity类就是一个大容器。所谓容器，就是你给它什么，它就表现什么。一个空杯子装上蓝色的水就是蓝色的，装上红色的水就是红色的。

要实现邮件系统，可以编写一个MailComponent加载到Player类中，代码如下：

```
Player->AddComponent<MailComponent>();
```

使用时，用GetComponent函数取出：

```
auto pMail = Player->GetComponent<MailComponent>();
```

使用组件的方式，代码显得更加简洁，还有另一个很好的用途就是重用。当我们把一些原子功能编写到组件内时，这些组件就可以被不同的实体加载，以达到功能复用的目的。

在目录10_01_game工程的在login进程中，Account类有PlayerCollectorComponent组件管理玩家。在game工程中也给Lobby类加上了PlayerCollectorComponent组件。PlayerCollectorComponent组件主要是管理Player的类，而Player类是一个空杯子，我们给这个空杯子装上什么组件，它就有什么特性。因此，即使是相同的类，在login进程和game进程中的表现完全不同。下面查看一下

PlayerCollectorComponent的定义:

```
class PlayerCollectorComponent :public Component<PlayerCollectorCo
public:
    Player* AddPlayer(NetworkIdentify* pIdentify, std::string acco
    void RemovePlayerBySocket(SOCKET socket);
    Player* GetPlayerBySocket(SOCKET socket);
    Player* GetPlayerByAccount(std::string account);
    ...
private:
    // <socket, Player*>
    std::map<SOCKET, Player*> _players;
    // <account, socket>
    std::map<std::string, SOCKET> _accounts;
};
```

在该组件中，我们维护了一个Player列表，保存了Socket与Player的对应关系。在login进程中，Player维护的是一些登录信息，这是login进程中所特有的，所以为这样的需求编写了PlayerComponentAccount组件，加载到Player实体上。

同样是Player类，它在game进程中并没有PlayerComponentAccount组件，也不需要。

Redis及其第三方库

在10.1节中遗留了一个问题没有处理，就是token的生成以及验证方式。只有将这部分内容完成，整个登录流程才算真正完成了。生成token是在login进程中，而验证token却是在game进程中，如何让一个数据在两个进程之间共享，这是我们需要解决的问题。

为了解决这个问题，使用到了Redis内存数据库。现在有一个新的知识点——Redis，先来看看它是什么。

和MySQL不同，Redis是一个非关系型数据库，它是Key-Value数据库，由一系列的列表和字典构成，既可持久化保存，又可以只保存在内存中。Redis因为是内存数据库，性能比MySQL高一些，而Redis也提供锁机制，非常适合进程之间的数据交互。除了用在token上外，任何需要进程交互或暂存数据的功能都可以使用Redis。

10.2.1 Redis的安装

使用Redis，除了安装Redis本身之外，还需要安装hiredis库，hiredis库主要提供编码时操纵Redis的一些API函数。在Linux下安装命令如下：

```
yum -y install --enablerepo=epel redis
yum -y install --enablerepo=epel hiredis
yum -y install --enablerepo=epel hiredis-devel
```

Redis默认安装的配置文件位于/etc/redis.conf，使用vim打开该文件，默认的端口为6379。在Linux安装Redis还需要修改配置，找到bind 127.0.0.1改为bind 192.168.0.120 127.0.0.1，其中192.168.0.120是本地IP地址，修改成开启的虚拟机IP地址。这样，Redis既可以被外部访问，又可以被本地访问。外部访问时可以用于测试和查看数据。

启动Redis的命令如下：

```
[root@localhost ~]# systemctl start redis
```

设置为开机启动的命令如下：

```
[root@localhost ~]# systemctl enable redis
```

本书源代码的dependence目录中提供了一个已编译好的用于查看Redis数据的工具redis-desktop-manager。其GitHub主页为<https://github.com/uglide/RedisDesktopManager/releases>。

如果没有特别指定，Redis默认的数据库就指向select (0)，即db0。默认的Redis是没有账号和密码的，大部分情况下Redis在内部网络中使用，不需要设置账号和密码。

在本书工程中使用hiredis时，需要修改工程的CMakeLists.txt文件，在属性target_link_libraries上增加-lhiredis。

Windows系统要操作Redis，需要取得hiredis库的Windows版，其下载地址为<https://github.com/MicrosoftArchive/redis/releases>。本书源代码的dependence目录中提供了一个3.2.100版本。解压后进入redis-win-3.2.100\msvs目录，打开RedisServer.sln工程，编译Hiredis和Win32_Interop工程，需要编译出32位版本。在编译之前修改运行库的设置，设置为“多线程调试DLL (/MDd)”。编译完成后，将生成的hiredis.lib和Win32_Interop.lib放入源代码目录libs\windows中，以便于后面的工程调用。

除此之外，要在Windows系统中使用Redis，还需要做一些善后工作：

(1) 打开redis-win-3.2.100\deps目录，将hiredis目录复制到根目录include\windows下，成为include\windows\hiredis子目录。

(2) 在include\windows\hiredis目录下新建一个Win32_Interop目录，将位于redis-win-3.2.100\src\Win32_Interop目录下的文件win32_types_hiredis.h复制过来。

(3) 修改hiredis.h文件：

- 修改前：#include"../src/Win32_Interop/win32_types_hiredis.h"。
- 修改后：#include"Win32_Interop/win32_types_hiredis.h"。

这样做的目的是让头文件脱离原工程，如果是多人协作，不用每个人都下载Redis编译修改，只需要调用编译好的lib和include目录就可以马上使用。准备工作做完之后，还需要在使用Redis的工程中加入lib引用。目前使用Redis的工程为login和game工程。本节参考源代码位于10_02_redis目录中，工程已配置完成。

10.2.2 Redis命令行命令

在深入使用Redis之前，需要进一步了解什么是Redis。Redis是一个Key-Value数据类型数据库，数据存放在内存中，所以它的读取速度相当快，需要时，它的数据也可以像MySQL一样保存在硬盘上。在Linux中，我们可以使用redis-cli来连接Redis。下面具体来看Redis的一些常用命令。

```
[root@localhost ~]# redis-cli
127.0.0.1:6379>
```

首先执行redis-cli进入，因为没有修改默认端口，所以默认进入的本地端口为6379。使用--help来查看redis-cli命令：

```
-h <hostname>      Server hostname (default: 127.0.0.1).
-p <port>           Server port (default: 6379).
```

假如要连接的Redis服务不在本地，可以使用“redis-cli-h-p”来指定IP地址和端口。下面做一个简单的Redis命令测试。

关键命令1: set和get

```
[root@localhost ~]# redis-cli
127.0.0.1:6379>
```

进入redis-cli，当我们输入set之后，会有一个灰色的提示“set key value[EX seconds][PX milliseconds][NX|XX]”。我们可以如下使用set：

```
127.0.0.1:6379> set mykey "hello world"
OK
```

这里的set命令相当于给mykey一个一对一的值，我们给出的值是一个字符串“hello world”。然后使用get命令取出：

```
127.0.0.1:6379> get mykey  
"hello world"
```

关键命令2: setex

命令setex的格式为：setex key seconds value。它是set的扩展，为一个key赋值的同时设置了这个值的生存时间。这条命令非常有用：

```
127.0.0.1:6379> setex mykey2 3 token1  
OK
```

将字符串mykey2设置为token1，同时让它存活3秒。如果在3秒之内使用get命令，就会得到之前赋给它的值。

```
127.0.0.1:6379> get mykey2  
"token1"
```

如果在3秒之后获取，就返回空值。

```
127.0.0.1:6379> get mykey2  
(nil)
```

这个功能非常适合用来登录token。token一旦生成，为了安全起见，一段时间之后就需要销毁。而销毁的这个过程由Redis自行控制，就不需要我们关心了，非常省事。

关键命令3: setnx

命令setnx的格式与set一样：setnx key value，用于对一个给定的key赋值。它们的区别在哪里呢？setnx是SET if Not eXists的缩写，如果不存在，就设置，这是一个原子操作。如果key值已经存在，就返回出错信息。

```
127.0.0.1:6379> setnx key1 1
(integer) 1
127.0.0.1:6379> setnx key1 1
(integer) 0
```

第一次对key1赋值时，返回值为1表示正确；第二次赋值时，返回值为0表示出错。使用setnx命令主要是用在账号登录上，它就像是一个开关，当某个账号在线时，如果该账号再次登录，setnx就会出错。

关键命令4: [expire](#)

命令expire的格式为：expire key seconds，是对一个已存在的键（Key）设置生存周期。例如，我们登录时用setnx产生的键，就可以对它进行一个expire存活时间设置。

目前学习这几条基本的命令即可，其他的命令可以参见Redis官方网站。关于这些命令的具体使用，在后面的章节还会讲解到。

10.2.3 hireids库

在Linux和Windows系统上都使用了第三方库hiredis。这个库和Redis的关系就好像MySQL Connector/C库与MySQL的关系。hiredis库提供API函数对数据库进行访问。先了解一下它的几个重要函数：

关键函数1: redisConnectWithTimeout

作为了一个数据库，当然必不可少就是连接函数。hiredis库提供的连接函数为redisConnectWithTimeout，函数参数如表10-1所示，函数原型如下：

```
redisContext *redisConnectWithTimeout(const char *ip, int port, co
```

表10-1 redis连接函数参数

参 数 名	说 明
ip	数据库 IP
port	数据库端口
tv	超时 timeval 结构

结构timeval定义如下：

```
struct timeval {
    __time_t tv_sec;          /* Seconds. */
    __suseconds_t tv_usec;    /* Microseconds. */
};
```

tv_sec为秒数，tv_usec为微秒数。例如，struct timeval outTime={1, 500000}；，即为1.5秒，500 000微秒为0.5秒。

调用redisConnectWithTimeout函数之后，其返回值为redisContext指针，若为null，则表示不成功；若不为null，并且返回的redisContext中的err不为null，则也认为是不成功的，此时需要检查一下返回值中的出错标志。

关键函数2：释放连接函数

不再用Redis时，需要手动释放之前的连接实例，其原型如下：

```
void redisFree(redisContext *c);
```

参数redisContext的指针是创建时生成的指针。

关键函数3：命令执行函数

操作Redis数据时，一般情况下都是采用命令方式。执行Redis命令的函数为redisCommand，类似于MySQL的SQL语句的执行。其函数原型有几种模板：

```
void *redisCommand(redisContext *c, const char *format, va_list ap);  
void *redisCommand(redisContext *c, const char *format, ...);  
void *redisCommandArgv(redisContext *c, int argc, const char **arg
```

函数redisvCommand返回redisReply指针。向Redis查询数据，返回结果类都是redisReply指针，但想象一下，我们查询一个数据，有时候可能想得到一个字符串，有时候可能想得到一个数值，怎么来区分呢？redisReply类中提供一个名为type的属性，可以得到执行的结果的类型，返回类型定义如下：

```
#define REDIS_REPLY_STRING 1  
#define REDIS_REPLY_ARRAY 2  
#define REDIS_REPLY_INTEGER 3  
#define REDIS_REPLY_NIL 4  
#define REDIS_REPLY_STATUS 5  
#define REDIS_REPLY_ERROR 6
```

分别表示字符串、数据、数值、空、状态和出错几种类型。后面具体用到的时候，我们再深入讲解这些类型。

关键函数4：释放查询结果

通过redisCommand函数得到的结果指针redisReply不使用时需要释放，调用函数freeReplyObject释放。其原型如下：

```
void freeReplyObject(void *reply);
```

综上，基本常用的就是这几个函数。下面通过具体的编码来了解 hiredis库中的这些函数的使用。

10.2.4 组件RedisConnector

本节的源代码位于目录10_02_redis中。为了更好地使用Redis，为它编写了一个新组件RedisConnector，源代码位于libserver工程的redis_connector.h文件中，其定义如下：

```
class RedisConnector :public Entity<RedisConnector>, public IAwake
public:
    void Awake() override;
    bool Connect();
    bool Disconnect();
    ...
protected:
    bool Ping() const;
    bool Setex(std::string key, std::string value, int timeout) co
    bool SetnxExpire(std::string key, int value, int timeout) cons
    void Delete(std::string key) const;
    ...
};
```

组件RedisConnector从对象池中被唤醒时进行了数据库的连接，连接目的地配置在engine.yaml文件中，组件RedisConnector的初始代码如下：

```
void RedisConnector::Awake() {
    ...
    auto pYaml = ComponentHelp::GetYaml();
    const auto pConfig = pYaml->GetConfig(APP_DB_MGR);
    auto pDBConfig = dynamic_cast<DBMgrConfig*>(pConfig);
    auto pRedisConfig = pDBConfig->GetDBConfig(DBMgrConfig::DBType
    if (pRedisConfig == nullptr) {
        LOG_ERROR("Init failed. get redis config is failed.");
        return;
    }
    _ip = pRedisConfig->Ip;
    _port = pRedisConfig->Port;
    // time function
    AddTimer(0, 2 * 60, false, 0, BindFunP0(this, &RedisConnector:
    Connect());
}
```

初始化时指定了一个定时器，从绑定函数名可以看出进行了ping操作。该定时器每两分钟进行一次ping操作。在初始化时，连接数据库的函数如下：

```
bool RedisConnector::Connect() {
    const struct timeval outTime = { 1, 500000 }; // 1.5 seconds
    redisContext* c = redisConnectWithTimeout(_ip.c_str(), _port,
    if (c == nullptr || c->err) {
        if (c) {
            LOG_ERROR("RedisConnector::Connect. errno=" << c->err
            redisFree(c);
        } else {
            LOG_ERROR("RedisConnector error: can't allocate redis
        }
        return false;
    }
    // 选择(1)号数据库
    int db_index = 1;
    std::string sql = strutil::format("select %d", db_index);
    redisReply* pRedisReply = static_cast<redisReply*>(redisComman
    if (nullptr == pRedisReply) {
        LOG_ERROR("RedisConnector::Connect. errno=" << c->err << "
        redisFree(c);
        return false;
    }
    if (!(pRedisReply->type == REDIS_REPLY_STATUS && strcmp(pRedi
        LOG_ERROR("RedisConnector::Connect. errno=" << c->err << "
        freeReplyObject(pRedisReply);
        redisFree(c);
        return false;
    }
    freeReplyObject(pRedisReply);
    _pRedisContext = c;
    LOG_DEBUG("\tRedisConnector::Connect: successfully!");
    return Ping();
}
```

连接时使用了hiredis的redisConnectWithTimeout函数，成功连接之后做了一个ping调用：

```
bool RedisConnector::Ping() const {
```

```
if (_pRedisContext == nullptr)
    return false;
redisReply* pRedisReply = static_cast<redisReply*>(redisCommand(_pRedisContext, "ping"));
if (nullptr == pRedisReply) {
    LOG_ERROR("RedisConnector::ping: errno=" << _pRedisContext->errno);
    return false;
}
if (!(pRedisReply->type == REDIS_REPLY_STATUS && strcmp(pRedisReply->str, "PONG"))) {
    LOG_ERROR("RedisConnector::ping: errno=" << _pRedisContext->errno);
    freeReplyObject(pRedisReply);
    return false;
}
freeReplyObject(pRedisReply);
return true;
}
```

在上面的代码中，使用了hiredis中的redisCommand函数，该函数执行了一条命令。在redisCommand中使用的命令可以在Shell中使用。例如，上面使用的ping可以在Shell中使用，结果如下：

```
[root@localhost ~]# redis-cli
127.0.0.1:6379> ping
PONG
```

在调用了redisCommand(_pRedisContext, "PING")代码之后，返回redisReply指针。在RedisConnector::Ping函数中，判断返回值是否是一个PONG，非PONG值都认为是失败。如果ping失败了，就会在定时器中再次重新连接。

组件RedisConnector作为Redis的基础组件，另外还提供了一些常用函数，如Setex函数就是将setex命令进行了封装，更多细节不再一一细述。下面来看Redis在框架中的应用。

10.2.5 Redis在login中的应用

前面有一个问题没有处理。如果一个玩家用两个客户端登录同一个账号，如果有两login进程这两次登录既可能同时落在同一个login进程上，又可能落在两个login进程上。如果落在不同的login进程上，那么它们之间肯定只有一个能登录，我们如何来限制这种事情发生呢？

这就是本节要用Redis处理的问题，如何跨进程检查账号是否在线。

本节的源代码位于目录10_02_redis中。在10.2.4小节，介绍了在libserver库中编写了一个RedisConnector组件，现在派上了用场。在login的工程中创建一个RedisLogin组件，其定义在redis_login.h文件中，定义如下：

```
class RedisLogin : public RedisConnector {
private:
    void RegisterMsgFunction() override;

    void HandleLoginTokenToRedis(Packet* pPacket);
    void HandleAccountQueryOnline(Packet* pPacket);

    void HandleAccountSyncOnlineToRedis(Packet* pPacket);
    void HandleAccountDeleteOnlineToRedis(Packet* pPacket);
};
```

RedisLogin类继承自基础的RedisConnector类，有3个任务：第一个任务是验证账号是否在线；第二个任务是生成可以登录game进程的token；第三个任务是给在线账号生成在线标志。下面逐一进行说明。

关键点1：验证账号是否在线

当收到一个客户端验证账号的协议时，Account类会转发一个协议向Redis询问登录的账号是否有相同的账号在线，协议号为MI_AccountQueryOnlineToRedis。组件RedisLogin收到了这个协议，它是这样处理的：

```

void RedisLogin::HandleAccountQueryOnline(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::AccountQueryOnlineTo
    auto account = proto.account();
    Proto::AccountQueryOnlineToRedisRs protoRs;
    protoRs.set_account(account.c_str());
    protoRs.set_return_code(Proto::AccountQueryOnlineToRedisRs::SO

    // 是否正在登录
    if (!SetnxExpire(RedisKeyAccountOnlineLogin + proto.account(),
        protoRs.set_return_code(Proto::AccountQueryOnlineToRedisRs

    // Game是否在线
    if (GetInt(RedisKeyAccountOnlineGame + proto.account()) != 0)
        protoRs.set_return_code(Proto::AccountQueryOnlineToRedisRs

    MessageSystemHelp::DispatchPacket(Proto::MsgId:: MI_AccountQue
}

```

在这个函数中，向Redis数据库查询了两条数据：第一条是RedisKeyAccountOnlineLogin；第二条是RedisKeyAccountOnlineGame。从名字上可以看出，第一条是想知道这个账号是否在login进程上，第二条则是查看它是否在game进程上。看看如下两个定义：

```

#define RedisKeyAccountOnlineLogin  "engine::online::login::" //
#define RedisKeyAccountOnlineLoginTimeout  6 * 60
#define RedisKeyAccountOnlineGame  "engine::online::game::" //
#define RedisKeyAccountOnlineGameTimeout  6 * 60

```

使用这两条数据时采用了字典型（含键值对的数据，Key-Value Pair）的数据库。它们的格式都是Key：：账号名。每个账号不同，Redis数据库中保存的键也不同，例如以账号名为test的账号登录，则它在login进程中的在线标志位为engine：：online：：login：：test，也就是上面定义的RedisKeyAccountOnlineLogin+账号名，组成了字典中的键。

在查询过程中调用了SetnxExpire函数，用于查询当前账号是否在其他login上登录过。所有的login进程连接的是同一个Redis，如果登录过，那么该键（Key）必有值（Value）。函数SetnxExpire的实现如下：

```

bool RedisConnector::SetnxExpire(std::string key, const int value,
    if (_pRedisContext == nullptr) {
        LOG_WARN("RedisContext == nullptr. connect failed.");
        return false;
    }
    std::string command = strutil::format("SETNX %s %d", key.c_str
redisReply* pRedisReply = static_cast<redisReply*>(redisComman
if (pRedisReply->type != REDIS_REPLY_INTEGER || pRedisReply->i
    // 没有设置成功，可能已经有值，也可能Redis挂掉，都认为已经在线
    LOG_COLOR(LogColorPurple, "[SETNX] failed 1. command:" <<
    freeReplyObject(pRedisReply);
    return false;
}
freeReplyObject(pRedisReply);

// 成功之后，为这个值设置一个时间限制，在这个时间内不能重复登录
command = strutil::format("EXPIRE %s %d", key.c_str(), timeout
pRedisReply = static_cast<redisReply*>(redisCommand(_pRedisCon
if (pRedisReply->type != REDIS_REPLY_INTEGER || pRedisReply->i
    // 没有设置成功，可能有异常，可能Redis挂掉，都认为已经在线
    LOG_COLOR(LogColorPurple, "[EXPIRE] failed 2. command:" <<
    freeReplyObject(pRedisReply);
    return false;
}

freeReplyObject(pRedisReply);
return true;
}

```

在上面的函数中使用了setnx和expire两个命令，它们在前面的章节中都已经讲过。如果两个玩家在两个login进程上同时登录，他们同时触发setnx命令向同一个键（Key）写值（Value），其中有一个必定会失败。因为setnx是一个原子操作，它本身是带锁的，执行该命令必定是一个串行操作，如果第一个操作成功，再想要写入值，就必定会失败。

命令setnx执行成功之后，再调用expire函数给键加一个限时销毁的时长，此后游戏逻辑中不再对这个数据进行操作时，由Redis去分担一部分销毁工作。

函数SetnxExpire既完成了查询，又完成了设值。这样就可以保证同一账号不会在多个login上登录。

关键点2：生成token

当所有的验证选择角色完成之后，需要生成一个token，这个token需要在login与game进程中同时访问，所以将它保存在Redis中成为一个很好的选择。在Account类中，验证账号成功之后，将发起一个MI_LoginTokenToRedis协议，该协议是根据账号和所选角色生成一个token，该协议的处理函数如下：

```
void RedisLogin::HandleLoginTokenToRedis(Packet* pPacket) {
    auto protoToken = pPacket->ParseToProto<Proto::LoginTokenToRed
    auto account = protoToken.account();
    auto playerSn = protoToken.player_sn();
    auto token = Global::GetInstance()->GenerateUUID();
    // 将tokeninfo序列化为一个串，存入Redis
    Proto::TokenInfo protoInfo;
    protoInfo.set_token(token);
    protoInfo.set_player_sn(playerSn);
    std::string tokenString;
    protoInfo.SerializeToString(&tokenString);
    const std::string key = RedisKeyAccountTokey + account;
    const int timeoue = RedisKeyAccountTokeyTimeout;
    if (!Setex(key, tokenString, timeoue)) {
        token = "";
        LOG_ERROR("account:" << account.c_str() << ". failed to se
    }
    // 将生成的token返回给Account类
    Proto::LoginTokenToRedisRs protoRs;
    protoRs.set_account(account.c_str());
    protoRs.set_token(token.c_str());
    MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_LoginTokenT
}
```

在上面的代码中，生成token使用了Global::GenerateUUID，可以参见源代码。这里使用了两套方案：在Windows下使用的是GUID；在Linux下使用的是UUID。如果Linux环境下还没有安装UUID，那么需要安装UUID，只需执行命令行：yum install libuuid-devel。在上面的代码中生成token之后，调用Setex函数发送了一条setex命令。

该命令的键定义如下：

```
#define RedisKeyAccountTokey " engine::token::"  
#define RedisKeyAccountTokeyTimeout 2 * 60
```

对于一个名为test的账号，生成的token键是engine::token::test。而键的值是一个string。该string是由Proto::TokenInfo结构序列化而来的，其中包括token与角色Id。函数Setex实现如下：

```
bool RedisConnector::Setex(std::string key, std::string value, con  
    const std::string command = strutil::format("SETEX %s %d %s",  
    return Setex(command);  
}  
bool RedisConnector::Setex(std::string command) const {  
    redisReply* pRedisReply = static_cast<redisReply*> (redisComma  
    if (pRedisReply->type != REDIS_REPLY_STATUS || strncmp(pRedisR  
        LOG_COLOR(LogColorPurple, "[SETEX] failed. command:" << co  
        freeReplyObject(pRedisReply);  
        return false;  
    }  
    freeReplyObject(pRedisReply);  
    return true;  
}
```

命令setex自带一个有效时间，在存储token时，采用的有效时间为RedisKeyAccountTokeyTimeout定义的2分钟。

关键点3：写入在线标志

通过Redis，在登录时可以保证同一账号进行串行登录，但是它依然有一个潜在的风险，这个登录标记有一个时限。分析代码可以发现RedisKeyAccountOnlineLogin这个键在玩家要求进行账号验证之前就产生了，但是账号验证是一个不确定的事件，向外部发起一个HTTP请求。假设我们将RedisKeyAccountOnlineLogin键的超时设为10秒，10秒之后HTTP请求还没有返回，Redis会将这个键销毁。但真实情况可能是在第11秒HTTP返回结果，这中间有1秒的时间差，如果在这1秒内，该账号再次发起登录，就会产生异常。

因此，需要有一种机制，只要玩家还在login进程上，就定时不断地向Redis写入在线的标志。将这个功能交给一个新组件

PlayerComponentOnlineInLogin来完成。这个组件在Redis验证成功之后、HTTP验证开始之前被加载到Player对象上。

```
void Account::HandleAccountQueryOnlineToRedisRs(Packet* pPacket) {
    auto protoRs = pPacket->ParseToProto<Proto::AccountQueryOnline>()
    ...
    // 在线组件
    pPlayer->AddComponent<PlayerComponentOnlineInLogin> (pPlayer->
    // 验证账号，发起一个HTTP请求
    MessageSystemHelp::CreateConnect(NetworkType::HttpConnector, p
}
```

该组件被初始化时，向时间堆写入了每隔一段时间就要调用的函数。这个时间正好是Redis设置在线标志销毁的一半时间，即在Redis要销毁在线标志之前会重新写入在线标志，并将它的时间重置。

```
void PlayerComponentOnlineInLogin::Awake(std::string account) {
    _account = account;
    AddTimer(0, (int)(RedisKeyAccountOnlineLoginTimeout*0.5), true
}
void o::SetOnlineFlag() const {
    // 设置在线标志
    Proto::AccountSyncOnlineToRedis protoSync;
    protoSync.set_account(_account.c_str());
    MessageSystemHelp::DispatchPacket (Proto::MsgId::MI_AccountSyn
}
```

函数SetOnlineFlag的任务就是向Redis发送一条在线标志设置消息，而Redis组件收到这个消息之后调用了Setex函数。

```
void RedisLogin::HandleAccountSyncOnlineToRedis(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::AccountSyncOnlineToRedis>()
    const std::string key = RedisKeyAccountOnlineLogin + proto.account
    const std::string value = std::to_string(Global::GetInstance()
    Setex(key, value, RedisKeyAccountOnlineLoginTimeout);
}
```

函数Setex会将键存活时间进行重新设置。综上所述，只要账号还

在login进程中，Redis中的在线标志就会一直存在。基于此，如果同一账号再次登录，就会发现在Redis中这个在线标志已经存在，不能再次登录了。

关键点4：效果测试

打开log4_help.h文件，打开宏LOG_REDIS_OPEN。编译工程，我们用robots登录看一下数据。使用robots登录账号test，可以看到有如下的打印输出：

```
[DEBUG] -    @@ SETNX. key:SETNX engine::online::login::test 0
[DEBUG] -    @@ EXPIRE. key:EXPIRE engine::online::login::test 360
[DEBUG] -    @@ SETEX. key:SETEX engine::online::login::test 360 0
[DEBUG] - account:test. gen token:03f291ba-b401-4406-892b-941f6d7
[DEBUG] -    @@ DEL. key:del engine::online::login::test
[DEBUG] -    @@ DEL. key:del engine::token::test
[DEBUG] -    @@ SETEX. key:SETEX engine::online::game::test 360 1
```

打印信息的最后两步是由game进程产生的，后面再进行讨论。Redis的流程示意图如图10-4所示。

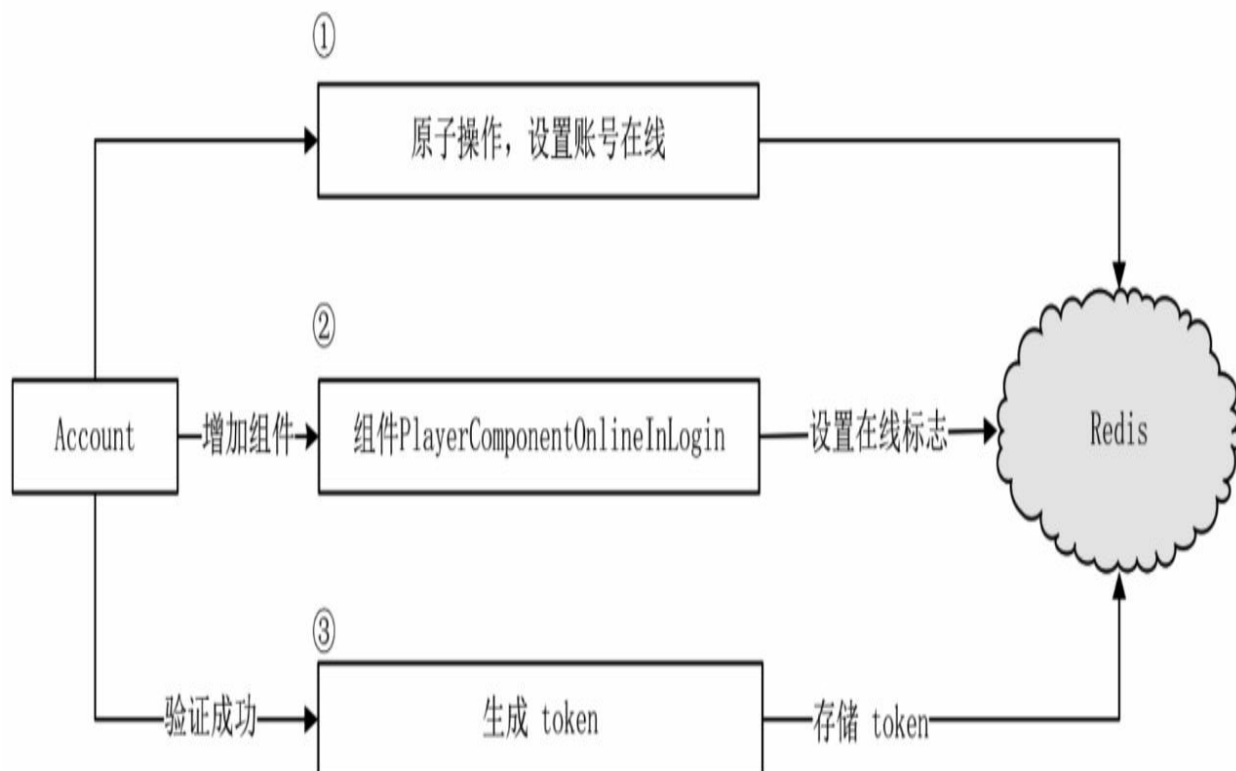


图10-4 Redis生成token

首先，向Redis执行了一个SETNX原子操作，使用EXPIRE为这个值在线标志设置了生命周期360秒。随着组件PlayerComponentOnlineInLogin的生成，为engine::online::login::test键重置了它的生命周期，随后，生成了一个token值。账号test登录game进程之后，组件PlayerComponentOnlineInLogin被销毁，导致了DEL.key: del engine::online::login::test的发生。

10.2.6 Redis在game中的应用

客户端登录game进程发起的第一个协议是C2G_LoginByToken，处理该协议的类为Lobby类。连接上game进程之后，就可以发起该协议，发送token以请求登录。怎么理解这个token呢？我们登录login时发送了账号和密码，这里的token也扮演着这样的角色。只是这个token的过程玩家并不知道。

在进程game中，为了处理请求token的协议增加了一个RedisGame类，这个类与之前的RedisLogin的原理是一致的。RedisGame有两个任务：第一个任务是查询已知的账号是否存在token；第二个任务是为当前在game进程中的玩家设置在线标志。

关键点1：查询一个已知账号的token

处理C2G_LoginByToken协议时，从客户端传来的token暂存到了PlayerComponentToken组件中，对于Lobby来说，它并不知道Redis数据在哪里，所以向系统广播了MI_GameTokenToRedis协议，向其他组件请求当前账号的token值。Lobby中对于C2G_LoginByToken的处理实现如下：

```
void Lobby::HandleLoginByToken(Packet* pPacket) {  
    ...  
    // 添加组件  
    pPlayer->AddComponent<PlayerComponentToken>(proto.token());  
    pPlayer->AddComponent<PlayerComponentOnlineInGame>(pPlayer->G  
    // 请求Token  
    Proto::GameTokenToRedis protoToken;  
    protoToken.set_account(pPlayer->GetAccount().c_str());  
    MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_GameTokenTo  
}
```

MI_GameTokenToRedis协议发出之后，是由RedisGame组件来处理的，实现如下：

```
void RedisGame::HandleGameTokenToRedis(Packet* pPacket) {
```

```

auto protoToken = pPacket->ParseToProto<Proto::GameTokenToRedis>
Proto::GameTokenToRedisRs protoRs;
protoRs.set_account(protoToken.account().c_str());
const std::string tokenValue = GetString(RedisKeyAccountTokey
protoRs.mutable_token_info()->ParseFromString(tokenValue);
Delete(RedisKeyAccountTokey + protoToken.account());
MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_GameTokenTo
}

```

在查询Redis时用到了GetString函数，token字段保存的是Proto：：TokenInfo结构的序列化字符串。

```

std::string RedisConnector::GetString(std::string key) const {
    std::string strRs = "";
    std::string command = strutil::format("get %s", key.c_str());
    redisReply* pRedisReply = static_cast<redisReply*>(redisComman
if (pRedisReply->type != REDIS_REPLY_STRING) {
    LOG_WARN("[GET failed] execute command:" << command.c_str(
} else {
    strRs = pRedisReply->str;
}
freeReplyObject(pRedisReply);
return strRs;
}

```

在上面的代码中使用了Redis的get命令，GetString是对这个命令的一个封装。如果插入的值是int，返回值就是REDIS_REPLY_INTEGER。之前插入的对象是序列化的一个字符串，所以这里检查一下返回值类型是不是REDIS_REPLY_STRING，得到字符串之后再反序列化。得到token数据之后，将数据设置到协议结构Proto：：GameTokenToRedisRs中，看一下msg.proto中关于该结构的定义：

```

message TokenInfo {
    string token = 1;
    uint64 player_sn = 2;
}
message GameTokenToRedisRs {
    string account = 1;
    TokenInfo token_info = 2;
}

```

在game进程向Redis查询协议的返回结构中包容了有一个TokenInfo结构。回到RedisGame::HandleGameTokenToRedis函数创建中来，得到TokenInfo之后注意，这里我们调用了一个Delete函数，删除了键（RedisKeyAccountTokey+protoToken.account（））。token只能使用一次，使用完成之后，将这个键对应的数据删除。

Lobby类接到RedisGame返回的MI_GameTokenToRedisRs协议，其处理函数为Lobby::HandleGameTokenToRedisRs，首先要做的就是验证玩家数据的合法性，前面我们已经反复强调了，每一步都是异步操作。当协议回到Lobby类中时，不确定这个玩家是否还在线，有可能已经断线了，所以需要进行一个账号的确认。确认之后，取出之前暂存的PlayerComponentToken组件，对两个token进行比较，如果出错，就返回给客户端一个出错编号；如果没有出错，就打印一句“enter game”。至此，玩家从login进程成功地登录game进程。

关键点2：设置在线标志

回看之前的代码，在login进程收到C2L_AccountCheck登录协议时，先向Redis发起了一个查询。这个查询分为两部分：一部分是检查账号是否在login进程中；另一部分是检查账号是否在game进程中。

当账号处于正在登录阶段时，login进程为每一个正在登录的Player对象绑定了一个组件PlayerComponentOnlineInLogin，该组件的任务是定时向Redis写入在线标志，标志该对象在login进程上。game进程中的在线标志设置原理和login进程中完全一致，其组件名为PlayerComponentOnlineInGame。只要玩家一直与game进程保持通信状态，组件PlayerComponentOnlineInGame就会定时向Redis中写入在线标志。

更多细节可以参见10_02_redis目录中的源代码。

10.2.7 从Redis删除数据

在login和game进程中有两个组件定时向Redis写入账号在线标志，当一个机器人类Robot退出时，可以看到类似“@@DEL key: del engine: : online: : game: : test”这样的打印信息。程序主动删除了Redis中的在线标志，那么这些标志是如何被销毁的呢？

在Lobby类中注册了一个函数来处理来自网络层的消息MI_NetworkDisconnect，该消息表示有一个Socket被关闭了，以Lobby为例的代码如下：

```
void Lobby::Awake() {  
    ...  
    auto pMsgSystem = GetSystemManager()->GetMessageSystem();  
    pMsgSystem->RegisterFunction(this, Proto::MsgId::MI_NetworkDis  
}
```

当一个玩家连接到game进程又断开了网络时，底层会分发MI_NetworkDisconnect协议，这正好是Lobby关心的协议之一，处理函数为HandleNetworkDisconnect。下面是Lobby类收到Socket断开时的处理方式：

```
void Lobby::HandleNetworkDisconnect(Packet* pPacket) {  
    GetComponent<PlayerCollectorComponent>()->RemovePlayerBySocket  
}  
void PlayerCollectorComponent::RemovePlayerBySocket(SOCKET socket)  
    const auto iter = _players.find(socket);  
    if (iter == _players.end())  
        return;  
    Player* pPlayer = iter->second;  
    _players.erase(socket);  
    _accounts.erase(pPlayer->GetAccount());  
    GetSystemManager()->GetEntitySystem()->RemoveComponent(pPlayer  
}
```

从代码来看，找到这个断开的Socket对应的玩家，然后将这个玩家从组件PlayerCollectorComponent中移除，以释放内存。那么它是如何

触发删除在线标志的呢？这里需要注意，实体销毁时，其附加的组件会同时全部销毁。

在RemovePlayerBySocket函数中，移除玩家时调用了GetEntitySystem::RemoveComponent函数，这个函数会将当前实体以及在实体上的所有组件移除，同时调用每个组件的BackToPool函数。而Player类的PlayerComponentOnlineInGame组件在销毁时发送了Redis数据销毁的协议，其实现如下：

```
void PlayerComponentOnlineInGame::BackToPool() {
    Proto::PlayerDeleteOnlineToRedis protoSync;
    protoSync.set_account(_account.c_str());
    protoSync.set_version(_onlineVersion);
    MessageSystemHelp::DispatchPacket(Proto::MsgId:: MI_PlayerDele
}
```

MI_PlayerDeleteOnlineToRedis协议的处理函数在RedisGame组件中。

```
void RedisGame::HandlePlayerDeleteOnlineToRedis(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::PlayerDeleteOnlineTo
    auto curValue = proto.version();
    const std::string key = RedisKeyAccountOnlineGame + proto.acco
    const auto onlineVersion = this->GetInt(key);
    if (curValue < onlineVersion)
        return;
    Delete(key);
}

void RedisConnector::Delete(std::string key) const {
    // Del返回值是被删除的数量
    std::string command = strutil::format("del %s", key.c_str());
    redisReply* pRedisReply = static_cast<redisReply*>(redisComman
    ...
    freeReplyObject(pRedisReply);
}
```

在删除过程中，调用基类函数RedisConnector::Delete删除了一个键。删除的时候，使用hiredis库的redisCommand函数执行了一个del命令。

这一系列的流程彻底地删除了玩家位于内存和Redis中的所有数据。在login进程中的在线标志也是一样的删除流程。

性能瓶颈分析

在目前的框架中，已经让玩家登录了game进程，生成了token，并做出了验证。前面的章节中增加了许多功能，加入了HTTP服务，加入了libplayer库，建立了game、space工程，等等。

本节缓和一下学习节奏，再来讨论一下性能。在10_01_game工程中，同时登录1000个账号，在Windows下的效率已经非常缓慢了，登录1000个账号至少花费了在Linux下的20倍时间。要么是我们在设计上有欠缺，要么就是有瓶颈产生。

如果在开发中产生了瓶颈，但又不知道瓶颈在哪里，那么可以使用本节介绍的通过查看日志来找出瓶颈的方法。

10.3.1 使用日志查看瓶颈

当我们面对一个性能瓶颈的时候，有时候即使使用了工具也找不到产生瓶颈的原因，更不知道该如何优化程序。这里提供一种思路，就是对现有框架的数据进行实时采集与分析，了解数据之后才能分析出问题的本源。

以10_01_game目录中的工程为例，在工程中准备了一个宏LOG_TRACE_COMPONENT_OPEN，用于跟踪Socket的日志。这个宏在libserver工程的trace_component.h文件中。这个宏在工程中默认是打开的。我们来看看它有什么作用？

在Windows系统下运行allinone进程，再批量登录500个账号。登录完成后，robots进程会出现“Login-SelectPlayer over”的提示。现在可以查看这些数据了，在robots控制台输入“trace-account test123”，表示想要看账号test123使用的所有socket值。

```
trace -account test123
socket: 884 1160
```

账号test123在整个登录过程中使用了两个Socket：一个是884；另一个是1160。我们可以分别看看两个Socket上的一些关键信息，输入“trace-connect socket”，可看到如下的信息：

```
trace -connect 884
key:884
...21:03:06.610 create connect != 0 waiting, err=10035 network ty
...21:03:06.610 create. network type:HttpConnector
...21:03:09.636 close. network type:HttpConnector
...21:03:09.636 create connect != 0 waiting, err=10035 network ty
...21:03:09.636 create. network type:TcpConnector
```

从上面的信息可知，在884这个Socket上使用了两次：一次是HTTP请求；另一次是TCP请求。当前这个TCP请求还没有断开，说明test123使用的可能是TCP。而HTTP请求时的Socket可能来自另一个账号。再来

看看在Socket值884上的一些状态：

```
trace -player 884
  key:884
...21:03:07.886  enter state:Http-Connecttd
...21:03:07.886  http connected. account:test55 socket:884
...21:03:08.905  enter state:Login-Connectting

...21:03:10.285  enter state:Login-Connecttd
...21:03:10.285  send check account. account:test123 socket:884
...21:03:12.710  enter state:Login-SelectPlayer
```

在前3行中，Socket值884是一个HTTP请求，它的关闭时间是09分，到了10分时，test123使用了值为884的Socket，两分钟之后，接收数据完成并进入了SelectPlayer状态。

trace这条命令在allinone进程上也可以使用，要查看trace命令的更多使用方法，可以执行命令“trace-help”。使用“trace-packet 884”可以看到值为884的Socket发送与接收过的所有协议，因为太长了，这里就不再一一列举出来了。

Trace命令只是用于辅助工作，我们不占用篇幅来详细说明它的实现，有兴趣的读者可以直接查看源代码。但是看过这些日志，分析一下性能消耗的原因，可能发生在网络层。为什么从连接成功到选择玩家协议中间花了两分钟？显然是不合理的。

以MI_NetworkConnected协议为例，这个协议的作用是当网络连接成功时向目标对象发送成功的消息，并附带SocketKey。现在我们有1000个Robot和两次连接（一次是HTTP连接，另一次是TCP连接），在很短的时间内发送了2000次MI_NetworkConnected协议。这个协议的注册函数使用的是过滤对象的方式，也就是说这2000个协议产生了 2000×1000 次函数对比。协议是无差别地发送到各个对象上的，对这个协议感兴趣的对象Robot类有1000个，一个协议要对比1000次才能找到自己真正的目的地，即使第一个对象调用GetPacketObject就找到了目标对象，但是后面的999次调用依然会产生，并没有机制让Packet找到了目标对象之后就退出分发这个流程。因此，有 2000×1000 次函数调用。而且每个Packet都会对所处线程中的对象询问是否对自己的协议号感兴趣，轮询的方式比较耗时。

通过上面的分析，相信读者已经发现了问题所在。显然Robot类作为注册处理协议的实体，这个设计思路是不行的。要解决这个问题有两个办法：第一个办法是在Robot类上再增加RobotCollection类，用于在线程中管理Robot，由RobotCollection类来处理协议，这时协议到来时其命中率将大大增加，只需要调用一次GetPacketObject就能够找到目标对象；第二个办法是做一个类似NetworkLocation的定位组件，以达到一步找到目标对象的目的。

本书中反复提到了性能测试的重要性。我们在编码的时候常常忽视这一环节，并不是功能效果达到了，功能就做完了。

本节的参考源代码目录为10_02_redis。在robots工程中，每个线程增加了一个RobotCollection组件。RobotCollection管理着本线程中所有的Robot对象，而将协议处理提到了RobotCollection层面之后，有几个线程就对比几次，调用RobotCollection::GetMsgObj函数一次就可以找到Robot对象。

虽然增加了新组件，相关协议处理函数从Robot迁移到了RobotCollection类，但是robots工程逻辑并没有改变。

10.3.2 优化MessageComponent组件

我们把目光放在底层框架上也会发现问题，首当其冲的就是MessageComponent组件。MessageComponent组件维护了一个字典，关联着MsgId和它的处理函数的对应关系。审视一下这个组件，我们可以发现，如果有1000个实体都关心网络断开事件，在MessageSystem中就有1000个MessageComponent，在这1000个MessageComponent组件中存放着这1000个实体对应网络断开事件的回调函数。图10-5的左图展示了这个流程中的内存数据。

MessageComponent组件阻隔了实体与MessageSystem之间的连接，它们彼此不知道对方的存在，但是MessageComponent带来了性能上的问题，而且它显然是多余的。鉴于该组件的操作频次，在MessageSystem系统中将MessageComponent取消，协议号MsgId直接注册到MessageSystem，这样做的好处一目了然，如图10-5的右图所示。对于一个确定的协议来说，不再需要遍历所有的MessageComponent组件就可以拿到所有需要回调的处理函数。本节的参考源代码目录为10_02_redis。MessageSystem的新定义如下：

```
class MessageSystem :virtual public ISystem<MessageSystem> {
public:
    void RegisterFunction(IEntity* obj, int msgId, MsgCallbackFun
        template<typename T>
    void RegisterFunctionFilter(IEntity* obj, int msgId, std::func
        ...
private:
    // message
    std::map<int, std::list<IMessageCallBack*>> _callbacks;
};
```

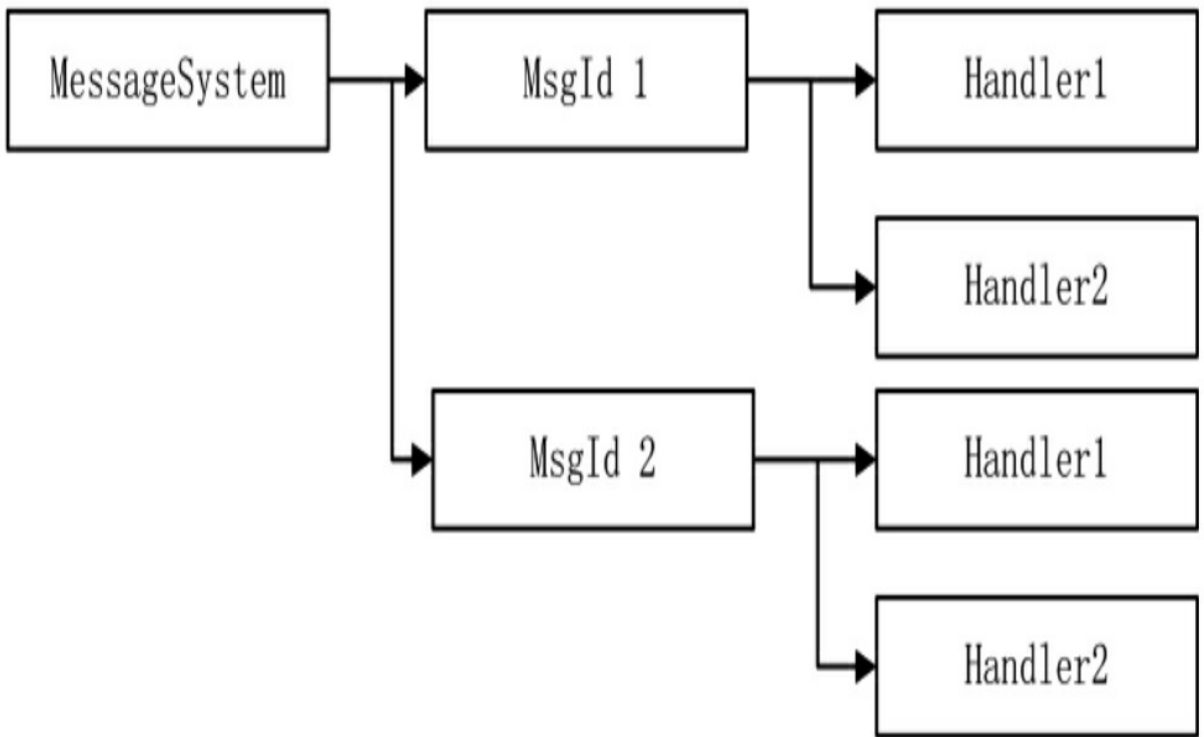
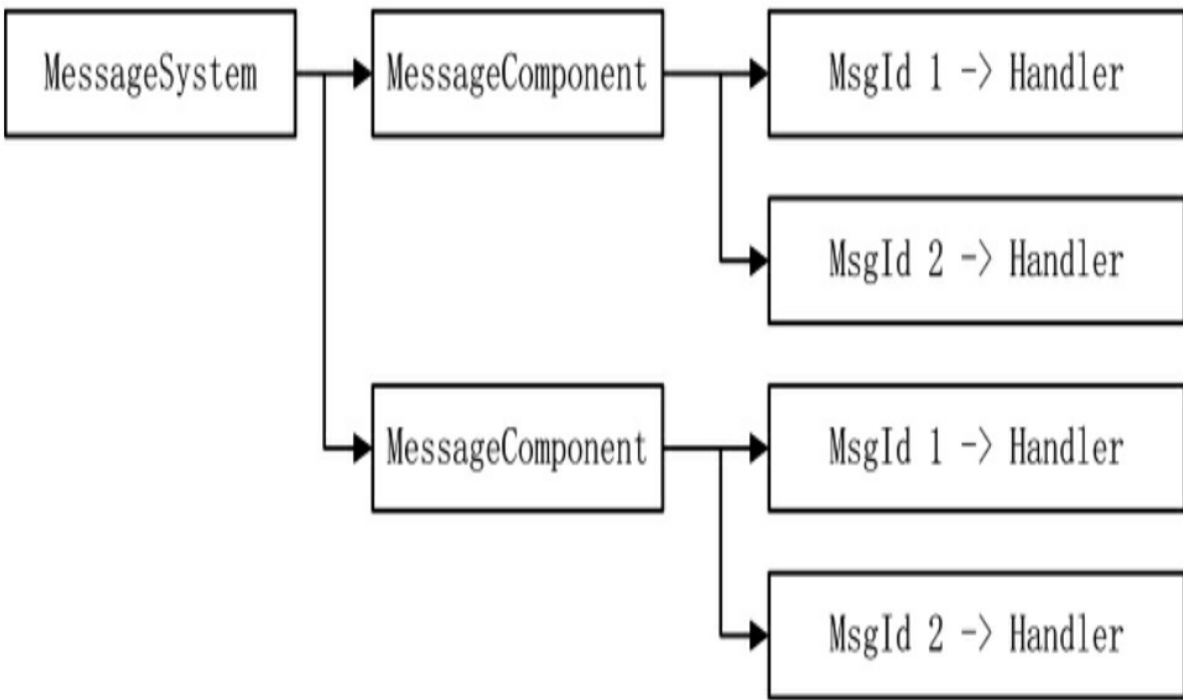


图10-5 MessageSystem性能优化

在注册协议函数时，将协议回调函数与MsgId绑定起来，这样免去了中间环节。以Account类为例，新的协议注册方式如下：

```
void Account::Awake() {  
    ...  
    auto pMsgSystem = GetSystemManager()->GetMessageSystem();  
    pMsgSystem->RegisterFunction(this, Proto::MsgId::C2L_AccountCh  
    ...  
}
```

Account类需要注册协议时，不再为自己增加一个MessageComponent组件，而是直接调用MessageSystem::RegisterFunction进行注册，该函数的实现如下：

```
void MessageSystem::RegisterFunction(IEntity* obj, int msgId, MsgC  
    auto iter = _callbacks.find(msgId);  
    if (iter == _callbacks.end()) {  
        _callbacks.insert(std::make_pair(msgId, std::list<IMessage  
    }  
    const auto pCallback = obj->AddComponent<MessageCallBack> (std  
        _callbacks[msgId].push_back(pCallback);  
}
```

取消MessageComponent组件之后，在MessageSystem中提供了两个函数：一个是RegisterFunction；另一个是RegisterFunctionFilter。下面来看一下注册函数：

```
template <typename T>  
void MessageSystem::RegisterFunctionFilter(IEntity* obj, int msgId  
    auto iter = _callbacks.find(msgId);  
    if (iter == _callbacks.end()) {  
        _callbacks.insert(std::make_pair(msgId, std::list<IMessage  
    }  
    auto pCallback = obj->AddComponent<MessageCallBackFilter<T>>()  
    pCallback->GetFilterObj = std::move(getObj);  
    pCallback->HandleFunction = std::move(fun);
```

```
        _callbacks[msgId].push_back(pCallback);  
    }
```

在RegisterFunctionFilter函数中用了过滤的流程，注册协议时需要提供一个过滤函数GetFilterObj。在robots工程的代码中，在为Robot类绑定处理函数时使用了RegisterFunctionFilter函数，其注册代码如下：

```
void RobotCollection::Awake() {  
    auto pMsgSystem = GetSystemManager()->GetMessageSystem();  
    pMsgSystem->RegisterFunctionFilter<Robot>(this, Proto::MsgId:::  
        ...  
}
```

收到MI_NetworkConnected网络连接协议时，RobotCollection::GetMsgObj函数根据Packet实例上的网络身份筛选出目标Robot类。

```
Robot* RobotCollection::GetMsgObj(NetworkIdentify* pIdentify) {  
    auto objKey = pIdentify->GetObjectKey();  
    if (objKey.KeyType != ObjectKeyType::Account)  
        return nullptr;  
    auto iter = _robots.find(objKey.KeyValue.KeyStr);  
    if (iter == _robots.end())  
        return nullptr;  
    return iter->second;  
}
```

对于每一个协议，GetMsgObj函数在线程上只需要调用一次就可以精准找到Robot实例，从而极大地提高了性能。

10.3.3 ConnectObj内存组织

如果将10_01_game目录下的工程在Windows和Linux下进行对比，就会发现在Windows系统下远远慢于在Linux系统下，接近慢了10倍。分析一下代码，除了网络层之外，几乎所有的代码都是共用的。怎么网络底层中采用的Select模型会慢这么多？

在robots工程中，随着功能的增加经历了3次网络连接，第一次是HTTP请求，第二次是连接到login，第三次是登录game进程。网络层的负担加重之后，它的性能问题就凸显出来了。下面来看之前的select函数中的代码：

```
void Network::Select() {
    for (auto iter = _connects.begin(); iter != _connects.end(); ++
        ...
    }
    ...
    const int nfds = ::select(_fdMax + 1, &readfds, &writefds, &ex
    if (nfds <= 0)
        return;
    auto iter = _connects.begin();
    while (iter != _connects.end()) {
        ...
    }
}
```

在Select函数调用中有两次循环：第一次是将所有的连接对象加入底层函数：`select`需要的集合中；第二次是分析底层函数：`select`的返回。这两个循环都不可缺少，这两个循环是Linux下`epoll`函数不需要的，这可能是Select模式变慢的原因。

在旧的Network类中，将连接对象放在一个字典中，代码如下：

```
std::map<SOCKET, ConnectObj*> _connects
```

现在来优化这个数据，使用如下数据结构代替：

```
#define MAX_CLIENT 10000
ConnectObj* _connects[MAX_CLIENT]{};
std::set<SOCKET> _sockets;
```

sockets集合中存放着当前所有有效的Socket值，_connects组件以数据的方式提供ConnectObj对象的访问。也就是说，不再使用字典，而是使用数组的方式来访问连接对象。考虑一种情况，我们登录1000个账号，按巅峰值来计算，最多时ConnectObj只有1000个，将字典换成数组，能大幅度提高整体的性能吗？

框架是按帧运行的，也就是说某些函数是每一帧都要运行的。运行完一帧用的时间越少，其性能就越高。下面用数据来说话，先来看图10-6中新数据的内存图。

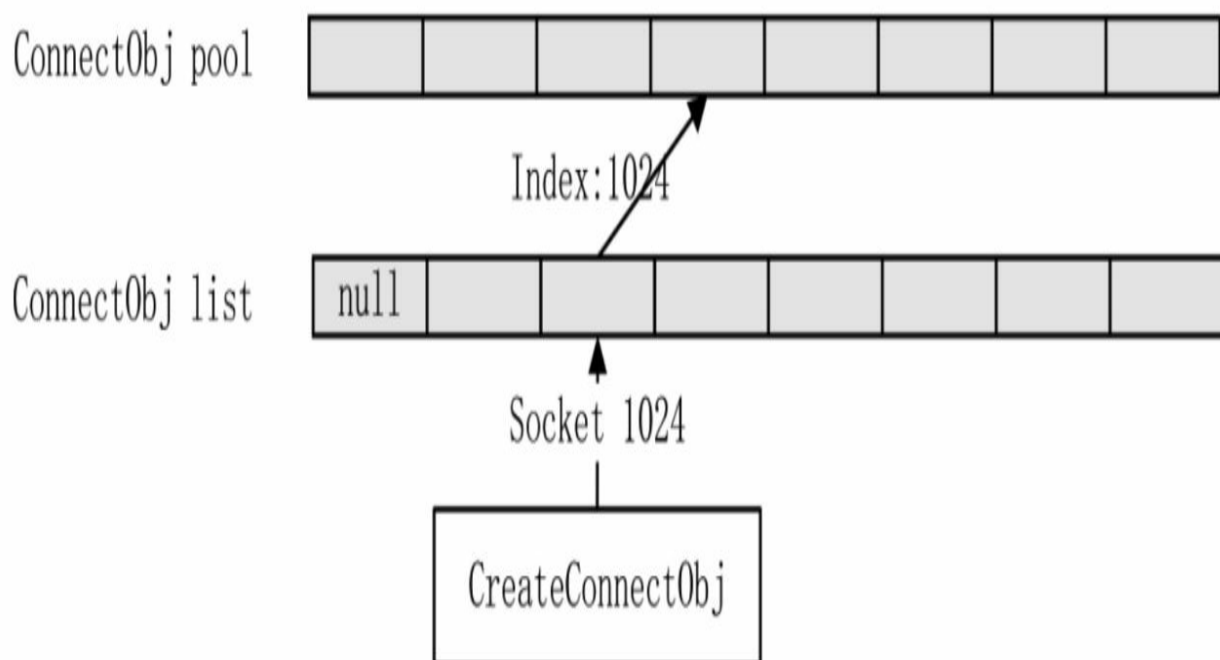


图10-6 ConnectObj内存列表

connects数组在初始化时都是空指针，创建Socket时将Socket值作为它的下标，因为在Windows下Socket分配值是随机的，所以使用了10000长度的数组。将Socket值作为下标之后，依然从对象池中分配一个对象，同时在connects数组对应的位置设置指针地址，以达到内存地

址一次寻址就能找到对象，而不再需要Find操作，从而提升效率。在编码中，遇到类似的问题都可以采用这种处理方式。

这个优化在10_02_redis目录中完成。现在可以比较一下10_01_game目录中工程的性能和10_02_redis目录中工程的性能，同时登录1000个账号，会发现不论是在Windows还是在Linux下都有一定的提升。这两个工程的功能完全一样，只是在数据结构上做了一些优化工作。

多进程登录协议回顾

截至目前的工程，我们介绍了如何从众多login中选择一个进程进行账号验证，同时阐述了如何从众多game中选择一个合适的进程进行登录的方案。

将所有细节抛到一边，先从协议入手，梳理一下最终的登录方案需要多少个协议。协议定义文件位于10_02_redis目录的proto_id.proto文件中。要完成角色登录，需要经历以下几个阶段：

第一个阶段：登录阶段

C2L_AccountCheck	= 1000;	// 1.验证账号
C2L_AccountCheckRs	= 1001;	
MI_AccountQueryOnlineToRedis	= 1002;	// 2.验证账号在
MI_AccountQueryOnlineToRedisRs	= 1003;	

在这个阶段中，客户端只需要发送C2L_AccountCheck协议验证账号，得到C2L_AccountCheckRs协议的返回，而服务器要做的事情却不是那么简单。

(1) 服务器需要判断当前的账号是否已经在本进程或其他的login中登录了，这里涉及一个新问题——跨线程的数据查询。为了方便描述，我们采用Redis这个内存数据库。

(2) 服务器向Redis查询该账号是否已经登录，如果已在其他地方登录，就通知客户端；否则，进行HTTP的账号验证。

第二个阶段：角色创建与选择阶段

L2DB_QueryPlayerList	= 1010;	// 1.选择角色
L2DB_QueryPlayerListRs	= 1011;	
L2C_PlayerList	= 1012;	
C2L_CreatePlayer	= 1014;	// 2.选择角色：如果没有角色
C2L_CreatePlayerRs	= 1015;	
L2DB_CreatePlayer	= 1016;	// 3.选择角色：创建角色
L2DB_CreatePlayerRs	= 1017;	
C2L_SelectPlayer	= 1018;	// 4.选择角色
C2L_SelectPlayerRs	= 1019;	

当验证成功之后，进入第二个阶段，这个阶段的主要目的是创建与选择角色，这个阶段有与数据库的交互。

(1) 所有的登录验证都完成，login向dbmgr进程发起对账号所有角色的查询，将数据通过L2C_PlayerList协议传递到客户端。

(2) 如果没有角色，客户端就会发起C2L_CreatePlayer创建角色，协议C2L_CreatePlayer中带上创建的一些指定的属性，例如名字、性别等。

(3) 如果有角色，客户端就会发起C2L_SelectPlayer选择一个角色。

第三个阶段：为跳转进程做准备

MI_LoginTokenToRedis	= 1022;	// 1.请求登录的token
MI_LoginTokenToRedisRs	= 1023;	
L2C_GameToken	= 1024;	// 2.将token发送给客户端

选择好角色之后，就可以进入真正的游戏世界了，在此之前需要做一些准备工作。

(1) 当所有验证都成功时，login进程需要向Redis请求一个token用于登录指定game进程，这个token将在几秒之中消失。

(2) 将token发送到客户端，至此，login的工作全部完成。

以上3个阶段是在login进程中完成的，整个login进程主要负责账号与登录部分的功能。不要以为这部分的代码很简单，真正到了上线时，由于游戏需要接入的平台很多，这部分的代码也是非常复杂的。

第四个阶段：登录game

```
C2G_LoginByToken      = 1100;           // 1.登录：客户端请求登录game
C2G_LoginByTokenRs    = 1101;
MI_GameTokenToRedis   = 1102;           // 2.登录：game请求登录的token
MI_GameTokenToRedisRs = 1103;
```

(1) 客户端收到可以登录的game进程的信息之后，将token发送到game进程上要求登录，会收到C2G_LoginByTokenRs消息，如果失败，就可能是token失效，客户端需要从login进程再次登录。

(2) game进程收到客户端要求登录的协议C2G_LoginByToken之后，查询这个token是否是有效的，如果有效，就成功进入地图。

至此，整个登录流程就完成了。

10.5

总结

本章引入了Redis，完整实现了登录的整个过程，同时完善了框架中的进程，加入了game进程和space进程，还引入了libplayer库，加入了Player基本组件，使用Redis保存token，完成了token的验证。

到本章为止，基础组件已经搭建得较为完善，第11章将进入真正的游戏世界。

第11章

分布式跳转方案

本章主要讲解玩家在game进程上跳转地图的流程。在这个过程中，可以看到Player组件是如何读取数据的，进一步阐述game进程与space进程是如何通信的，同时在完成这一系列功能之后，将使用客户端进行登录查看。本章包括以下内容：

- 游戏逻辑配置文件读取。
- 进入地图流程，介绍跳转地图是如何实现的。
- 引入一个Unity客户端。

资源数据配置与读取

框架基本创建完成之后，本章会在其基础之上进行扩展。如果在之前的章节中还没有完全了解这个框架的精髓，那么在本章这些扩展中可以更加深入地了解框架是如何工作的。现在开始第一个扩展：读取资源。本节工程的源代码目录为11_01_resource。为了便于读写，示例中采用CSV数据格式。这种数据格式以文本形式存储表格数据，可以使用Excel打开。在工程中有一个编写好的world.csv格式的配置，使用文本编辑器打开它，其内容如下：

```
Id,Name,Init,Type,AbPath,ResName,UiResType,PlayerInitPos
1,login,0,1,scenes/login,Login,1,-1
2,lobby,0,2,scenes/roles,Roles,2,-1
...
```

可以看到，CSV格式的数据是纯文件的，列与列之间用半角的逗号“，”分隔开，有时也可能用半角的分号“；”进行分隔。在本书的示例中，规定使用“，”。在客户端源代码目录有Editor查看器，可以更加直观地看到这些数据的意义。如图11-1所示，在编辑器中展示了一些枚举数据的实际意义。

表格 ▾						
ID	名字	初始地图	类型	AB包路径	资源名	初始UI
1	login	☐	登录	scenes/login	Login	登录界面
2	roles	☐	角色选择	scenes/roles	Roles	选择角色界面
3	jiangxia	☒	公共地图	scenes/jiangxia	jiangxia	无
4	dungeon02	☐	副本地图	scenes/jiangxia	jiangxia	无

图11-1 资源配置表格

这些数据文件是按需求组织起来的，需要什么数据就加什么数据。需要说明的是，并不是所有的游戏逻辑配置文件都是用的CSV格式。也有人喜欢使用XML格式或其他格式，只是本书框架中是以CSV格式来举例的，这种逻辑配置是与游戏逻辑结合起来的。下面我们来看看在服务端如何解析这些数据。

配置文件并不是每个服务进程都需要读取，我们将它写成一个独立的库工程，名为libresource，与libserver在同一目录。

首先需要有一个总的管理类，管理所有游戏逻辑的配置文件，名为ResourceManager。在这个类下可能存在ResourceWorldMgr和ResourceItemMgr，分别管理地图和道具的相关配置，其逻辑结构如图11-2所示。对于ResourceWorldMgr来说，它有一个地图相关的列表需要维护。例如，ID=1时，对应一个ResourceWorld实例；ID=2时，对应另一个ResourceWorld实例。ResourceWorld就是我们之前看到的CSV文件中的某一行。理解了 this 逻辑，下面来详细说明代码是如何实现的。

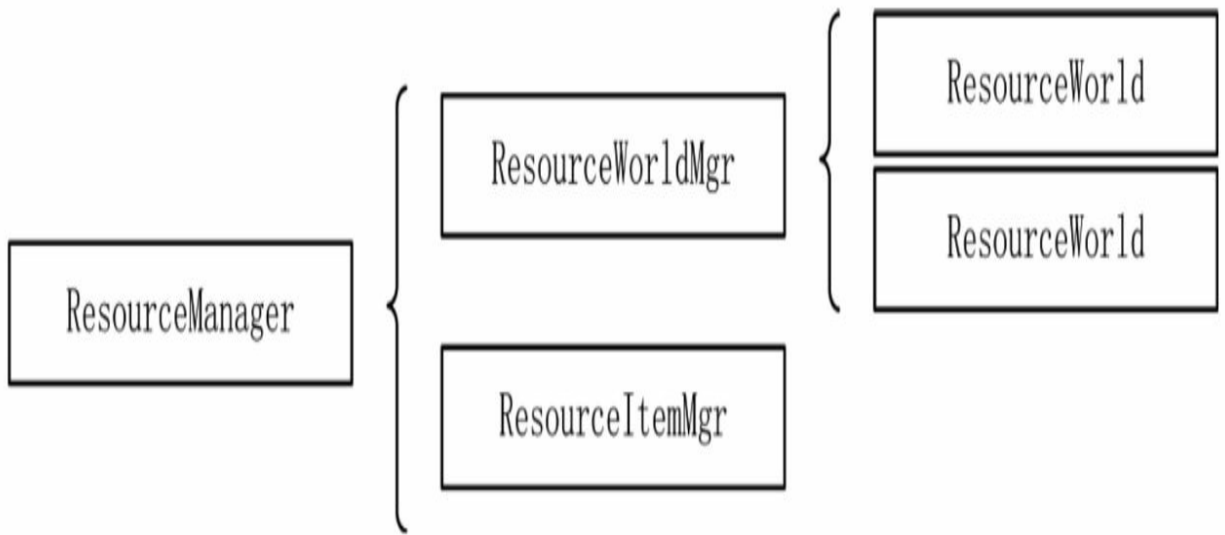


图11-2 逻辑结构

11.1.1 资源管理类ResourceManager

先从调用方式讲起，假如需要读取某个资源配置文件，简单的调用方式如下：

```
GetResMgr()->WorldMgr->GetWorld(id)
GetResMgr()->ItemMgr->GetItem(id)
```

先取得一个总的入口，从入口取得需要的地图或道具的管理器，再从管理器中取得想要的一行数据。定义ResourceManager类来管理所有资源配置，其定义如下：

```
class ResourceManager :public Entity<ResourceManager>, public IAwake
public:
    void Awake() override;
    void BackToPool() override;
public:
    ResourceWorldMgr* Worlds;
};
```

从定义可以看出，ResourceManager类是一个Entity实体，这会引出一个问题，这个实体应该被加载在哪个进程中？考虑到所有进程都可能读取这个实体，将它加载到主线程中比较合适。

在主线程中也有一个EntitySystem，我们将这个实例命名为GlobalEntitySystem，其作用是保存那些在多线程中共用的实体或组件。例如ResPath类、Log4类，还有现在讨论的ResourceManager这些线程安全的组件。服务端一旦开启，ResourceManager初始化完成，数据就只读不写，对于只读的数据，在任何线程中都是安全的。以下是ResourceManager初始化的代码实现：

```
void ResourceManager::Awake() {
    const auto pResPath = ComponentHelp::GetResPath();
    Worlds = new ResourceWorldMgr();
    if (!Worlds->Initialize("world", pResPath)) {
        LOG_ERROR("world txt Initialize. failed.");
    }
}
```

```
    }  
    LOG_COLOR(LogColorYellowEx, "all resource loaded.");  
}
```

在ResourceManager类中维护了一个指向ResourceWorldMgr类的指针，这是一个地图管理类。初始化时，对地图管理类进行了初始加载，指定了它的加载目录与加载文件。函数Worlds::Initialize的第一个参数world就是它的加载文件，也就是说要在资源目录中找到world.csv文件，因为后缀都是.csv，所以这里省略了后缀。

11.1.2 地图资源管理类ResourceWorldMgr

地图资源管理类的定义如下：

```
class ResourceWorldMgr :public ResourceManagerTemplate<ResourceWor
public:
    bool AfterInit() override;
    ResourceWorld* GetInitMap();
private:
    int _initMapId{ 0 };
};
```

ResourceWorldMgr类继承自一个模板类，模板类的定义在resource_mgr_template.h文件中。除了地图管理之外，随着逻辑功能越来越多会陆续加入很多其他的配置，例如道具、任务等，这些资源配置管理类其实都是相似的，所以编写了一个管理模板类。

```
template<class T>
class ResourceManagerTemplate {
public:
    bool Initialize(std::string table, ResPath* pResPath);
    virtual bool AfterInit( ) { return true; }
    T* GetResource(int id);

protected:
    bool ParserHead(std::string line);
    bool LoadReference(std::string line);
protected:
    std::string _cvsName;
    std::map<std::string, int> _head;
    std::map<int, T*> _refs;
};
```

每一个管理模板的实现都有一些共有的特性：一个是CSV文件格式；另一个是对于表格中表头的处理。ResourceManagerTemplate类相当于一个表基类，我们先来看模板类中的函数。

(1) Initialize函数：配置文件初始化数据。

(2) AfterInit函数：初始化完成之后的操作由具体的继承类实现。

(3) GetResource函数：根据Id取得一个对象实体。

(4) ParserHead函数：分析CSV文件的头。

(5) LoadReference函数：分析CSV文件的行。

在模板类中存储了一个字典数据，用于Id与行实例的一一对应：

```
std::map<int, T*> _refs;
```

其中，map的键（key）是行的Id。在world配置文件中，每一行的开头都有一个Id，这样设计数据可以很方便地读取某个Id的配置对象。在初始化函数时，CSV文件被一行一行地分析，变成一个一个的实例加载到内存中，其代码如下：

```
template <class T>
bool ResourceManagerTemplate<T>::Initialize(std::string table, Res
_cvsName = table;
std::string path = pResPath->FindResPath("/resource");
path = strutil::format("%s/%s.csv", path.c_str(), table.c_str(
std::ifstream reader(path.c_str(), std::ios::in);
if (!reader) {
    LOG_ERROR("can't open file. " << path.c_str( ));
    return false;
}
if (reader.eof()) {
    LOG_ERROR("read head failed. stream is eof.");
    return false;
}
// 分析第一行，头部标题
std::string line;
std::getline(reader, line);
std::transform(line.begin(), line.end(), line.begin(), ::tolower
if (!ParserHead(line)) {
    LOG_ERROR("parse head failed. " << path.c_str( ));
    return false;
}
// 循环读出每一行，将每一行转换为一个内存对象
```

```

while (true) {
    if (reader.eof())
        break;
    std::getline(reader, line);
    if (line.empty())
        continue;
    std::transform(line.begin(), line.end(), line.begin(), ::t
LoadReference(line);
}
if (!AfterInit())
    return false;
return true;
}

```

在分析加载文件的过程中，需要注意ParserHead和LoadReference两个函数，ParserHead函数用于分析文件的第一行，也就是标题行，而LoadReference函数用于分析行数据。ParserHead函数的实现代码如下：

```

template <class T>
bool ResourceManagerTemplate<T>::ParserHead(std::string line) {
    if (line.empty())
        return false;
    std::vector<std::string> propertyList = ResourceBase::ParserLi
for (size_t i = 0; i < propertyList.size(); i++) {
    _head.insert(std::make_pair(propertyList[i], i));
}
return true;
}

```

静态函数ResourceBase::ParserLine将标题数据变成vector<string>数据。表头数据保存到内存中，变成一个字典<std::string, int>，这个字典的键（Key）为标题，值（Value）为当前标题所在的index索引，表示这个标题位于表的第几列。以resource目录下的world.csv文件为例，其标题依次为：Id, Name, Init, Type, AbPath, ResName, UiResType, PlayerInitPos。头字典中的数据则是：{{Id, 0}, {Name, 1}, {Init, 2}, {Type, 3}, ...}。这样的结构便于通过名字来找到索引列。

每一行都有一个实例与之对应，函数LoadReference的实现如下：

```
template <class T>
bool ResourceManagerTemplate<T>::LoadReference(std::string line) {
    auto pT = new T(_head);
    if (pT->LoadProperty(line) && pT->Check()) {
        _refs.insert(std::make_pair(pT->GetId(), pT));
        return true;
    }
    return false;
}
```

在模板函数中，T类型一定是基于ResourceBase类的。加载数据函数，读到某一行，将行数据的处理转交给ResourceBase::LoadProperty函数来处理。下面以地图资源类ResourceWorld来说明这一过程。

11.1.3 地图资源类ResourceWorld

基类ResourceBase的主要任务是完成一行的加载以及提供一些基础函数供子类调用。该类定义在resource_base.h文件中，定义如下：

```
class ResourceBase {
public:
    explicit ResourceBase(std::map<std::string, int>& head) : _id(
        int GetId() const { return _id; }
        bool LoadProperty(const std::string line);
        static std::vector<std::string> ParserLine(std::string line);
        ...
protected:
    virtual void GenStruct() = 0;          // 生成内存结构
private:
    int _id;
    std::map<std::string, int>& _head;
    std::vector<std::string> _props;
};
```

当我们取到行数据串时，调用的函数是LoadProperty，将其变为一个内存对象。其实现如下：

```
bool ResourceBase::LoadProperty(const std::string line) {
    std::vector<std::string> propertyList = ParserLine(line);
    // 行最后可能是一个看不见的字符
    if (propertyList.size() < _head.size()) {
        LOG_ERROR("LoadProperty failed. " << "line size:" << prope
        return false;
    }
    for (size_t i = 0; i < propertyList.size(); i++) {
        _props.push_back(strutil::trim(propertyList[i]));
    }
    _id = std::stoi(_props[0]);
    GenStruct();
    return true;
}
```

加载函数将行字符串变成vector<string>数组。具体要生成什么内

存数据，是在GenStruct函数中实现的，GenStruct函数是一个虚函数，需要具体的子类去实现。

在基类中，我们将数据分成了两部分：一部分是行数据中重要的数据行Id；另一部分将这一行所有的数据读到vector<string>中。读取表头部分和行数据使用的是同一个函数ParserLine，我们按半角符号将数据拆分了，也就是说每一行（包括表头），数据都装在std::vector<std::string>中，只是第一行获取的是属性名，而其他行获取的是具体数据。以ResourceWorld的配置文件为例：

```
Id,Name,Init,Type,AbPath,ResName,UiResType,PlayerInitPos
1,login,0,1,scenes/login,Login,1,-1
```

如果CSV文件中有以上两行，在内存中它的头部数据就是一个vector<std::string>() {"Id", "Name", ...}，数据行是vector<std::string>() {"1", "login", ...}。

在CSV文件中，所有的字段读取到内存中都是string类型的，但在使用时的需求却不一样，例如Id需要的是int类型，所以在ResourceBase基类中提供了一系列取值的函数。例如GetInt，即按列名取得int类型。其实现如下：

```
int ResourceBase::GetInt(std::string name) {
    const auto iter = _head.find(name);
    if (iter == _head.end()) {
        LOG_ERROR("GetInt Failed. id:" << _id << " name:[" << name
        DebugHead();
        return 0;
    }
    return std::stoi(_props[iter->second]);
}
```

在ResourceBase类中，head的结构是std::map<std::string, int>，保存了名字与索引的对应关系。快速取得索引之后，就可以从行数据的vector<string>中直接取出值了。除了GetInt函数之外，该类中还提供了GetString、GetBool等一系列函数，方便读取数据。有了这些基础数据，不同的配置文件可以生成不同的内存数据结构。以地图资源

类来举例说明。其定义如下：

```
class ResourceWorld : public ResourceBase {
public:
    explicit ResourceWorld(std::map<std::string, int>& head);
    bool Check() override;
    std::string GetName() const;
    ResourceWorldType GetType() const;
    ...
protected:
    void GenStruct() override;
private:
    std::string _name{ "" };
    bool _isInit{ false };
    ResourceWorldType _worldType{ ResourceWorldTypePublic };
    Vector3 _initPosition{ 0,0,0 };
};
```

在这个具体的ResourceWorld类中，有了ResourceWorldType、Vector3这些属性。重点关注的函数是GenStruct，该函数是为了生成具体的内存数据而存在的，在每个类中都不一样。ResourceWorld类中的实现如下：

```
void ResourceWorld::GenStruct() {
    _name = GetString("name");
    _isInit = GetBool("init");
    _worldType = static_cast<ResourceWorldType>(GetInt("type"));
    std::string value = GetString("playerinitpos");
    std::vector<std::string> params;
    strutil::split(value, ',', params);
    if (params.size() == 3) {
        _initPosition.X = std::stof(params[0].c_str());
        _initPosition.Y = std::stof(params[1].c_str());
        _initPosition.Z = std::stof(params[2].c_str());
    }
}
```

GenStruct函数在基类数据分析时会被调用。在基类中，每一行数据都被保存在一个std::vector<std::string>中，如果我们使用时再去转换为bool或者int，就太消耗时间了。因此，牺牲了一些内存，

采用空间换取时间的方式，在每个行数据被读取上来之后就进行数据的转换。

在world.csv文件中，PlayerInitPos列中的数据相当复杂，使用的是一个（x，y，z）的结构，类似于“369.78，29.35，493.20”。该列表示玩家初始进入地图的位置坐标。在GenStruct函数中预先把这些数据转换成需要的内存数据结构Vector3。

除了GenStruct外，在ResourceBase基类中还有一个虚函数：

```
virtual bool Check() = 0;
```

当一行数据读到内存中时，调用该函数对这一行数据进行有效性检查，地图类和道具类的校验方式不一样，所以这里让继承的类去实现该函数。目前该功能太过简单，没有什么数据需要检查，所以在代码中，此函数也是一个空函数。

综上所述，我们以地图配置阐述了在服务端读取配置文件的流程，配置文件不单服务端需要用到，客户端也需要用到。表11-1解释了world文件每一列的用途。

表11-1 world文件的配置列表

列 名	说 明
Id	唯一序列号
Name	名字
Init	是不是初始进入的地图，也就是默认出生点
Type	地图类型
AbPath	地图客户端 3D 资源 AB 包名
ResName	在 AB 包中的资源名
UiResType	进入地图时是否有初始加载的 UI 界面，若有则为界面的类型
PlayerInitPos	玩家进入地图的初始坐标

在world配置表中，除了包含服务端需要的数据外，同时还有客户端需要的数据。该文件的前两行是关于“登录场景”和“选择角色场景”的数据，是供客户端使用的，在服务端并不会区分这两个场景，这些功能都是在Account类中完成的，而Account类不是一个场景类。

对于客户端而言，就每一张地图来说，它都有一个地图资源类ResourceWorld。同时，本书使用的是U3D，所以有AbPath和ResName属性，AbPath是打包文件，一个打包文件中可以有多个资源，ResName则指定了具体的资源名称。这些数据在服务端是不需要读取的。有关资源配置文件就介绍到这里，在后面真正应用时再进一步讲解。

地图类World与代理类WorldProxy

现在有了地图的配置数据，玩家可以进入地图了。本节将介绍服务端的地图类。因为是异步多进程，所以内容比单进程更加复杂一些。首先需要理解的概念是World和WorldProxy，这两个概念都很重要，而且相互关联。本节的源代码位于11_02_world目录中，下面讲解其重要部分。

图11-3展示的是World与WorldProxy的关系，可以从名字中得知WorldProxy是World的代理类。当一个玩家进入地图A时，地图A的实例类为World A，它的实例在某个space进程中，而玩家登录的进程是game进程，所以需要有一个WorldProxy来帮助玩家对数据进行中转，也就是WorldProxy类，每个WorldProxy代理一个World类。

进入某个地图时，在game进程上会为地图World实例创建一个代理类，但并不是每个World实例在game进程上都有代理类，只有game进程关心的World实例才创建代理类。也就是说，如果game1进程中没有玩家访问World B，就没有World B的代理类。一般来说，作为公共地图，每一个game进程上都有公共地图的代理类，但是作为副本地图，有些副本代理类仅在本地game进程中出现。

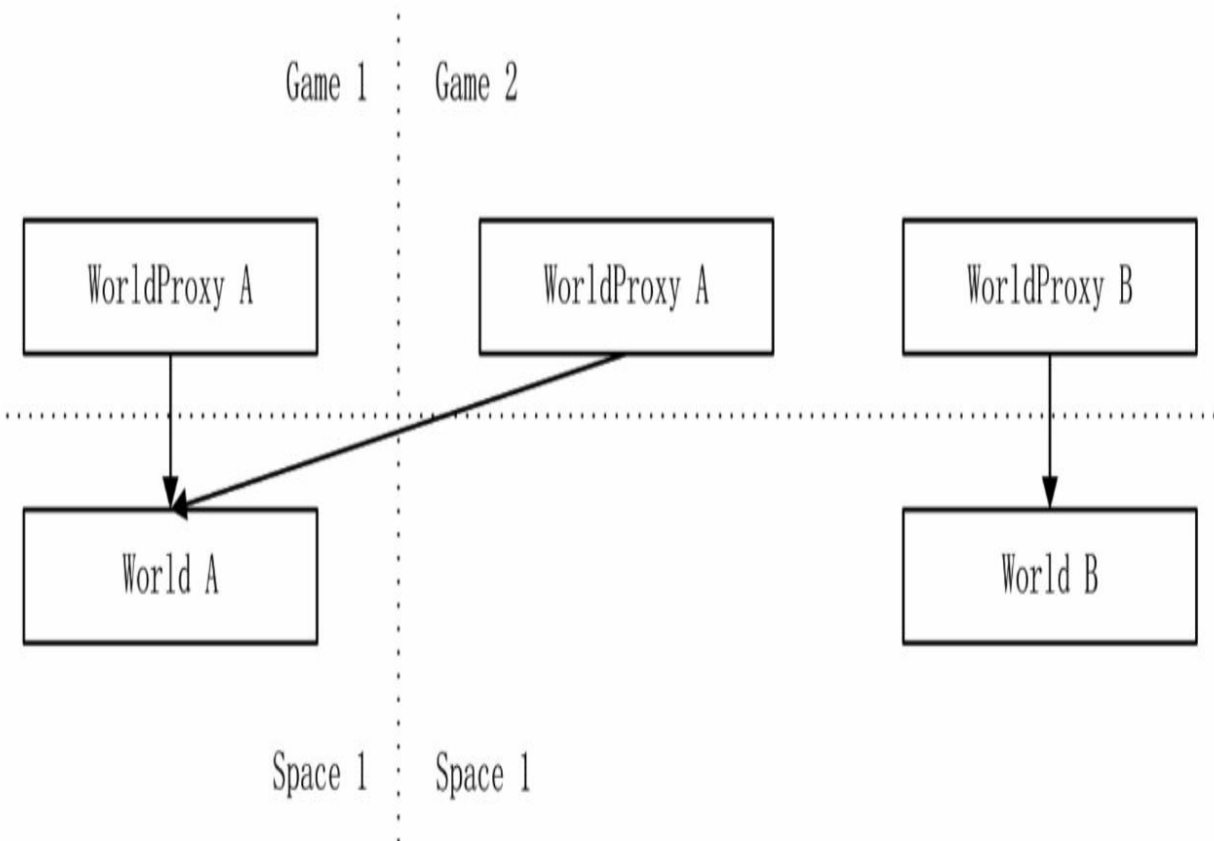


图11-3 World与WorldProxy的关系

先来看看11_02_world工程的结果，编译工程，执行allinone和robots，在robots进程中输入“login-a test”，登录账号test。在allinone控制台中可以看到如下信息：

```
[DEBUG] - enter game. account:test token:52EE6F81-83CF-428c-943E-
[DEBUG] - create world. id:3
[DEBUG] - create world proxy. Id:3
```

在分布式服务器中进入地图是一个比较复杂的过程，在11_02_world工程中完成进入地图的第一步，也就是World和WorldProxy的创建。当一个玩家进入game进程之后，将创建默认进入地图的实例与代理。打开world.csv文件，Id为3的地图是配置中默认初始进入的地图。现在我们来分析一下这个过程。

11.2.1 地图类World

World类在space进程上，而WorldProxy类在game进程上。从创建顺序上来讲，我们是先创建World类，再创建WorldProxy类，那么World类是何时被创建的呢？

打开game工程中的lobby.cpp文件，向Redis询问数据对客户端传来的token进行验证。验证成功之后，就真正进入地图的环节了。注意lobby类，该类本身并不是地图，它只是一个处理game进程中玩家进入地图的一个中间类。

首先，通过玩家角色数据读到内存中，分析角色数据，就可以分析出这个角色上一次登录的地图信息，这个信息放在了PlayerComponentLastMap组件中。

其次，得到了地图信息，知道这个玩家上次登录的地图是否还在，如果不在，就要读取上次登录的公共地图信息，如果玩家是第一次登录的，上次登录的公共地图就是初始id=3的地图。第一个玩家登录时，这个地图在game进程中并没有实例，所以game进程向appmgr发起了申请创建的协议。这个事件发生在向DB读取数据之后，读取DB数据的处理函数是Lobby::HandleQueryPlayerRs，所以申请创建地图协议的代码需要写在该函数中，实现如下：

```
void Lobby::HandleQueryPlayerRs(Packet* pPacket) {
    ...
    // 分析进入地图
    auto protoPlayer = protoRs.player();
    const auto playerSn = protoPlayer.sn();
    pPlayer->ParserFromProto(playerSn, protoPlayer);
    const auto pPlayerLastMap = pPlayer->AddComponent <PlayerCompo
    auto pWorldLocator = ComponentHelp::GetGlobalEntitySystem()->
    // 进入副本
    auto pLastMap = pPlayerLastMap->GetLastDungeon();
    if (pLastMap != nullptr && pWorldLocator->IsExistDungeon (pLas
        // 存在副本，跳转
        WorldProxyHelp::Teleport(pPlayer, GetSN(), pLastMap->World
        return;
    }
    // 进入公共地图
```

```

pLastMap = pPlayerLastMap->GetLastPublicMap();
const auto lastMapSn = pWorldLocator->GetWorldSnById(pLastMap-
if (lastMapSn != INVALID_ID) {
    // 存在公共地图，跳转
    WorldProxyHelp::Teleport(pPlayer, GetSN(), lastMapSn);
    return;
}
...
// 向appmgr申请创建地图
Proto::RequestWorld protoToMgr;
protoToMgr.set_world_id(pLastMap->WorldId);
MessageSystemHelp::SendPacket(Proto::MsgId::G2M_RequestWorld,
}

```

当第一个玩家进来的时候，是不可能有副本地图和公共地图的，所以向appmgr发起了请求协议G2M_RequestWorld，该协议请求一个地图所在的位置。因为我们要创建的是公共地图，公共地图是全局唯一的，必须采用串行的方式创建，所以appmgr是一个不错的选择。同一瞬间，有多个玩家登录多个game时，在appmgr中的处理也是串行的，不会出错。

在创建公共地图的问题上有非常多的解决方案，这取决于游戏的类型。有些游戏为每个game进程中创建一个主城实例，这样可以分担主城压力。在本框架中，认为无论有多少进程，公共地图（如主城）的实例都是全局唯一的。

本节的示例工程提供一种在分布式框架下创建全服唯一公共地图的思路以供读者参考。在工程中，创建协议发送之后，appmgr收到创建的消息，它会查询所有space的状态，找到一个相对空闲的space发起真正的创建地图协议。在appmgr工程中，处理协议G2M_RequestWorld的组件为CreateWorldComponent，文件是create_world_component.h，其处理函数如下：

```

void CreateWorldComponent::HandleRequestWorld(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::RequestWorld>();
    auto worldId = proto.world_id();
    auto pResMgr = ResourceHelp::GetResourceManager();
    const auto mapConfig = pResMgr->Worlds->GetResource(worldId);
    if (mapConfig == nullptr) {
        LOG_ERROR("can't find map config. id:" << worldId);
        return;
    }
}

```

```

if (!mapConfig->IsType(ResourceWorldType::Public)) {
    LOG_ERROR("appmgr recv create dungeon map. map id:" << worldId);
    return;
}

// 正在创建中，等待
const auto iter = _creating.find(worldId);
if (iter != _creating.end())
    return;

// 因为是异步的，有可能发送过来时已经被创建了
const auto iter2 = _created.find(worldId);
if (iter2 != _created.end())
    return;

AppInfo appInfo;
if (!GetOneApp(APP_SPACE, &appInfo)) {
    LOG_ERROR("appmgr recv create map. but no space process. m");
    return;
}

Proto::CreateWorld protoCreate;
protoCreate.set_world_id(worldId);
protoCreate.set_request_world_sn(0);
protoCreate.set_request_game_id(0);
MessageSystemHelp::SendPacket(Proto::MsgId::G2S_CreateWorld, p

    _creating[worldId] = appInfo.AppId;
}

```

G2S_CreateWorld协议发送给一个指定的space进程，该协议真正创建一个地图，定义如下：

```

message CreateWorld {
    int32 world_id = 1;
    uint64 request_world_sn = 2;
    int32 request_game_id = 3;
}

```

变量world_id是我们想要创建的地图在world.csv中的Id，变量request_game_id则是发出请求的game进程的AppId，request_world_sn是在game进程中发起这个请求的实体的唯一标识SN。创建协议

G2S_CreateWorld都是由WorldProxy代理类或Lobby发起的，这个SN是WorldProxy或Lobby的SN，以方便创建成功之后将结果发送到指定的进程中。注意，request_game_id和request_world_sn可能都为0。当request_game_id为0时，该地图创建信息将广播给所有game进程。

在space工程中，WorldOperatorComponent组件处理创建协议，定义在文件world_operator_component.h中。下面是它处理创建地图的方式：

```
void WorldOperatorComponent::HandleCreateWorld(Packet* pPacket) {
    auto protoWorld = pPacket->ParseToProto<Proto::CreateWorld>();
    int worldId = protoWorld.world_id();
    const int requestGameId = protoWorld.request_game_id();
    const uint64 requestWorldSn = protoWorld.request_world_sn();

    auto worldSn = Global::GetInstance()->GenerateSN();
    ThreadMgr::GetInstance()->CreateComponentWithSn<World>(worldSn)
    // 验证数据是否正确
    const auto pResMgr = ResourceHelp::GetResourceManager();
    const auto pWorldRes = pResMgr->Worlds->GetResource(worldId);
    if (pWorldRes->IsType(ResourceWorldType::Dungeon) && requestWo
        LOG_ERROR("create world error. dungeon is created. but req
    }

    // 如果requestWorldSn == 0，就广播给所有game和appmgr
    // 如果 requestWorldSn != 0，就广播给指定的game和appmgr
    Proto::BroadcastCreateWorld protoRs;
    protoRs.set_world_id(worldId);
    protoRs.set_world_sn(worldSn);
    protoRs.set_request_game_id(requestGameId);
    protoRs.set_request_world_sn(requestWorldSn);

    if ((Global::GetInstance()->GetCurAppType() & APP_APPMGR) == 0
        // 本进程中不包括appmgr，向appmgr发送消息
        MessageSystemHelp::SendPacket(Proto::MsgId::MI_BroadcastCr
    )
    // 本进程中不包括game
    if ((Global::GetInstance()->GetCurAppType() & APP_GAME) == 0)
        if (requestWorldSn == 0) {
            // 向所有game进程发送数据
            MessageSystemHelp::SendPacketToAllApp(Proto::MsgId:: M
        } else {
            // 向指定game发送数据
            MessageSystemHelp::SendPacket(Proto::MsgId:: MI_Broadc
```

```

        }
    }

    if ((Global::GetInstance()->GetCurAppType() & APP_GAME) != 0 |
        // 本进程中包括game和appmgr其中一个，需要中转消息
        MessageSystemHelp::DispatchPacket(Proto::MsgId:: MI_Broadc
    )
}

```

在上面的代码中，调用ThreadMgr::CreateComponentWithSn函数生成了一个World实例，同时广播了MI_BroadcastCreateWorld协议。全局搜索一下这个协议，会发现它有两个处理函数：一个在appmgr进程中；另一个在game进程中。当appmgr收到创建地图的消息时，其处理函数如下：

```

void CreateWorldComponent::HandleBroadcastCreateWorld(Packet* pPacket)
{
    auto proto = pPacket->ParseToProto<Proto::BroadcastCreateWorld>();
    const auto worldId = proto.world_id();
    const auto worldSn = proto.world_sn();
    const auto gameId = proto.request_game_id();

    const auto pResMgr = ResourceHelp::GetResourceManager();
    const auto pWorldRes = pResMgr->Worlds->GetResource(worldId);
    if (pWorldRes->IsType(ResourceWorldType::Public)) {
        _created[worldId] = proto.world_sn();
        _creating.erase(worldId);
    } else {
        _dungeons[worldSn] = gameId;
    }
}

```

收到创建地图成功的协议，appmgr对创建的数据进行清理，同时保存一份地图的Id与SN的对应数据。如果后面还有game进程请求公共地图的信息，就可以直接将数据反馈给game进程。

game进程收到创建地图的协议要做的事情更多，它必须为这个World创建一个WorldProxy代理类。创建协议的处理函数位于WorldProxyLocator组件中，处理代码如下：

```

void WorldProxyLocator::HandleBroadcastCreateWorld(Packet* pPacket)

```

```

std::lock_guard<std::mutex> guard(_lock);
auto proto = pPacket->ParseToProto<Proto::BroadcastCreateWorld>
const int worldId = proto.world_id();
const auto worldSn = proto.world_sn();
const auto gameId = proto.request_game_id();
const uint64 lastWorldSn = proto.request_world_sn();
const auto worldCfg = ResourceHelp::GetResourceManager()->World
if (worldCfg == nullptr) {
    LOG_ERROR("WorldProxyLocator::HandleBroadcastCreateWorld.
    return;
}
if (worldCfg->IsType(ResourceWorldType::Public)) {
    if (_publics.find(worldId) != _publics.end()) {
        LOG_ERROR(" WorldLocator. find same key. worldId:" <<
    }
    ThreadMgr::GetInstance()->CreateComponentWithSn<WorldProxy>
} else {
    if (gameId != Global::GetInstance()->GetCurAppId()) {
        LOG_ERROR("gameId != cur game id");
    }
    if (lastWorldSn > 0 && _worlds.find(lastWorldSn) == _world
        LOG_ERROR("can't find request world. world sn:" << las
    }
    ThreadMgr::GetInstance()->CreateComponentWithSn<WorldProxy>
}
}

```

不论何时，在game进程中收到地图的创建成功协议，都必须为其创建一个WorldProxy。如果是副本，就一定是game进程与space进程一对一发送的，创建结果space会发送到指定的game进程上，不需要再广播给所有game进程。

在启动时，如果采用的不是allinone，而是启动的多进程，那么在登录一个玩家时可以看到在每个game进程上都创建了WorldProxy。因为我们初始进入的是一个公共地图，所以公共地图的信息会广播给所有game进程。

11.2.2 为什么需要WorldProxy

对于分布式服务架构来说，game进程并不是一个最终的进程，它起着网关的作用。玩家进入的地图实际上是在space进程上，我们称为World A。如果有两个玩家在不同的game进程中，同时进入了World A，那么在每个game进程中为他们建立一个WorldProxy与World A进行交互。

从客户端的角度来看，两个玩家通过不同的IP或端口进入两个game进程，但是他们在同一张地图上。如果我们要做的是MOBA类游戏，匹配玩家之后，就一起进入某个地图实例中战斗，采用这种分布式的方式横向扩展game、space服务器，可以达到最优方案，对物理机进行充分利用，所有实例几乎平均分布在所有进程上。

除了本书中的方案外，也有一部分框架是这样操作的：进入副本地图之后，客户端与game进程的网络断开，重新创建一个与副本所在进程连接的Socket。本书中没有采用这种方式，而是采用WorldProxy方式，即保持Socket不变而采用代理。客户端直接连接space，操作是直接的，但是跳转地图时可能不断地更换网络连接。同时，也有一些不方便的处理。例如，Space A上的玩家要与Space B上的玩家进行交互，为了达到这个需求，这时所有space需要两两连接成一个巨大的网状网络结构，而且逻辑的交互会变得非常耦合。

用哪种方式完全取决于我们做的游戏类型。

本书中的框架采用WorldProxy代理，客户端与space之间是不会直接通信的，space只需要维护好自己的地图实例即可，交互是由game进程去完成的。客户端也不需要频繁更换网络连接，永远只有一个与game进程的网络通道。假设这时地图A上的玩家要跳转到地图B上，在game进程中存在WorldProxy A和WorldProxy B两张代理地图，在代理层面就变成了WorldProxy A和WorldProxy B之间的跳转交互，至于WorldProxy B对应的实例在哪里，WorldProxy A并不关心，反正最终协议会投递给真正的World B上的玩家。

在分布式服务器中，客户端连接的game进程的功能相对简单，它是一个网关，也是一个协议分发器。而玩家数据的真正操作是在某个space进程中的。

图11-4展示了WorldProxy的创建流程。我们以一个例子来说明这个流程：如果要做MOBA游戏，它的流程在框架中的设计为：两个客户端随机登录game进程，通过匹配功能模块，两个玩家建立的战场实例被放到space N进程上。这时，game1和game2都会为这个地图创建WorldProxy，作为客户端协议的中断处。操作协议最终的处理点在space N进程的某个地图实例中。当有无数个客户端同时匹配的时候，这种模式会将战场平均分配在所有space进程上，以达到负载均衡的目的。

关于World的创建更多的细节可以参考源代码。下面进入11_02_world目录，看一下内存数据。启动6个进程：

```
./dbmgrd
./appmgrd
./logind -sid=101
./gamed -sid=201
./gamed -sid=202
./spaced -sid=301
```

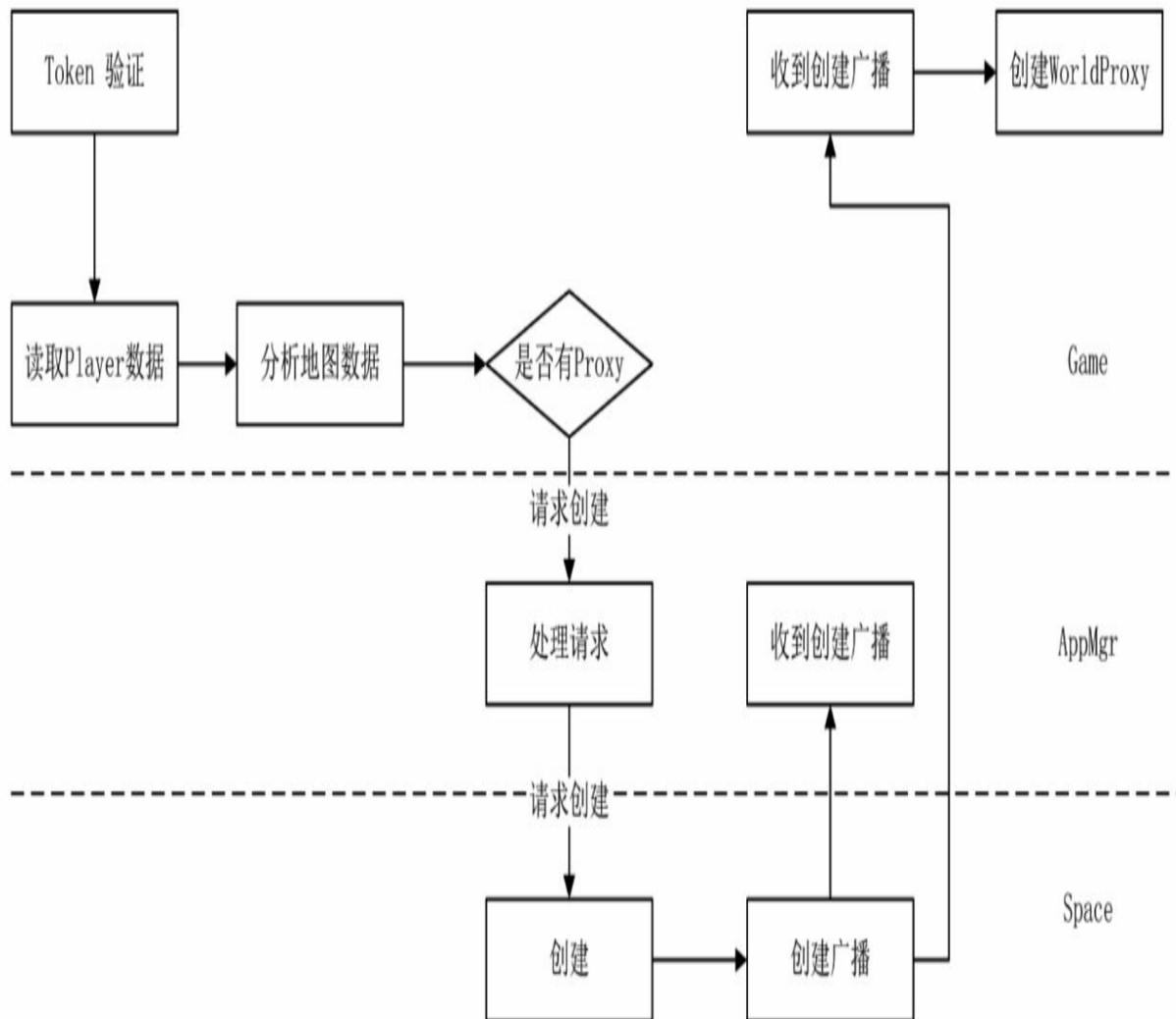


图11-4 WorldProxy的创建流程

随后，启动机器人./robotsd。输入“login-a test”登录test账号，这时可以看到两个game进程中打印“create world proxy.id: 3”。当有玩家登录就会触发创建Id为3的World，两个game进程都为这个World创建了WorldProxy代理类。

在game进程中输入“proxy-all”，可以看到当前服务器上全部的WorldProxy实例信息：

```

**** world proxy gather ****
sn:103842198881043186 proxy sn:103842198881043186 online:1 name:lo
sn:103842214819398893 proxy sn:103842214819398893 online:0 name:ji

```

其中一个game进程上的online数据必为1。但我们可以发现，虽然地图有了，但是这个玩家依然在Lobby大厅中。我们创建了地图与代理地图，但真正困难的工作还没有开始，当WorldProxy创建成功之后，玩家是如何跳转到WorldProxy上的，玩家数据又是如何传递到space上的呢？这是11.3节要阐述的问题。

分布式地图跳转流程

在分布式模式中，角色的跳转是比较困难的部分，要考虑的事情比较多，先从简单的入手。验证账号初次进入地图，可以看成是一次跳转，框架中有Lobby类，它相当于一个大厅。每个玩家进入game进程，首先进入的就是这个大厅。在大厅中进行token验证，验证成功之后才能进入上次登录的副本地图或公共地图。如果不是一个分布式系统，那么一个单进程跳转流程图如图11-5所示。

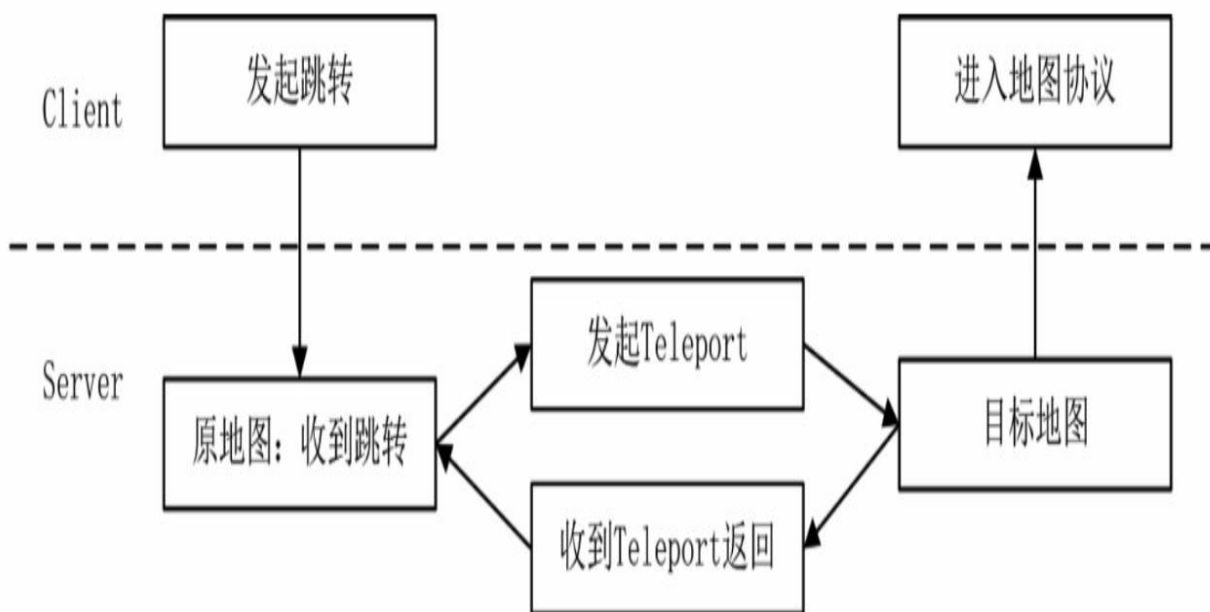


图11-5 单进程跳转流程图

首先由客户端发起一个跳转请求，原地图收到这个请求之后，组织跳转协议，并且带上玩家数据发送到目标地图。目标地图收到协议并处

理完成之后，给原地图返回一个数据，原地图删除玩家数据。如果目标地图返回失败，就维持在原地图，客户端显示跳转失败；如果跳转成功，客户端就会收到一个进入地图的协议，加载新地图资源。

图11-6所示为多进程跳转流程图，这个过程稍微显得有点复杂。我们以从Lobby大厅跳转到地图为例进行简单的说明。在异步流程中，首先需要判断目标地图的代理是否存在，如果不存在，就申请创建，创建是一个异步的过程，只有等到创建地图成功的广播，原地图才能向目标地图的代理WorldProxy发起跳转Teleport协议，收到Teleport协议的返回协议表示已跳转成功，原地图才能删除玩家数据。

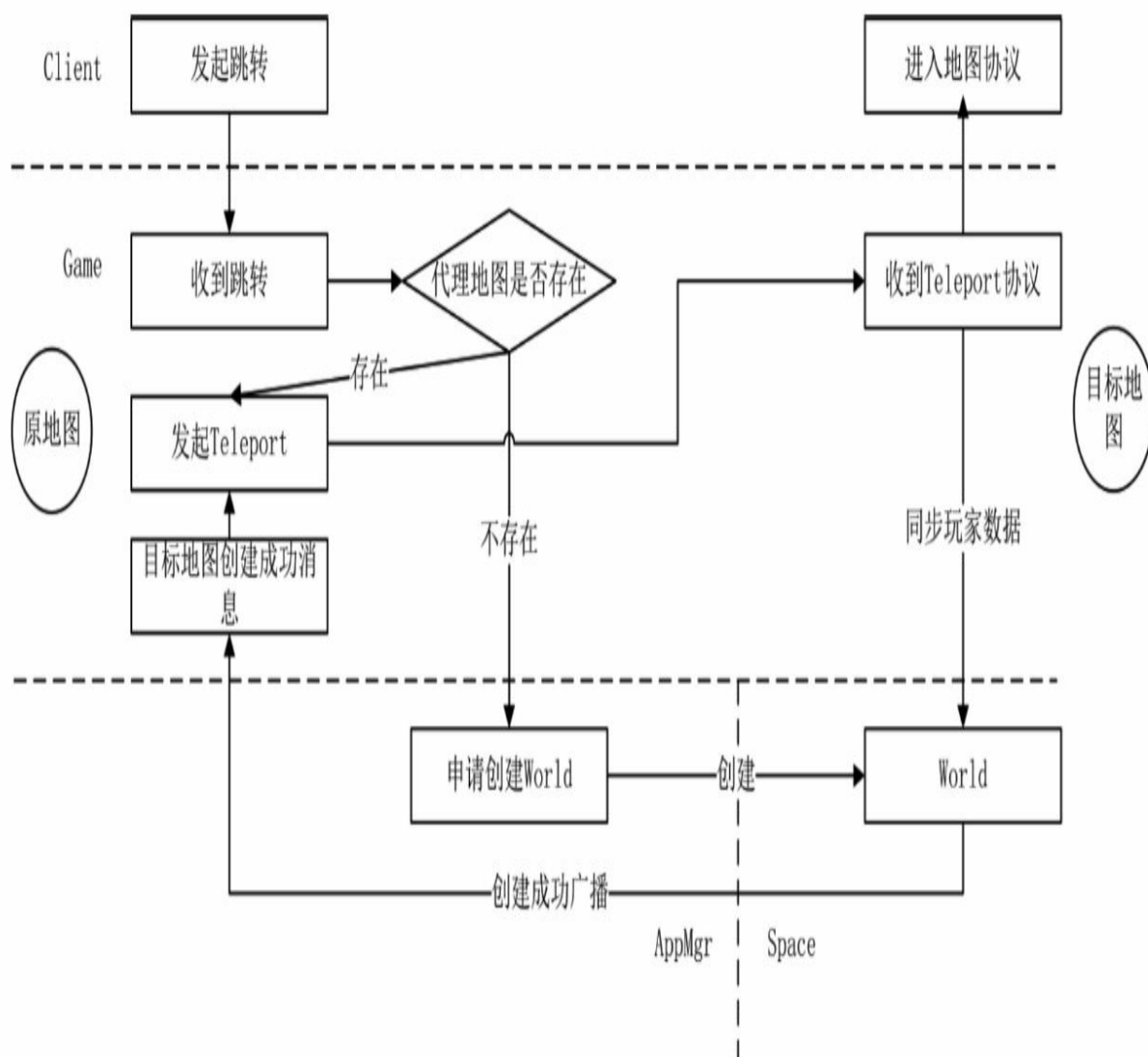


图11-6 多进程跳转流程图

本节的源代码位于目录11_03_teleport中。先来看工程的执行结果。编译工程，启动allinone。启动robots，以test账号登录，登录完成之后，在allinone控制台输入“proxy-all”，打印信息如下：

```
**** world proxy gather ****
sn:103842460991488957 proxy sn:103842460991488957 online:0 name:lo
sn:103842461931013936 proxy sn:103842461931013936 online:1 name:ji
```

在allinone控制台输入“world-all”，打印信息如下：

```
**** world gather ****
sn:103842461931013936 online:1
```

通过与11.2节的示例进行比较，发现之前在Lobby中的玩家已经进入地图中。下面我们来分析一下角色跳转的几个重点。

11.3.1 发起跳转协议的时机

当角色token验证成功之后，登录大厅Lobby，有两个时机发送跳转Teleport协议：

（1）第一个时机，当玩家登录进来时没有WorldProxy代理类，所以必须申请创建一个地图。创建完成之后，收到WorldProxy创建成功的广播，此时Lobby类中等待跳转到该地图的玩家统一发送跳转协议给目标地图。

等待WorldProxy的创建是一个异步过程，Lobby为每一张地图的跳转角色做了一个等待队列。在Lobby类中，类型为std::map<int, std::set<uint64>>的字典保存了地图Id与其对应的玩家等待列表。有时并不只有一个玩家在等待，所以是set数据（集合数据）。

处理WorldProxy创建成功的广播的函数是Lobby::HandleBroadcastCreateWorldProxy。在这个函数中调用了WorldProxyHelp::Teleport函数进行地图的跳转。

（2）第二个时机，当玩家登录进来时，如果WorldProxy已经存在了，Lobby就可以立即向WorldProxy发起跳转请求。在这个流程中不需要任何缓存数据，可以直接调用WorldProxyHelp::Teleport函数发起跳转。

11.3.2 跳转协议的数据定义

下面是跳转协议Teleport的数据定义：

```
message Teleport {
    uint64 last_world_sn = 1;
    uint64 player_sn = 2;
    string account = 3;
    Player player = 4;
}
```

在这个协议中传递了玩家数据、账号、当前地图的SN以及玩家SN。服务器进程上有无数个WorldProxy。我们是如何将跳转协议发送到正确的WorldProxy中去呢？

因为会被多次调用，所以代码中将跳转功能移到了一个独立的函数中。以下是它的实现：

```
void WorldProxyHelp::Teleport(Player* pPlayer, const uint64 lastWo
    const uint64 playerSn = pPlayer->GetPlayerSN();
    Proto::Teleport proto;
    proto.set_last_world_sn(lastWorldSn);
    proto.set_account(pPlayer->GetAccount().c_str());
    proto.set_player_sn(playerSn);
    pPlayer->SerializeToProto(proto.mutable_player());

    NetIdentify netIdentify;
    netIdentify.GetSocketKey()->CopyFrom(pPlayer->GetSocketKey());
    netIdentify.GetTagKey()->AddTag(TagType::Entity, targetWorldSn
    MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_Teleport, p
}
```

在上面的代码中组织了Proto::Teleport结构，将需要的数据传递进去。值得注意的是NetIdentify类，这个类是为了定义一个网络身份而存在的。之前的名字叫NetworkIdentify，其定义如下：

```
struct NetworkIdentify {
```

```
...
protected:
    SocketKey _socketKey{ INVALID_SOCKET, NetworkType::None };
    ObjectKey _objKey{ ObjectKeyType::None , {0, ""} };
};
```

现在优化一下它的数据，将ObjectKey替换为TagKey：

```
struct NetIdentify {
    ...
protected:
    SocketKey _socketKey{ INVALID_SOCKET, NetworkType::None };
    TagKey _tagKey;
};
```

下面来看ObjectKey与TagKey的定义：

```
struct ObjectKey {
    ObjectKeyType KeyType{ ObjectKeyType::None };
    ObjectKeyValue KeyValue{ 0, "" };
    ...
}
enum class ObjectKeyType {
    None = Proto::NetworkObjectKeyType::ObjectKeyTypeNone,
    Account = Proto::NetworkObjectKeyType::ObjectKeyTypeAccount,
    App = Proto::NetworkObjectKeyType::ObjectKeyTypeApp,
};
```

在简单的网络中，使用一个ObjectKey就可以定义出一个关系。但随着网络变得复杂，除了有客户端与服务端之间的连接外，还有服务端内部各个进程的连接，采用打标签的方式来定义一个网络身份。

```
struct TagKey {
    ...
    std::map<TagType, TagValue> _tags;
};
enum class TagType {
    None = Proto::TagType::TagTypeNone,
    Account = Proto::TagType::TagTypeAccount,
```

```
App = Proto::TagType::TagTypeApp,  
Entity = Proto::TagType::TagTypeEntity,  
ToWorld = Proto::TagType::TagTypeToWorld,  
Player = Proto::TagType::TagTypePlayer,  
};
```

结构TagKey是一个升级版的ObjectKey。在之前的Identify类中只有一个单一的对象识别，而现在可以打上多个标签。回到跳转的协议上，当我们发送协议时，为Packet打上了一个标签：

```
NetIdentify;  
netIdentify.GetSocketKey()->CopyFrom(pPlayer->GetSocketKey());  
netIdentify.GetTagKey()->AddTag(TagType::Entity, targetWorldSn  
MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_Teleport, p
```

我们将目标地图的SN赋值保留在了Packet的标签中。这样做的好处是，通过这个标签可以更快找到需要的对象。

在消息处理系统中，这些Tag可以很方便找到实例对象，直接将Packet塞给它处理。以避免不必要的轮询。下面是消息系统的处理实现：

```
void MessageSystem::Update(EntitySystem* pEntities) {  
    ...  
    auto packetLists = _cachePackets.GetReaderCache();  
    for (auto iter = packetLists->begin(); iter != packetLists->en  
        auto pPacket = (*iter);  
        const auto finditer = _callbacks.find(pPacket->GetMsgId())  
        if (finditer != _callbacks.end()) {  
            auto handleList = finditer->second;  
            auto pTagValue = pPacket->GetTagKey()-> GetTagValue(Ta  
            if (pTagValue != nullptr) {  
                auto entitySn = pTagValue->KeyInt64;  
                auto pMsgCallback = handleList[entitySn];  
                if (pMsgCallback != nullptr) {  
                    pMsgCallback->ProcessPacket(pPacket);  
                }  
            }  
        }  
        ...  
    }  
    ...  
}
```

```
    }  
    _cachePackets.GetReaderCache()->clear();  
}
```

在处理Packet时，如果Packet被打上了Entity的标签，那么标志着这个Packet是一个定向处理，可以先从数据中优先找到对应的Entity对象，直接丢给这个Entity处理。

11.3.3 目标代理地图收到跳转协议

WorldProxy类收到跳转协议的处理函数为WorldProxy::HandleTeleport，代码实现如下：

```
void WorldProxy::HandleTeleport(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::Teleport>();
    const auto playerSn = proto.player_sn();
    auto pCollector = GetComponent<PlayerCollectorComponent>();
    auto pPlayer = pCollector->AddPlayer(pPacket, proto.account())
    if (pPlayer == nullptr) {
        LOG_ERROR("failed to teleport, account:" << proto.account()
        return;
    }

    pPlayer->ParserFromProto(playerSn, proto.player());
    pPlayer->AddComponent<PlayerComponentOnlineInGame> (pPlayer->G

    // 将数据转给真实的world
    Proto::SyncPlayer protoSync;
    protoSync.set_account(proto.account().c_str());
    protoSync.mutable_player()->CopyFrom(proto.player());
    SendPacketToWorld(Proto::MsgId::G2S_SyncPlayer, protoSync, pPl

    // 通知旧地图，跳转成功
    Proto::TeleportAfter protoTeleportRs;
    protoTeleportRs.set_player_sn(pPlayer->GetPlayerSN());
    NetIdentify indentify;
    indentify.GetTagKey()->AddTag(TagType::Player, pPlayer->GetPla
    indentify.GetTagKey()->AddTag(TagType::Entity, proto.last_worl
    MessageSystemHelp::DispatchPacket(Proto::MsgId::MI_TeleportAft
}
```

在上面的代码中做了两件事：第一，通知旧地图跳转成功，删除玩家数据，使用了MI_TeleportAfter协议；第二，将玩家的数据传递到被代理的真实地图中，使用了G2S_SyncPlayer协议。在发送该协议时使用了SendPacketToWorld函数，为了不写冗余代码，我们将发送协议封装成一个函数：

```
void WorldProxy::SendPacketToWorld(const Proto::MsgId msgId, ::goo
```

```
TagKey tagKey;
tagKey.AddTag(TagType::Player, pPlayer->GetPlayerSN());
tagKey.AddTag(TagType::Entity, _sn);
MessageSystemHelp::SendPacket(msgId, proto, &tagKey, APP_SPACE
}
```

从WorldProxy发送任何协议都需要打上两个标签：一个是Entity；另一个是Player。这是为了方便在space进程中快速找到目标。如果我们运行的是allinone，这个Packet就会被DispatchPacket转发，不会经过网络层。如果启动的是多进程，这个Packet就会先被序列化为字符串并传递到space进程上，这就有了一个新的问题：Tag数据是如何在网络数据中传递的呢？

11.3.4 网络标识如何在网络中传递

如果对于前面的网络格式还有印象的话，还记得自定义协议格式吗？自定义协议的格式是：总长度+PacketHead+协议体。其中，PacketHead中包括一个MsgId。

在本节的示例代码中，我们修改一下这个格式，对于客户端来说，“总长度+PacketHead+协议体”这样的格式是够用的，一旦需要数据在game到space进程之间进行传递，这个格式就不够用了，WorldProxy会发送很多中转协议给space进程。

当我们传递一段网络数据时，在game进程中，一段网络数据一定是从一个特定的Socket中取出来的，一个特定的Socket对应着一个特定的玩家。当我们需要进入3号地图时，只需要发送3这个数字，就表示这个玩家要进入3号地图。但在服务进程内部则不能这样，从game进程发送到space进程的数据，它们之间只有一个Socket，却对应着许多玩家。有一种解决办法是在每一个协议定义时加上PlayerSn和EntitySn，以定位该协议的归属，但这种方法既不方便，处理起来又不高效，因为需要将整个包打开之后才知道具体数据。

但是没有这两个数据的话，这个协议又应该发送给谁呢？本框架中采用了打标签的方式，不修改具体协议的内容，而是给传递的Packet类打上标签。通过网络传输之后，这些标签内容也需要传输到对端，所以修改了整体协议的格式，变成了可变的格式：

(1) 总长度+头长度+PacketHead+协议体

(2) 总长度+头长度+PacketHeadS2S+协议体

在客户端采用第一种方式，而在服务端内部采用第二种方式。两个Packet头定义如下：

```
struct PacketHead {
    unsigned short MsgId;
};
struct PacketHeadS2S :public PacketHead {
    uint64 EntitySn;
```

```
uint64 PlayerSn;
};
```

在network_buffer.cpp文件中发送数据和接收数据的两个函数需要进行相应修改。其中发送函数代码如下：

```
void SendNetworkBuffer::AddPacket(Packet* pPacket) {
    const auto pTagValue = pPacket->GetTagKey()->GetTagValue(TagTy
    ...
    // 1.整体长度
    MemcpyToBuffer(reinterpret_cast<char*>(&totalSize), sizeof(Tot
    // 2.头部
    if (pTagValue == nullptr) {
        PacketHead head{};
        head.MsgId = pPacket->GetMsgId();
        TotalSizeType headSize = sizeof(PacketHead);
        MemcpyToBuffer(reinterpret_cast<char*>(&headSize), sizeof(
        MemcpyToBuffer(reinterpret_cast<char*>(&head), sizeof(Pack
    } else {
        PacketHeadS2S head{};
        head.MsgId = pPacket->GetMsgId();
        head.EntitySn = pTagValue->KeyInt64();
        const auto pTagPlayer = pPacket->GetTagKey()->GetTagValue(
        if (pTagPlayer == nullptr)
            head.PlayerSn = 0;
        else
            head.PlayerSn = pTagPlayer->KeyInt64();
        TotalSizeType headSize = sizeof(PacketHeadS2S);
        MemcpyToBuffer(reinterpret_cast<char*>(&headSize), sizeof(
        MemcpyToBuffer(reinterpret_cast<char*>(&head), sizeof(Pack
    }
    ...
}
```

上面的代码将一个Packet序列化到缓冲区中，主要来看头部数据。在服务端内部，Head结构与客户端发送到服务器的数据是不一样的，判断的标识就是Packet中是否有TagType: : Entity数据，如果有，就认为是内部服务端与内部服务端之间的S2S协议。在space进程上收到网络数据时进行了解析，解析代码如下：

```
Packet* RecvNetworkBuffer::GetTcpPacket() {
```

```

// 1.读出整体长度
unsigned short totalSize;
MemcpyFromBuffer(reinterpret_cast<char*>(&totalSize), sizeof(T
...
// 2.头部长
unsigned short headSize;
MemcpyFromBuffer(reinterpret_cast<char*>(&headSize), sizeof(To
RemoveDate(sizeof(TotalSizeType)));
// 3.读出 PacketHead
PacketHead* pHead;
PacketHeadS2S* pHeadS2s = nullptr;
if (headSize == sizeof(PacketHead)) {
    PacketHead head;
    MemcpyFromBuffer(reinterpret_cast<char*>(&head), sizeof(Pa
RemoveDate(sizeof(PacketHead)));
    pHead = &head;
} else {
    PacketHeadS2S head;
    MemcpyFromBuffer(reinterpret_cast<char*>(&head), sizeof(Pa
RemoveDate(sizeof(PacketHeadS2S)));
    pHead = &head;
    pHeadS2s = &head;
}
...
Packet* pPacket = MessageSystemHelp::CreatePacket((Proto::MsgI
...
if (pHeadS2s != nullptr) {
    auto pTagKey = pPacket->GetTagKey();
    pTagKey->AddTag(TagType::Entity, pHeadS2s->EntitySn);
    pTagKey->AddTag(TagType::Player, pHeadS2s->PlayerSn);
}

return pPacket;
}

```

在收到数据时，通过头结构的长度来判断是PacketHeadS2S结构还是PacketHead结构。如果是PacketHeadS2S结构，就将Tag重新写到Packet上，这个Packet实例随后会进入各个线程中，在消息处理系统中找到真正的World并处理。

在World中，我们定义了玩家数据同步协议G2S_SyncPlayer的处理函数：

```

void World::HandleSyncPlayer(Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::SyncPlayer>();
    const auto playerSn = proto.player().sn();

    auto pPlayerMgr = GetComponent<PlayerManagerComponent>();
    auto pPlayer = pPlayerMgr->AddPlayer(playerSn, GetSN(), pPacke
    pPlayer->ParserFromProto(playerSn, proto.player());
    pPlayer->AddComponent<PlayerComponentDetail>();

    const auto pComponentLastMap = pPlayer->AddComponent<PlayerCom
    pComponentLastMap->EnterWorld(_worldId, _sn);
    const auto pLastMap = pComponentLastMap->GetCur();

    //通知客户端进入地图
    Proto::EnterWorld protoEnterWorld;
    protoEnterWorld.set_world_id(_worldId);
    pLastMap->Position.SerializeToProto (protoEnterWorld.mutable_p
    MessageSystemHelp::SendPacket(Proto::MsgId::S2C_EnterWorld, pr
}

```

11.3.5 space进程发送的协议如何转发到客户端

在11.3.4小节的代码中，向客户端发送了一个S2C_EnterWorld协议，带上了地图的Id与玩家所处的坐标，表示该玩家已经进入某地图，但是space进程与客户端并没有直接的Socket通信。那么，这个协议是如何发送到客户端的呢？

回顾前面同步玩家数据G2S_SyncPlayer协议的处理代码，每个World类中有一个PlayerManagerComponent来管理玩家数据。同步玩家数据时，会将玩家加入这个管理类中。下面来看加入的代码：

```
Player* PlayerManagerComponent::AddPlayer(const uint64 playerSn, u
    ...
    const auto pPlayer = GetSystemManager()->GetEntitySystem()-> A
    _players[playerSn] = pPlayer;
    return pPlayer;
}
void Player::Awake(NetIdentify* pIdentify, uint64 playerSn, uint64
    _account = "";
    _playerSn = playerSn;
    _player.Clear();

    if (pIdentify != nullptr)
        _socketKey.CopyFrom(pIdentify->GetSocketKey());
    _tagKey.Clear();
    _tagKey.AddTag(TagType::Player, playerSn);
    _tagKey.AddTag(TagType::Entity, worldSn);
}
```

当Player实例被创建时，它的NetIdentify就被确定好了，SocketKey是发送G2S_SyncPlayer协议的Socket，也就是game与space之间的Socket，同时给Player打上TagType::Player标签和TagType::Entity标签。TagType::Entity标签中的值是WorldProxy的SN。当发送协议时，调用代码如下：

```
MessageSystemHelp::SendPacket(Proto::MsgId::S2C_EnterWorld, protoE
```

其原型如下：

```
MessageSystemHelp::SendPacket(const Proto::MsgId msgId, google::pr
```

该函数发送数据时，向game发送了TagType：：Player标签和TagType：：Entity标签，究竟这个协议内部是什么，转发时不需要关心，通过TagType：：Entity标签可以找到WorldProxy。

作为一个中转类，WorldProxy不可能一个一个协议去解析再发送，所以增加一个HandleDefaultFunction函数，这个WorldProxy中默认的消息处理函数，将客户端的协议转发到space进程上，同时将space进程上的协议转发给客户端。下面来看它的实现：

```
void WorldProxy::HandleDefaultFunction(Packet* pPacket) {
    auto pPlayerMgr = this->GetComponent<PlayerCollectorComponent>
    Player* pPlayer = nullptr;
    const auto pTagKey = pPacket->GetTagKey();
    if (pTagKey == nullptr) {
        LOG_ERROR("world proxy recv msg. but no tag. msgId:" << Lo
        return;
    }

    bool isToClient = false;
    const auto pTagPlayer = pTagKey->GetTagValue(TagType::Player);
    if (pTagPlayer != nullptr) {
        isToClient = true;
        pPlayer = pPlayerMgr->GetPlayerBySn(pTagPlayer->KeyInt64);
    } else {
        pPlayer = pPlayerMgr->GetPlayerBySocket(pPacket->GetSocket
    }

    // 可能协议传来时已经断线了
    if (pPlayer == nullptr)
        return;

    // 默认只进行中转操作
    if (isToClient) {
        auto pPacketCopy = MessageSystemHelp::CreatePacket((Proto:
        pPacketCopy->CopyFrom(pPacket);
        MessageSystemHelp::SendPacket(pPacketCopy);

        LOG_DEBUG("transfer msg to client. msgId:" << Log4Help::Ge
```

```

    } else {
        CopyPacketToWorld(pPlayer, pPacket);
        LOG_DEBUG("transfer msg to space. msgId:" << Log4Help::Get
    }
}

```

从space进程上发送的进入地图的S2C_EnterWorld协议最终走到了HandleDefaultFunction函数中。这时，协议被打上了TagType：：Player标签，它一定是从另一个服务进程中发送来的协议，所以可以直接转发给客户端。

HandleDefaultFunction函数是如何被调用的呢？我们修改了MessageSystem的流程，首先注册一个默认处理函数：

```

class MessageSystem :virtual public ISystem<MessageSystem> {
...
private:
    // 默认处理函数
    // <objsn, callback>
    std::map<uint64, IMessageCallBack*> _defaultCallbacks;
};

```

在MessageSystem系统中，默认处理函数对应的键（Key）为EntitySn，也就是WorldProxy的SN。当有协议到来时，调用Update处理协议：

```

void MessageSystem::Update(EntitySystem* pEntities) {
...
    auto packetLists = _cachePackets.GetReaderCache();
    for (auto iter = packetLists->begin(); iter != packetLists->en
        auto pPacket = (*iter);
        const auto finditer = _callbacks.find(pPacket->GetMsgId())
        if (finditer != _callbacks.end()) {
            ...
        } else {
            // 是否有默认处理函数
            const auto pTagValue = pPacket->GetTagKey()->GetTagVal
            if (pTagValue != nullptr) {
                // 如果是一个给World的协议，那么是不可能默认处理函数的
            }
        }
    }
}

```

```

// 采用allinone方式启动时，地图World实例和它的代理地图Wo:
const auto pTagToWorld = pPacket->GetTagKey()->Get
if (pTagToWorld == nullptr) {
    auto entitySn = pTagValue->KeyInt64;
    auto pMsgCallback = _defaultCallbacks[entitySn
    if (pMsgCallback != nullptr) {
        pMsgCallback->ProcessPacket(pPacket);
    }
}
}
}
...
}
...
}

```

在处理消息时，如果发现消息没有处理函数，就转向查看这个指定的Entity是否有默认的处理函数。在进入地图S2C_EnterWorld协议时，WorldProxy没有单独处理这个协议的函数，转向WorldProxy之前注册的HandleDefaultFunction函数中，从而发送给客户端。

通过客户端进入游戏

为了测试服务器，本书提供了一个Unity版本的客户端工程。关于客户端工程的获取，参见前言中获取本书源代码部分。本书使用的是Unity 2018.3.7f1（64-bit）版本。

在客户端目录下有两个工程文件，其中art目录是美术工程，包括场景、角色3D模型和UI，这些资源已经在工程中整理完成，可以直接使用。用Unity打开art工程，选择菜单命令Engine→Assets→BuildAB来打包Windows版本，这个菜单的功能是为美术资源打包。单击菜单之后，Unity会自动打包所有需要的资源，稍等一会，执行完成之后会弹出完成对话框。在Assets目录下会生成StreamingAssets目录，因为打包的是Windows版本，所有的文件将存放在assetbundle.win目录中，这个目录包括已经打包好的所有AB文件。如果想要在远端访问，那么可以直接将打包好的目录放到HTTP服务器下。

关闭art目录，用Unity打开Demo，在这个工程下有客户端的代码，包括网络层、UI和自动加载等一系列的代码。整个客户端采用事件驱动的方式，主要可以看EventDispatcher类和MessagePackDispatcher类，UI采用动态加载的方式，UI的数据更新采用UI中间层数据来实现，UI中间层的设计方式可以很好地将UI与协议之间的耦合断开。在实际开发中，我们经常会对UI进行修改，使用UI中间层数据可以很好地将注意力放在界面上，而不是放在逻辑上。

先来运行一下客户端，在Unity编辑器中选中Scenes目录，其中有Loader和Start两个场景文件，Loader用于加载，Start是我们的开始场景。Start场景中只有一个Main Camera实例。在这个相机实例上添加了一个脚本，这个脚本指定了一些运行所必需的数据：

· ResPath: 填入art项目中生成的assetbundle.win全路径, 这里也可以是一个远端地址。

· ReferencePath: CSV文件的目录, 指向服务端11_03_teleport\res\resource目录的绝对地址。如果我们要运营一个正式的游戏, 那么CSV文件肯定是打包成AB文件进行加载的, 因为这里只是一个测试示例, 所以直接写上本地路径。

· ServerIp: 服务端的HTTP地址。

· ServerPort: 服务端的HTTP端口。在engine.yaml文件中默认是7070。

设置好之后, 启动服务端, 就可以开始游戏了。

(1) 进入登录界面, 输入账号, 不填写密码。

(2) 如果输入的账号没有角色, 就会弹出角色创建界面, 输入名字并选择性别, 创建一个角色。

(3) 账号有现成的角色或创建新角色成功之后会进入角色选择界面, 选择一个角色进行游戏, 随后加载场景, 角色进入场景中。

简要分析客户端的协议, 打开NetworkMgr.cs文件:

```
protected override void OnAwake() {  
    ...  
    RegisterPacket(Proto.MsgId.C2LAccountCheckRs, Process<Proto.Ac  
    RegisterPacket(Proto.MsgId.C2LCreatePlayerRs, Process<Proto.Cr  
    RegisterPacket(Proto.MsgId.C2LSelectPlayerRs, Process<Proto.Se  
    RegisterPacket(Proto.MsgId.S2CEnterWorld, Process<Proto.EnterW  
    RegisterPacket(Proto.MsgId.L2CGameToken, Process<Proto.GameTok  
    RegisterPacket(Proto.MsgId.C2GLoginByTokenRs, Process<Proto.Lo  
    RegisterPacket(Proto.MsgId.L2CPlayerList, Process<Proto.Player  
    RegisterPacket(Proto.MsgId.G2CSyncPlayer, Process<Proto.SyncPl  
    RegisterPacket(Proto.MsgId.S2CRoleAppear, Process<Proto.RoleAp  
    RegisterPacket(Proto.MsgId.S2CMove, Process<Proto.Move>);  
}
```

虽然在服务端做了很多线程、进程, 编写了很多跳转、转发, 但是对于客户端来说, 从登录到进入场景, 需要处理的仅涉及上面这些协

议。前3个是返回协议，分别处理了验证账号、创建角色和选择角色的返回信息。

这里要特别强调一点，一般来说，返回协议都是出错时的返回显示，也就是说，如果一个协议是正确的，那么它必然引发一个事件，这个事件不是一个返回协议可以描述的。在本书框架的代码设计中，协议的返回协议只是出错描述。打个比方，在背包里，我们使用道具、拆分道具或丢掉道具，这3个协议都涉及一个事件，那就是背包的同步事件，背包在操作中必然发生改变。

如果我们设计协议时使用道具协议返回UseItemRs{ItemId, ItemCount}，丢弃道具协议返回DropItemRs{ItemId, ItemCount}，每个协议都返回一个它涉及的道具Id与数量，那么在客户端需要处理每个协议对道具进行更新。这样编写服务端程序的程序员麻烦，编写客户端程序的程序员也麻烦。一个更为有效的方式应该是，仅在UseItemRs或是DropItemRs协议中返回一个出错码，如果没有错，可以返回0。使用道具或是丢弃道具引发的道具的变更则是统一用道具同步协议SyncItem来通知客户端，这时逻辑会被大大简化。无论什么时候，操作道具只需要处理好当前这个协议要解决的事情，而SyncItem自然有道具的底层函数去处理它。

有些框架设计时，认为一个请求协议一定要有一个返回协议，这是对异步网络很大的误解。请求是可以没有返回的，如果要等待一个返回，那么是十分不明智的设计。因为网络的变数太多，有可能这个数据丢失了或对端有Bug没有发起返回，难道逻辑就要卡在这里吗？对于异步网络编码，正确的设计思路应该是，你来了，我就处理，你不来，我也不操心。任何协议最好没有前因后果的关系，不强求收发顺序，我们编写过程或调用函数时一定有一个返回，但是在网络中不一定需要返回。

回到客户端上来，验证角色之后，如果验证成功，C2LAccountCheckRs协议就不会发送，而是会触发两个事件：

第一，查询角色的事件，这个事件在服务端发生之后会发送一个角色列表L2C_PlayerList的协议到客户端。

第二，通过客户端进入场景，会发送进入地图的S2C_EnterWorld协议。

从输入账号到进入场景中，触发了两次S2C_EnterWorld协议：第一次是验证成功之后，是从Login（登录）场景进入选择Roles（角色）场景；第二次是选择角色之后，进入3D世界的场景。为了让客户端的操作统一，也就是说改变场景必须由S2C_EnterWorld协议来决定，不由客户端代码决定。这样客户端就非常处理好跳转，只需要关注S2C_EnterWorld的变化就可以了。

另一个对客户端比较重要的问题是UI是如何加载出来的。收到S2C_EnterWorld协议后让客户端重新加载了场景。而world.csv文件中，每个场景都可能有一个默认的初始界面，场景加载成功，这些界面就会出现。例如，进入Roles场景时，就要求加载Roles界面。

下面来看Roles界面是如何处理的，代码在UiRoles.cs中：

```
class UiRoles : UiBase {
    public UiRoles() : base(UiType.Roles, 0) { }
    protected override void OnAwake() { }
    protected override bool IsLoaded() { return true; }
    protected override void OnInit() { }
    protected override void OnDestroy() { }
    protected override void OnUpdate() {
        var uiMgr = UiMgr.GetInstance();
        ToUiAccountInfo obj = uiMgr.GetUpdateData<ToUiAccountInfo>
        if (obj == null)
            return;
        if (obj.Version == _lastVersion)
            return;
        _lastVersion = obj.Version;
        if (obj.Players.Count == 0) {
            uiMgr.OpenUi(UiType.RoleCreate);
        } else {
            uiMgr.OpenUi(UiType.RoleSelect);
        }
        CloseThisUi();
    }
}
```

UiRoles是一个空界面，它没有显示任何内容，虽然实现了基类的所有虚函数，但是基本为空，唯一一个实现就是OnUpdate，提供界面跳转的功能。UiRoles不断向UI的数据层请求是否有账号数据AccountInfo的信息，如果有就判断是否有角色，如果有角色就显示角色选择页面，

如果没有角色就显示创建角色页面。至于AccountInfo的信息是怎么来的，显然UiRoles不关心这些，只要有这个数据就行了。实际上，AccountInfo的信息是由L2C_PlayerList协议生成的，当L2C_PlayerList协议到达之后，我们将它解析成了一个AccountInfo类保存到客户端。

```
class AccountInfo : IToUi<ToUiAccountInfo> {
    ...
    protected override void ToUi() {
        ToUiAccountInfo updateObj = new ToUiAccountInfo {Account =
        foreach (var one in _players) {
            ToUiPlayerProperties uiOne = new ToUiPlayerProperties {
                updateObj.Players.Add(uiOne);
            }
        }
        UpdateUiData(updateObj);
    }
}
```

AccountInfo实现了基类IToUi的ToUi函数，将数据推送到了UI的数据层，每个ToUi类都有一个Version（版本号），每次数据发生变更Version值就自增。需要说明的是，UI的数据层维护了一份类似于逻辑数据的结构，这些结构（例如ToUiAccountInfo）和逻辑层的类（例如AccountInfo）有一定的数据重合，但又不完全一致。这些ToUi的类将逻辑层与UI分离开来，而且灵活性非常大。逻辑层的数据发生改变时，如果是ToUi类不关心的数据，ToUi类的数据就不会发生变动，这使得界面不会受到逻辑层的影响。当逻辑层结构中的数据有修改时，Version值增加，界面也会马上收到它改变的数据，并做出相应的修改。

举例说明，假如将玩家的数据放在ToUiPlayer类中，有3个界面需要显示该类中的Level属性。这里要做的就是，在这3个界面的更新函数中都关心ToUiPlayer类的Version变化，不再需要过多的复杂更新操作，而且也不需要取得逻辑层面的Player数据，所有的数据都是由UI层自己维护的。

关于客户端更多的细节这里就不再细说了，感兴趣的读者可以查看源代码。通过客户端，现在我们登录到场景中，但是如果反复登录，就会发现登录不上去，服务端出现一个报错信息，这是因为我们的服务器没有处理断线，后面的章节会整体来看如何处理断线。但在这之前我们

还需要解决一个问题，就是玩家在WorldProxy之间的跳转。

玩家在WorldProxy之间的跳转

在前面的章节中，玩家从Lobby创建了代理地图WorldProxy并成功跳转，玩家的数据也传递到了World地图中。本节要讲的是从WorldProxy跳转到WorldProxy，它与Lobby跳转到WorldProxy是不一样的，原因在于对玩家数据的同步处理。Lobby跳转到WorldProxy的流程图如图11-7所示。

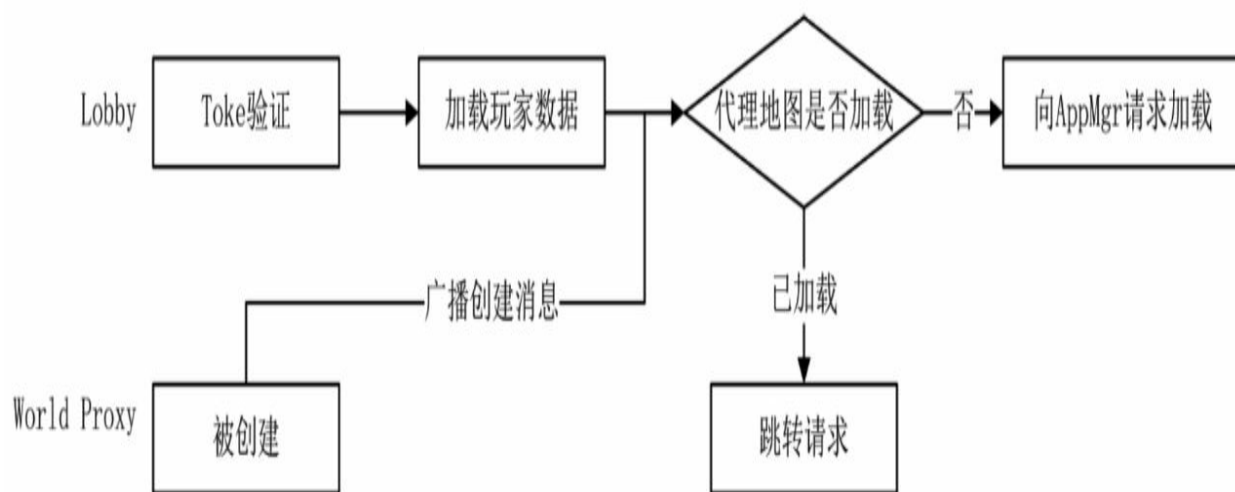


图11-7 Lobby跳转到WorldProxy的流程图

简单来说，玩家从Lobby到WorldProxy的跳转只需要关注目标代理地图是否存在，一旦存在就可以发起跳转，但是在WorldProxy之间却不是这么回事。图11-8展示了WorldProxy之间的跳转流程图。

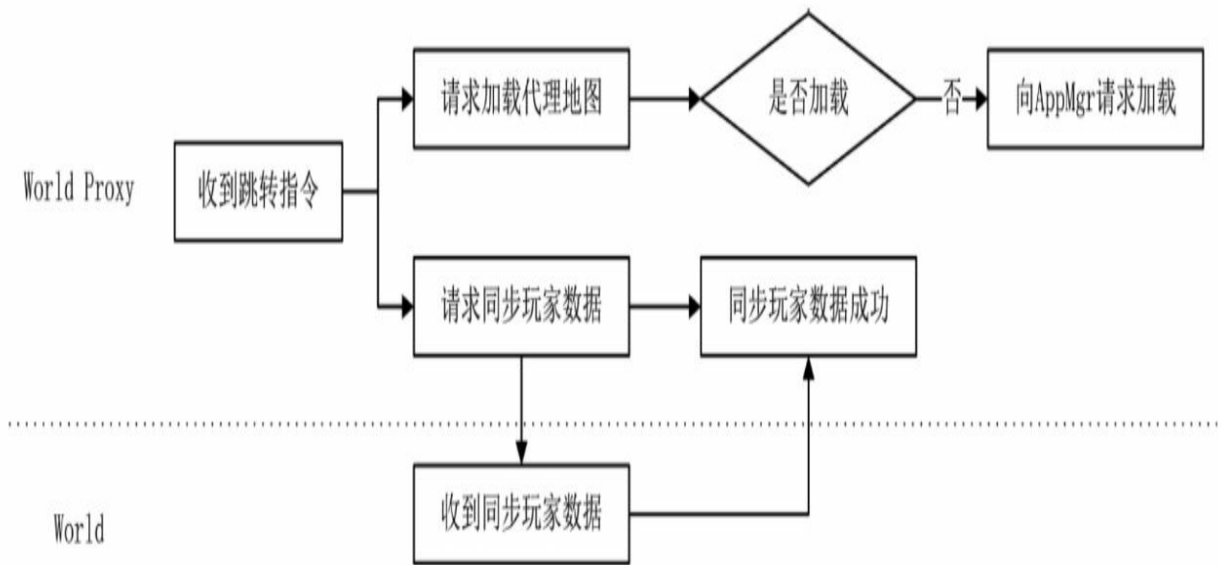


图11-8 WorldProxy之间的跳转流程图

从图11-7和图11-8不难看出这两个流程图的区别，WorldProxy之间的跳转多了一步请求同步玩家数据。为什么呢？作为代理地图，WorldProxy只是一个代理，真正的数据还是在World中，为了取得跳转时的真实数据，必须实时向World请求整个玩家数据，然后跳转到代理地图中，后面的操作就与Lobby跳转到WorldProxy一致了。在同步玩家数据的处理方式上有很多种方案。可以使用Redis数据来进行缓存，Redis数据是可以跨线程读取的。但本书采用的方案是使用协议进行同步，下面来看重点代码。本节的源代码位于11_04_teleport目录中。

跳转部分的代码越来越庞大了，为了隔离跳转部分的代码，本例专门做了一个组件。在game工程中，文件world_component_teleport.h中定义了WorldComponentTeleport组件。

```

class WorldComponentTeleport :public Component<WorldComponentTelep
...
private:
    // <playersn, obj>
    std::map<uint64, TeleportObject*> _objects;
};
  
```

在这个组件中维护了当前代理地图中正在准备跳转的玩家数据，这

些跳转数据保存在TeleportObject结构中。

```
class TeleportObject :public Component<TeleportObject>, public IAw
public:
    TeleportFlag<uint64> FlagWorld;
    TeleportFlag<bool> FlagPlayerSync;
    ...
private:
    int _targetWorldId{ 0 };
    uint64 _playerSn{ 0 };
};
```

在这个结构中有两个重要的数据：一个是标记WorldProxy加载进度的属性值FlagWorld；另一个是标记玩家同步状态的FlagPlayerSync。这两个变量的主要任务是标记创建代理地图和请求玩家数据同步这两个流程处于哪个阶段。因为它们的结构一致，所以采用了一个模板：

```
template<typename T>
struct TeleportFlag {
public:
    TeleportFlagType Flag;
    void SetValue(T value) {
        this->Value = value;
        this->Flag = TeleportFlagType::Completed;
    }
    T GetValue() {
        return this->Value;
    }
    bool IsCompleted() {
        return this->Flag == TeleportFlagType::Completed;
    }
private:
    T Value;
};
```

标记枚举定义如下：

```
enum class TeleportFlagType {
    None = 0,
    Waiting = 1,
```

```
Completed = 2,  
};
```

当有玩家发起一个跳转C2G_EnterWorld协议时，代理地图的处理函数如下：

```
void WorldProxy::HandleC2GEnterWorld(Player* pPlayer, Packet* pPac  
    auto proto = pPacket->ParseToProto<Proto::EnterWorld>();  
    auto worldId = proto.world_id();  
    const auto pResMgr = ResourceHelp::GetResourceManager();  
    const auto pWorldRes = pResMgr->Worlds->GetResource(worldId);  
  
    if (pWorldRes == nullptr)  
        return;  
  
    auto pTeleportComponent = this->GetComponent<WorldComponentTel  
    if (pTeleportComponent->IsTeleporting(pPlayer))  
        return;  
  
    // create teleport object  
    GetComponent<WorldComponentTeleport>()->CreateTeleportObject(w  
}
```

在收到客户端指定跳转C2G_EnterWorld协议时，为玩家创建一个跳转对象TeleportObject实例。

```
void WorldComponentTeleport::CreateTeleportObject(int worldId, Pla  
    const auto pObj = GetSystemManager()->GetEntitySystem()->AddCo  
    _objects.insert(std::make_pair(pPlayer->GetPlayerSN(), pObj));  
    const auto pWorldProxy = GetParent<WorldProxy>();  
    // World  
    CreateWorldFlag(pWorldProxy, worldId, pObj);  
    // Sync  
    CreateSyncFlag(pWorldProxy, pObj);  
}
```

从上面的代码中可以看出，创建TeleportObject对象之后调用了CreateWorldFlag和CreateSyncFlag两个函数，只有这两个功能完成后，跳转地图功能才可能真正开始。先来看玩家同步请求，代码相对简

单:

```
void WorldComponentTeleport::CreateSyncFlag(WorldProxy* pWorldProx
    auto pPlayerMgr = _parent->GetComponent<PlayerCollectorCompone
    const auto pPlayer = pPlayerMgr->GetPlayerBySn(pObj->GetPlayer

    Proto::RequestSyncPlayer protoSync;
    protoSync.set_player_sn(pObj->GetPlayerSN());
    pWorldProxy->SendPacketToWorld(Proto::MsgId::G2S_RequestSyncPl

    pObj->FlagPlayerSync.Flag = TeleportFlagType::Waiting;
}
```

请求玩家数据同步的协议发送到World地图中，同时设置标志位为等待状态，状态在同步数据到达之后会修改为完成状态。同理，在处理代理地图时采用的是相同的方式：

```
void WorldComponentTeleport::CreateWorldFlag(WorldProxy* pWorldPro
    // 创建地图
    const auto pResMgr = ResourceHelp::GetResourceManager();
    const auto pWorldRes = pResMgr->Worlds->GetResource(targetWorld
    if (pWorldRes->IsType(ResourceWorldType::Public)) {
        auto pWorldLocator = ComponentHelp::GetGlobalEntitySystem(
        const auto worldSn = pWorldLocator->GetWorldSnById(targetW
        if (worldSn == static_cast<uint64>(INVALID_ID)) {
            // 向appmgr申请创建地图
            Proto::RequestWorld protoToMgr;
            protoToMgr.set_world_id(targetWorldId);
            MessageSystemHelp::SendPacket(Proto::MsgId::G2M_Reques
            pObj->FlagWorld.Flag = TeleportFlagType::Waiting;
        } else {
            pObj->FlagWorld.SetValue(worldSn);
        }
    } else if (pWorldRes->IsType(ResourceWorldType::Dungeon)) {
        // 马上创建一个副本
        auto pSpaceSyncHandler = ComponentHelp::GetGlobalEntitySys
        AppInfo info;
        if (!pSpaceSyncHandler->GetSpaceApp(&info)) {
            LOG_ERROR("can't find space");
            return;
        }
        Proto::CreateWorld protoCreate;
        protoCreate.set_world_id(targetWorldId);
```



```

        protoCreate.set_last_world_sn(pWorldProxy->GetSN());
        protoCreate.set_game_app_id(Global::GetInstance()->GetCurA
        MessageSystemHelp::SendPacket(Proto::MsgId::G2S_CreateWorld
        pObj->FlagWorld.Flag = TeleportFlagType::Waiting;
    }
}

```

对于公共地图和副本地图采用了不同的方式，公共地图信息向 appmgr 请求，而副本地图是直接向某个 space 请求创建一个地图。无论哪种方式，当地图创建成功时，game 进程的 WorldProxyLocator 会收到广播协议，这个组件会创建一个对应的 WorldProxy，而 WorldProxy 被创建成功之后，会向当前进程或某个指定目标广播，这时原地图就会收到 MI_BroadcastCreateWorldProxy 协议。WorldProxy 对于该协议的处理方式就是转交给组件处理。其代码如下：

```

void WorldProxy::HandleBroadcastCreateWorldProxy(Packet* pPacket)
{
    auto proto = pPacket->ParseToProto<Proto::BroadcastCreateWorld>();
    GetComponent<WorldComponentTeleport>()->HandleBroadcastCreateWorldProxy(proto);
}

void WorldComponentTeleport::HandleBroadcastCreateWorldProxy(const Packet* pPacket)
{
    const auto pResMgr = ResourceHelp::GetResourceManager();
    const auto pWorldRes = pResMgr->Worlds->GetResource(worldId);
    if (pWorldRes->IsType(ResourceWorldType::Public)) {
        // 公共地图:所有等待中的玩家全部跳过去
        do
        {
            auto iter = std::find_if(_objects.begin(), _objects.end(),
                [worldId](const WorldObject* pObj) {
                    return pObj->GetTargetWorldId() == worldId;
                });

            if (iter == _objects.end())
                break;

            auto pObj = iter->second;
            pObj->FlagWorld.SetValue(worldSn);
            Check(pObj);
        } while (true);
    } else if (pWorldRes->IsType(ResourceWorldType::Dungeon)) {
        // 非公共地图，一次只有一个玩家跳过去
        auto iter = std::find_if(_objects.begin(), _objects.end(),
            [worldId](const WorldObject* pObj) {
                return pObj->GetTargetWorldId() == worldId;
            });
    }
}

```

```

// 副本是定向协议，如果没有找到，就一定有Bug
if (iter == _objects.end()) {
    LOG_ERROR("BroadcastCreateWorldProxy, can't find telep
    return;
}

auto pObj = iter->second;
pObj->FlagWorld.SetValue(worldSn);
Check(pObj);
}
}

```

收到WorldProxy创建成功的消息，修改对应的TeleportObject的值。最后调用Check函数检查整个TeleportObject的状态。

```

bool WorldComponentTeleport::Check(TeleportObject* pObj) {
    const auto worldId = pObj->GetTargetWorldId();
    const auto pWorldProxy = GetParent<WorldProxy>();

    if (!pObj->FlagPlayerSync.IsCompleted() || !pObj->FlagWorld.Is
        return false;

    //所有准备工作已完成
    auto pPlayerMgr = _parent->GetComponent<PlayerCollectorCompone
    const auto pPlayer = pPlayerMgr->GetPlayerBySn(pObj->GetPlayer

    // teleport
    WorldProxyHelp::Teleport(pPlayer, pWorldProxy->GetSN(), pObj->

    // 清理数据
    _objects.erase(pPlayer->GetPlayerSN());
    GetSystemManager()->GetEntitySystem()->RemoveComponent(pObj);
    return true;
}

```

当所有工作都完成时，调用WorldProxyHelp::Teleport函数发起正式跳转。现在测试一下工作，进入11_04_teleport目录，启动allinone进程和robots进程，在robots进程中输入“login-a test”指令。这时，玩家登录服务器，依次收到了S2C_EnterWorld协议，第一次是进入2号地图，第二次是进入3号地图。2号地图是roles地图，用于选择角色，3号地图是默认的公共地图。

这时，在robots控制台输入“world-enter 4”，表示想让玩家进入4号地图。在world.csv配置文件中，可以看到4号地图是一个副本地图。指令执行之后，robots控制台会打印“account: test enter world.id: 4”，表示该角色已进入4号地图。在allinone控制台输入“world-all”和“proxy-all”，我们可以看到如下信息：

```
proxy -all
**** world proxy gather ****
sn:103848004384982099 proxy sn:103848004384982099 online:0 name:ro
sn:103848004720527189 proxy sn:103848004720527189 online:1 name:ji
sn:103848011766958690 proxy sn:103848011766958690 online:1 name:du

world -all
**** world gather ****
sn:103848004720527189 online:1
sn:103848011766958690 online:1
```

玩家已经从公共地图转移到了副本地图。

11.6

总结

本章讨论了分布式中很重要的一个环节，玩家如何在多进程之间切换地图。切换地图的功能非常复杂，涉及同步数据、存储数据等操作。在切换地图时，大部分情况下是在WorldProxy代理类之间进行切换的。

完成地图切换之后，主要是玩家与World之间的交互，因为World地图位于space进程中，这时位于game进程上的WorldProxy的主要作用就是转发协议。

第12章

断线与动态加载系统

本章主要讨论一个异常处理——网络断线和重连。其中断线又包括两个部分：一个是玩家的断线；另一个是服务端进程之间的断线。这两种情况需要分开处理。除此之外，本章在现有框架的基础之上介绍可以动态加载System的功能，这样可以在不修改底层libserver的情况下自主加载新的系统。本章包括以下内容：

- 梳理各个进程对于断线的处理。
- 在space进程中加入一个新的移动系统。

玩家断线

玩家断线是一种正常的网络断线。在整个游戏框架中，玩家数据会保存在3个进程和4种实体中。3个进程分别是login进程、game进程和space进程。当玩家断线时，要充分考虑到3个进程中的数据，相关数据都需要处理。

4种实体分别如下：

(1) 在login中，实体对象为Account类，该类用于玩家进行账号验证。

(2) 在game进程中，有两个与玩家有关的实体类：

- Lobby类，该类是玩家进入game进程的第一个实体类。
- WorldProxy类，该类是地图类的代理类。

(3) 在space进程中，玩家存在于一个特定的World类中，该类是真正的地图类。

在以上4种实体类中，我们都必须处理玩家断线的问题。当一个网络连接中断时，网络底层会发送MI_NetworkDisconnect协议给各个线程。也就是说，只需要在这4个实体类中处理好MI_NetworkDisconnect的后续操作就可以了。本节涉及的有关源代码位于12_01_disconnect目录中。

12.1.1 玩家在login进程中断线

当玩家还没有进入game进程之前，是与login进行通信的，其主要数据位于Account类中。下面的代码是Account类对于断线协议做出的反应：

```
void Account::HandleNetworkDisconnect(Packet* pPacket) {
    const auto socketKey = pPacket->GetSocketKey();
    if (socketKey->NetType != NetworkType::TcpListen)
        return;
    auto pPlayerCollector = GetComponent<PlayerCollectorComponent>
    pPlayerCollector->RemovePlayerBySocket(pPacket-> GetSocketKey()
}
```

收到断线协议后，Account类会将玩家的数据进行销毁，附加在Player实体上的组件会将玩家在Redis中的数据也销毁。在login进程中，除了有TCP连接外，还有HTTP连接。HTTP连接用于玩家向外部Nginx服务器请求账号验证时使用。在断线处理时，需要判断当前断线Socket的类型，只有TCP类型的断线才进行处理。

12.1.2 玩家在game进程中断线

当玩家进入game进程时，可能存在于两个实体中：一个是Lobby；另一个是WorldProxy。在这两个类中，框架都做了断线处理。

```
void Lobby::HandleNetworkDisconnect(Packet* pPacket) {
    auto pTagValue = pPacket->GetTagKey()->GetTagValue(TagType::Ac
    if (pTagValue == nullptr)
        return;
    GetComponent<PlayerCollectorComponent>()->RemovePlayerBySocket
}
```

在Lobby类中处理得比较简单，只需要移除玩家数据即可。在WorldProxy类中相对要多几个步骤，下面是它的实现：

```
void WorldProxy::HandleNetworkDisconnect(Packet* pPacket) {
    TagValue* pTagValue = pPacket->GetTagKey()->GetTagValue (TagTy
    if (pTagValue != nullptr) {
        const auto pPlayer = GetComponent<PlayerCollectorComponent
        if (pPlayer == nullptr)
            return;

        auto pCollector = GetComponent<PlayerCollectorComponent>()
        pCollector->RemovePlayerBySocket(pPacket->GetSocketKey()->
        SendPacketToWorld(Proto::MsgId::MI_NetworkDisconnect, pPla
    }
    ...
}
```

从上面的代码中可以看到，当一个玩家的网络中断之后，WorldProxy代理地图类将玩家数据移除，同时需要向space进程中的World发送一个断线消息，告诉World类有一个玩家断线了。

12.1.3 玩家断线时World类的处理

以下代码展示了space进程收到断线消息所进行的处理：

```
void World::HandleNetworkDisconnect(Packet* pPacket) {
    auto pTags = pPacket->GetTagKey();
    const auto pTagPlayer = pTags->GetTagValue(TagType::Player);
    if (pTagPlayer != nullptr) {
        auto pPlayerMgr = GetComponent<PlayerManagerComponent>();
        const auto pPlayer = pPlayerMgr->GetPlayerBySn(pTagPlayer->
        if (pPlayer == nullptr) {
            LOG_ERROR("world. net disconnect. can't find player. p
            return;
        }

        Proto::SavePlayer protoSave;
        protoSave.set_player_sn(pPlayer->GetPlayerSN());
        pPlayer->SerializeToProto(protoSave.mutable_player());
        MessageSystemHelp::SendPacket(Proto::MsgId::G2DB_SavePlaye

        // 玩家掉线
        pPlayerMgr->RemovePlayerBySn(pTagPlayer->KeyInt64);
    }
    ...
}
```

当World收到玩家断线的消息后，它需要做两件事：第一件事是将内存中玩家的数据移除，将其从PlayerManagerComponent管理类中移除；第二件事是在移除之前，将玩家的数据发送到dbmgr进程中进行数据存储，发送的协议号为G2DB_SavePlayer。

12.1.4 玩家数据的读取与保存

当玩家下线时，需要从World对象中移除玩家，并对当前数据进行存储，存储玩家数据时使用Proto: : SavePlayer协议结构。其定义如下：

```
message Player {
    uint64 sn = 1;
    string name = 2;
    PlayerBase base = 3;
    PlayerMisc misc = 4;
}
```

它的结构很简单，两个变量，一个64位无符号整型数据作为玩家唯一标识，另一个是结构Proto: : Player，用于存储玩家当前所有数据。从DB中加载一个数据时，也是使用的Proto: : Player结构，该结构定义在db.proto文件。在读取与存储玩家数据时，保持了一致的代码结构。下面来看Protobuf中对于Player的定义：

```
message Player {
    uint64 sn = 1;
    string name = 2;
    PlayerBase base = 3;
    PlayerMisc misc = 4;
}
```

Proto: : Player是DB存储的基本单元，结构PlayerBase中包括一些基础数据，结构PlayerMisc中包括一些杂项数据，如果要增加道具数据，那么可以再增加一个PlayerItems结构。当玩家登录game进程之后，在Lobby类中会生成一个Player实例，game进程会读取出玩家选中的角色，并从DB中取读Proto: : Player传递到Player实体中。其读取代码如下：

```
void Player::ParserFromProto(const uint64 playerSn, const Proto::P
    _playerSn = playerSn;
    _player.CopyFrom(proto);
```

```

    _name = _player.name();

    // 在内存中修改数据
    for (auto pair : _components) {
        auto pPlayerComponent = dynamic_cast<PlayerComponent*>(pair.second);
        if (pPlayerComponent == nullptr)
            continue;
        pPlayerComponent->ParserFromProto(proto);
    }
}

```

对于附加在Player实体上的组件，如果有存储或读取数据库的需要，都是基于PlayerComponent接口的，要实现两个虚函数。PlayerComponent定义如下：

```

class PlayerComponent {
public:
    virtual void ParserFromProto(const Proto::Player& proto) = 0;
    virtual void SerializeToProto(Proto::Player* pProto) = 0;
};

```

Player类解析了Proto::Player的结构，并将它传递到自己所有的组件上，让组件选择自己需要的数据来填充内存数据。例如PlayerComponentLastMap组件，用于分析最后一次登录地图的数据。它的ParserFromProto函数实现如下：

```

void PlayerComponentLastMap::ParserFromProto(const Proto::Player& proto) {
    // 公共地图
    auto protoMap = proto.misc().last_world();
    int worldId = protoMap.world_id();
    auto pResMgr = ResourceHelp::GetResourceManager();
    auto pMap = pResMgr->Worlds->GetResource(worldId);
    if (pMap != nullptr) {
        _pPublic = new LastWorld(protoMap);
    } else {
        pMap = pResMgr->Worlds->GetInitMap();
        _pPublic = new LastWorld(pMap->GetId(), 0, pMap->GetInitPos());
    }
    ...
}

```

当Proto: : Player数据传递到PlayerComponentLastMap组件时，它从中挑选了自己感兴趣的PlayerMisc数据。又如space进程中的PlayerComponentDetail组件，这个组件存储着玩家的基础数据，诸如level、gender等常用数据，其ParserFromProto函数实现如下：

```
void PlayerComponentDetail::ParserFromProto(const Proto::Player& p
    auto protoBase = proto.base();
    _gender = protoBase.gender();
}
```

从上面的代码可以看出，PlayerComponentDetail组件没有对PlayerMisc进行处理，它只对PlayerBase感兴趣。总之，当Proto: : Player传递到Player实体上时需要遍历所有组件，填充组件需要的数据。

还有另一种情况，Player已经存在了，也解析过Proto: : Player结构了，这时加入组件，初始化工作将由Awake组件来完成。以PlayerComponentLastMap组件为例：

```
void PlayerComponentLastMap::Awake() {
    Player* pPlayer = dynamic_cast<Player*>(_parent);
    ParserFromProto(pPlayer->GetPlayerProto());
}
```

在组件初始化时，主动调用了ParserFromProto函数对数据进行分析。通过以上两步将DB中的数据加载到内存中。除了读取之外，还需要关注玩家身上的组件是如何进行数据存储的。当玩家断线时，调用以下4行代码来实现保存：

```
Proto::SavePlayer protoSave;
protoSave.set_player_sn(pPlayer->GetPlayerSN());
pPlayer->SerializeToProto(protoSave.mutable_player());
MessageSystemHelp::SendPacket(Proto::MsgId::G2DB_SavePlayer, proto
```

看上去很简洁，其中值得注意的是SerializeToProto。下面的代码是它的实现：

```
void Player::SerializeToProto(Proto::Player* pProto) const {
    // 基础数据
    pProto->CopyFrom(_player);
    // 在内存中修改数据
    for (auto pair : _components) {
        auto pPlayerComponent = dynamic_cast<PlayerComponent*>(pair.second);
        if (pPlayerComponent == nullptr)
            continue;
        pPlayerComponent->SerializeToProto(pProto);
    }
}
```

当需要保存数据时，也是遍历玩家的所有组件，让组件把自己的数据传递到给定的参数Proto::Player中。还是以PlayerComponentLastMap组件为例，它的SerializeToProto函数实现如下：

```
void PlayerComponentLastMap::SerializeToProto(Proto::Player* pProto) const {
    if (_pPublic != nullptr) {
        const auto pLastMap = pProto->mutable_misc()->mutable_last_map();
        _pPublic->SerializeToProto(pLastMap);
    }
    if (_pDungeon != nullptr) {
        const auto pLastDungeon = pProto->mutable_misc()->mutable_last_dungeon();
        _pDungeon->SerializeToProto(pLastDungeon);
    }
}
```

PlayerComponentLastMap组件主要是保存上一次登录的地图信息，它将自己的内存数据（也就是最近的登录地图数据）写入了给定的Proto::Player中，这样该角色下次上线时取到的数据就是下线时最后的数据。

总之，在Player组件的SerializeToProto函数中，将内存的数据重新写回到Proto::Player中，再将它传递到dbmgr进程实现存储。

分析完玩家的整个断线过程可能发生的事件，现在编译12_01_disconnect工程，使用robots进行登录。登录完成之后，在allinone控制台输入“proxy-all”和“world-all”，会发现有一个玩

家在公共地图上。随后将robots进程关闭，让玩家下线。如果是通过客户端登录的，就关闭客户端。退出客户端，服务端会打印如下两行数据：

```
[INFO] - HandleSavePlayer sn:103088522096807185
[DEBUG] - @@ DEL. key:del engine::online::game::test
```

这两行数据对应两个事件：一个是存储；另一个是Redis在线标志被删除。再使用“proxy-all”和“world-all”查看地图上的数据，玩家数据已经从地图上被清除了，以上就是整个玩家断线过程的处理。

12.1.5 如何进入断线之前的地图

当一个玩家进入game进程时，第一个到达的地方一定是Lobby类中，这是一个中转站。Lobby类从数据库中读出了玩家的数据，其处理函数为Lobby::HandleQueryPlayerRs。在12_01_disconnect目录的工程中查看该函数。该函数从玩家数据中取出最近登录的副本地图，以保证玩家优先进入之前被中断的副本。如果副本在本地不存在，就向appmgr进行查询，查询时将玩家放在等待队列中。下面来看代码实现：

```
void Lobby::HandleQueryPlayerRs(Packet* pPacket) {
    auto protoRs = pPacket->ParseToProto<Proto::QueryPlayerRs>();
    auto account = protoRs.account();
    auto pPlayer = GetComponent<PlayerCollectorComponent>()->GetPl
    ...
    // 分析进入地图
    auto protoPlayer = protoRs.player();
    const auto playerSn = protoPlayer.sn();
    pPlayer->ParserFromProto(playerSn, protoPlayer);
    const auto pPlayerLastMap = pPlayer->AddComponent <PlayerCompo
    auto pWorldLocator = ComponentHelp::GetGlobalEntitySystem()->
    // 进入副本
    auto pLastMap = pPlayerLastMap->GetLastDungeon();
    if (pLastMap != nullptr) {
        if (pWorldLocator->IsExistDungeon(pLastMap->WorldSn)) {
            // 存在副本，跳转
            WorldProxyHelp::Teleport(pPlayer, GetSN(), pLastMap->W
            return;
        }
        // 查询副本是否存在
        if (_waitingForDungeon.find(pLastMap->WorldSn) == _waiting
            _waitingForDungeon[pLastMap->WorldSn] = std::set<uint6
        }
        if (_waitingForDungeon[pLastMap->WorldSn].empty()) {
            // 向appmgr查询副本
            Proto::QueryWorld protoToMgr;
            protoToMgr.set_world_sn(pLastMap->WorldSn);
            protoToMgr.set_last_world_sn(GetSN());
            MessageSystemHelp::SendPacket(Proto::MsgId::G2M_QueryW
        }
        _waitingForDungeon[pLastMap->WorldSn].insert (pPlayer->Get
        return;
    }
    // 进入公共地图
```

```
EnterPublicWorld(pPlayer);  
}
```

向appmgr进程进行副本查询之后，如果副本在space中存在，就会在本地创建一个WorldProxy代理地图进行跳转。多进程启动时，存在多个game进程，有可能玩家第二次登录的game进程并不是之前那个game进程，也就不存在WorldProxy。虽然不存在，但可以创建一个代理。这引出了另一个问题，WorldProxy是何时被销毁的呢？

理论上来说，当World在某种条件下被销毁了，应该广播一个销毁协议，这个销毁协议会发送到所有game进程和appmgr进程上，以清除这个World当前的代理数据。appmgr的数据被清除之后，玩家登录时向appmgr请求副本地图时会返回失败，这时就会选择最近的公共地图进行登录。在本小节的示例中，我们没有完成这一步，读者如果感兴趣，那么可以自行上手实现。

对于服务器来说，除了玩家断线之外，还有一种更为复杂的断线是服务进程之间的断线。

进程之间的断线

在服务端，大部分进程都在内网。除了宕机之外，很少会出现断线的问题。但是我们依然要处理断线重连。

其中一个原因是多进程启动的顺序不是固定的。如果一定要按一个固有的顺序来启动服务器的所有进程，那么显然不够灵活。既然进程的启动不是先启动appmgr，再启动game，对于game进程来说，它启动时就需要连接appmgr，如果它启动时appmgr还没有启动，这里就相当于一个断线重连的情况。下面依次来梳理每个进程断线时可能发生的事情。

12.2.1 login进程断线与重连

工程login的断线处理相对来说比较简单，打开login工程的main.cpp文件，可以看到与login有网络连接的是dbmgr进程和appmgr进程。每个login进程需要向appmgr进程同步自己当前的状态。一个login断线了，在appmgr中的状态就需要清除。下面的代码是网络断线的流程，在appmgr中的处理如下：

```
void AppSyncComponent::HandleNetworkDisconnect(Packet* pPacket) {
    if (!NetworkHelp::IsTcp(pPacket->GetSocketKey()->NetType))
        return;
    SyncComponent::HandleNetworkDisconnect(pPacket);
    ...
}

void SyncComponent::HandleNetworkDisconnect(Packet* pPacket) {
    SOCKET socket = pPacket->GetSocketKey()->Socket;
    const auto iter = std::find_if(_apps.begin(), _apps.end(), [&s
        return pair.second.Socket == socket;
    ]);
    if (iter == _apps.end())
        return;
    _apps.erase(iter);
}
```

在appmgr中，login的数据关系到我们登录时请求登录IP的功能。清除login的数据，这样玩家上线时就不会请求到已断线的login进程数据。现在来测试一下，启动appmgr，再启动login，先后顺序并没有关系。login的启动命令如下：

```
[root@localhost bin]# ./logind -sid=101
```

在appmgr的控制台输入“app-info”，查看当前所有进程的状态：

```
app -info
[DEBUG] - appId: 101 type:login online:0
[DEBUG] - appId: 201 type:game online:0
[DEBUG] - appId: 301 type:space online:0
```

其中，online表示在线人数，客户端向appmgr发起请求登录地址时，就是根据当前所有login进程的online数据决定被分配到哪个login进程上去的。关闭login进程之后，其信息在appmgr进程被销毁。再次连接之后，又会重新同步状态。

12.2.2 game进程断线与重连

如果game进程发生断线，那么与它有联系的所有进程都需要做出反应。在game工程的main.cpp文件中，可以看到与其有连接的进程是dbmgr、appmgr和space。game连接dbmgr是为了读取数据，连接appmgr是为了创建公共地图。因此，这两个进程都不需要对game进程的断线做出什么特别的操作，就剩下最后一个space进程。

game是玩家与space的中间进程，当game进程发生宕机或其他事件引起的断线时，在game进程上的所有玩家网络全部中断。对于space来说，需要做的就是检查自己的每一个地图实例，与断线game进程有关联的玩家全部踢下线并保存。

当有网络断线时，所有World实例都会收到MI_NetworkDisconnect断线消息。本小节的源代码还是在12_01_disconnect目录中。下面的代码是World对于断线消息的处理函数：

```
void World::HandleNetworkDisconnect(Packet* pPacket) {
    auto pTags = pPacket->GetTagKey();
    const auto pTagPlayer = pTags->GetTagValue(TagType::Player);
    if (pTagPlayer != nullptr) {
        // 玩家掉线
        ...
    } else {
        // dbmgr, appmgr or game断线
        const auto pTagApp = pTags->GetTagValue(TagType::App);
        if (pTagApp != nullptr) {
            auto pPlayerMgr = GetComponent<PlayerManagerComponent>
            pPlayerMgr->RemoveAllPlayers(pPacket);
        }
    }
}
```

一旦发现收到的断线来自于game进程，与这个game有关的所有玩家将被从PlayerManagerComponent管理组件中踢出，在踢出的过程中执进行了保存数据的操作。其代码如下：

```
void PlayerManagerComponent::RemoveAllPlayers(NetIdentify* pNetIde
```

```

auto iter = _players.begin();
while (iter != _players.end()) {
    auto pPlayer = iter->second;
    if (pPlayer->GetSocketKey()->Socket != pNetIdentify->GetSo
        ++iter;
        continue;
    }
    iter = _players.erase(iter);
    // save
    Proto::SavePlayer protoSave;
    protoSave.set_player_sn(pPlayer->GetPlayerSN());
    pPlayer->SerializeToProto(protoSave.mutable_player());
    MessageSystemHelp::SendPacket(Proto::MsgId::G2DB_SavePlaye
    // remove obj
    GetSystemManager()->GetEntitySystem()->RemoveComponent(pPl
}
}

```

综上所述，game进程网络断开，space进程上所有地图的World实例都会收到断线消息。在World处理该消息时，将断线协议中的Socket值与玩家身上的Socket值进行对比，找到这些Socket值相同的玩家，让这些玩家下线，同时保存玩家的数据。

12.2.3 space进程断线与重连

space进程断线的情况比game进程稍微复杂一些。space还连接了dbmgr和appmgr。space与dbmgr的连接断开了不会有什么逻辑上的问题，因为很快会重新连接，也不会有存储上的问题。下面先讨论与game的断开。space与game的连接是由game进程发起的。space断线，game进程中的WorldProxy必须做出反应，处理函数如下：

```
void WorldProxy::HandleNetworkDisconnect(Packet* pPacket) {
    if (!NetworkHelp::IsTcp(pPacket->GetSocketKey()->NetType))
        return;

    TagValue* pTagValue = pPacket->GetTagKey()->GetTagValue(TagType)
    if (pTagValue != nullptr) {
        // 玩家掉线
        ...
    } else {
        // 可能是space、login、appmgr、dbmgr断线
        auto pTags = pPacket->GetTagKey();
        const auto pTagApp = pTags->GetTagValue(TagType::App);
        if (pTagApp == nullptr)
            return;
        const auto appKey = pTagApp->KeyInt64;
        const auto appType = GetTypeFromAppKey(appKey);
        const auto appId = GetIdFromAppKey(appKey);

        if (appType != APP_SPACE || _spaceAppId != appId)
            return;

        // 玩家需要全部断线
        auto pPlayerCollector = GetComponent<PlayerCollectorCompon
        pPlayerCollector->RemoveAllPlayerAndCloseConnect();

        // locator
        auto pWorldLocator = ComponentHelp::GetGlobalEntitySystem(
        pWorldLocator->Remove(_worldId, GetSN());

        // worldproxy 需要销毁
        GetSystemManager()->GetEntitySystem()->RemoveComponent(thi
    }
}
```

在game进程中，WorldProxy代理类的目标World可能位于各个space进程上，所以WorldProxy类收到断线消息，需要判断是否是自己代理的space进程断线了。如果是，当前代理地图中的玩家就全部下线，同时销毁WorldProxy自己。这是比较简单的一种做法。复杂一点的话，可以让所有玩家回到默认地图中去。

当space断线时，除了game进程需要处理外，在appmgr中保存了一些地图实例与space的对应数据，这些数据也必须在断线时处理掉，处理函数位于appmgr工程的CreateWorldComponent组件中，实现如下：

```
void CreateWorldComponent::HandleNetworkDisconnect(Packet* pPacket
...
auto appId = GetIdFromAppKey(pTagApp->KeyInt64);
// 断线的Space上是否有正在创建的地图
do
{
    auto iterCreating = std::find_if(_creating.begin(), _creat
        return pair.second == appId;
    });
    if (iterCreating == _creating.end())
        break;
    // 正在创建时，Space进程断开了，另找进程创建
    auto worldId = iterCreating->first;
    _creating.erase(iterCreating);
    ReqCreateWorld(worldId);
} while (true);
// 断线的Space上有已创建的公共地图全部删除
do
{
    auto iterCreated = std::find_if(_created.begin(), _created
        return Global::GetAppIdFromSN(pair.second) == appId;
    });
    if (iterCreated == _created.end())
        break;
    _created.erase(iterCreated);
} while (true);
// 断线的Space上创建的副本地图全部删除
do
{
    const auto iter = std::find_if(_dungeons.begin(), _dungeon
        return pair.second == appId;
    });
    if (iter == _dungeons.end())
        break;
```

```
        _dungeons.erase(iter);  
    } while (true);  
}
```

现在来做一个测试，以多进程启动服务端，启动5个进程：

```
./dbmgrd  
./appmgrd  
./logind -sid=101  
./gamed -sid=201  
./spaced -sid=301
```

启动robots用于登录。在robots上登录一个账号，完成之后，在appmgr上输入“create-all”命令，可以查看当前所有的地图以及所在space的信息。

在space进程下，按Ctrl+C组合键或者输入“-exit”退出进程。此时，在appmgr上输入“create-all”命令就会看到地图信息，会发现被销毁了，就是上面CreateWorldComponent类处理断线的代码让它销毁的。

同时，game进程中的代理地图实例被销毁，在robots进程中的Robot从地图中弹了出来，它会再次申请进入game中，但是因为没有space进程为它创建真实地图，所以Robot停在了game进程的Lobby类中，在客户端的表现就是它停在了选择角色界面。假设这时有两个space进程，一个断线，另一个正常的space进程会马上创建一个新的公共地图实例，而玩家即使被弹出了地图，再次登录时也可以马上重新进入游戏。

以上流程充分展示了分布式框架的特点，即使遇到严重问题，产生了宕机，但是由于服务端多进程分散处理，对于玩家来说，他可能以为自己断线，重新登录之后可以重新进入游戏中，丝毫不会减少游戏体验。分布式框架比起单进程的服务框架，在容灾方面要更有利。

12.2.4 appmgr 进程断线与重连

如果appmgr断线了或者宕机了，会发生什么事情呢？appmgr需要解决两个问题：一个是全局共有数据；另一个是HTTP请求。

重启appmgr后，所有连接它的进程会将自己的玩家在线情况发送过来，但appmgr的问题并不是重启那么简单。在前面的功能中，appmgr不仅负责维护公共地图所在的space信息，还负责维护副本地图的space信息。这些信息在appmgr断线之后变成一片空白，现在需要考虑如何重建这些数据。有以下两种思考模式：

(1) space进程发现自己和appmgr连接上之后，马上向它推送一个协议，这个协议中包括space当前所有的地图信息。但这也有一个问题，当appmgr被重启了，在space还没有向appmgr发送同步地图的信息之前，game进程向appmgr请求某个公共地图的信息，这时appmgr是应该创建还是应该等待呢？

(2) 如果事情变得如(1)一样复杂，我们应该考虑另一个问题，appmgr上是否真的需要保存这些数据？如果不保存，那么game进程应该如何知道某个公共地图的实例在哪里呢？这个问题有很多解决方案，例如将数据推送到Redis中，space中的数据就不必一定保存到appmgr上。同时，我们还可以省略每个进程中的采集数据的流程。

纵观整个框架的布局，除了appmgr和dbmgr之外，都有可以替代的方案，game1关闭了还有game2，space1宕机了还有space2。如果appmgr宕机了，就没有可以替代的方案了。

实际上，appmgr同样可以采用集合的方式，这样即使其中一个appmgr关闭了，还有另一个appmgr。如果读者感兴趣，那么可以沿着这个思路继续编码，鉴于篇幅所限，本书不再继续深入介绍。

一个分布式框架的较佳方案是，所有节点都是N+1个实例，这意味着每一个节点关闭时，立即有另一个节点替代继续工作。

在现在的工程中，appmgr只有一个唯一实例，如果它宕机了，那么新来的客户端请求登录地址时肯定会出现连接不上的问题，这显然是不太好的体验。在框架中，appmgr收集了login的数据，客户端通过HTTP

请求到一个轮询的login进程数据用于登录。HTTP请求返回JSON类型的数据，内容包括客户端可以登录的login地址。

在实际部署中，我们可以采取另一种方式来实现，就是充分使用Nginx的upstream功能。首先在login进程上实现一个HTTP接口，返回自己的IP与端口，也就是客户端登录的IP与端口。将login打开的HTTP端口配置到Nginx上，例如：

```
Upstream login_server {  
    server 192.168.0.172:9000 weight=5;  
    server 192.168.0.171:9000 weight=5;  
}
```

当我们访问Nginx时，Nginx会以轮询的方式分发客户端请求到每一个login进程上，如图12-1所示。

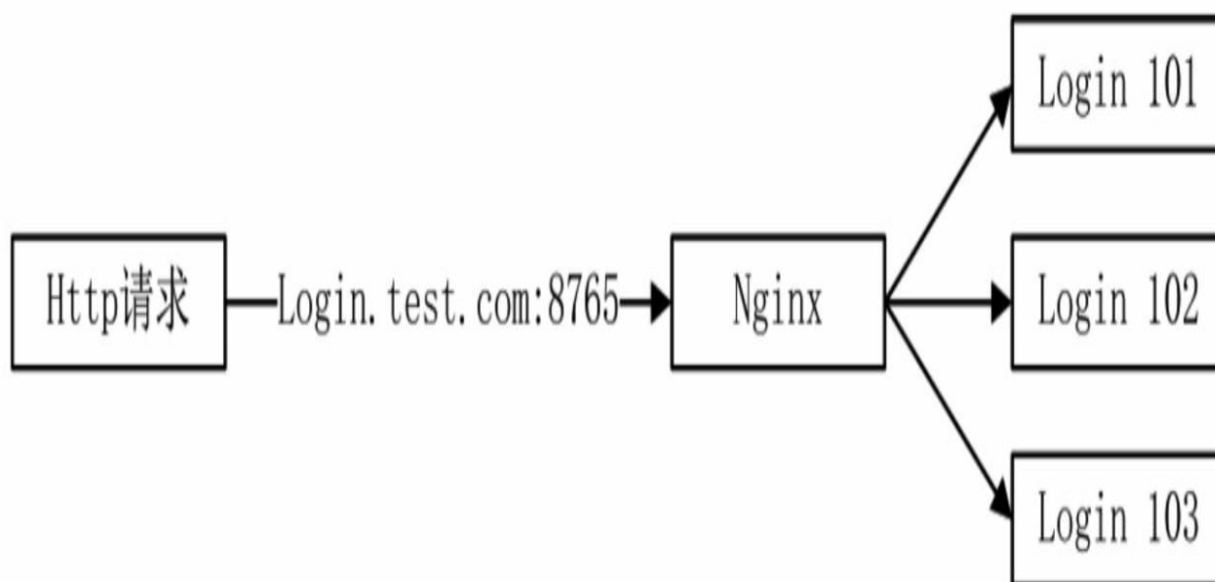


图12-1 反向代理均衡

对于客户端来说，它访问的只是一个指向Nginx的域名，而Nginx会自行对后端服务器加权轮询。这种架构叫反向代理负载均衡，这种部署将appmgr的均衡工作移到了外部的Nginx进行处理。Nginx有检测的机制，如果103服务器宕机了，就会自动从轮询中剔除。

动态新增系统

在前面的示例中，基本框架搭建完成，游戏逻辑在此之上增加新的组件即可。但考虑到一种情况，我们的ECS框架关于System系统的部分几乎固定在了底层，如果上层有需求新增一个系统，就只能修改底层。在游戏制作的过程中会不断地加入新系统，但这些系统和框架libserver之间是无关的。

本节是本书框架的深入应用，即如何动态新增系统。本节以直观的移动功能为例进行介绍。在本节的示例中，在space进程上增加一个移动的系统。

本节的源代码位于12_02_move目录中。首先，我们来看想要达到的最终效果是什么。打开12_02_move目录，编译并执行allinone。用Unity打开客户端工程，注意修改相应的配置，编出一个PC版本。同时启动两个客户端，以两个账号登录。在任何一个客户端单击地图，都可以看到角色的移动，在另一个客户端对该角色进行了移动同步。

要实现移动功能，首先客户端需要发送一条移动协议，在处理移动协议时可以分成两种情况：一种情况是给一个目标点，让人物移动到目标点；另一种情况是给一个移动方向。本节的例子中将要讨论第一种情况。如果读者有游戏编码的经验，那么应该知道常用的处理移动的方式：当玩家收到这个移动协议之后，将这个数据保存在玩家对象中，然后每一帧对其移动位置进行计算与调整。伪代码如下：

```
class player {
private:
    std::list<Vector3> pos;
public:
```

```
void HandleMove(Packet* pPacket){
    This->pos = ... // 收到移动协议，初始化
}
void Update(){
    if (IsMove()){
        this->curpos = ... // 计算移动点
    }
}
}
```

我们来看如何改造这个流程，要牢记新框架的基本特点——解耦。如果所有数据都堆积在Player类中，就有太多且杂乱数据。因此，用一个组件来存储移动数据。

12.3.1 MoveComponent组件

新组件的名字为MoveComponent，可以在space工程中找到它。收到客户端传来的C2S_Move移动协议时，通过game进程的WorldProxy，协议将被中转到指定的World实例上。在处理协议时，将移动数据全部存到MoveComponent组件中，Player类就再不需要关心了。在本例中处理移动时，要求客户端发送从起始位置到终点位置的所有坐标点。客户端使用了Unity的寻路NavMeshAgent组件，当我们在屏幕上点下一个点时，NavMeshAgent可以计算出从玩家当前位置到目标点上所有路径的关键位置，是由Unity的寻路算法计算出来的两点间的最短路径。将路径传给服务端，让服务端以相同的速度计算出玩家的位移情况。在World类中，收到移动消息的处理函数如下：

```
void World::HandleMove(Player* pPlayer, Packet* pPacket) {
    auto proto = pPacket->ParseToProto<Proto::Move>();
    proto.set_player_sn(pPlayer->GetPlayerSN());
    const auto positions = proto.mutable_position();
    auto pMoveComponent = pPlayer->GetComponent<MoveComponent>();
    if (pMoveComponent == nullptr) {
        pMoveComponent = pPlayer->AddComponent<MoveComponent>();
    }
    std::queue<Vector3> pos;
    for (auto index = 0; index < proto.position_size(); index++) {
        Vector3 v3(0, 0, 0);
        v3.ParserFromProto(positions->Get(index));
        pos.push(v3);
    }
    const auto pComponentLastMap = pPlayer->GetComponent<PlayerComponent>();
    pMoveComponent->Update(pos, pComponentLastMap->GetCur()->Posit);
    BroadcastPacket(Proto::MsgId::S2C_Move, proto);
}
```

协议中的位置数据存储了从原点到终点之间路径上的所有点。相邻的两个点之间是没有阻碍的，也就是说相邻的两个点可以走直线。在本例中，没有在服务端判断客户端的路径是否正确，完全信任了客户端。当收到一个移动协议之后，做了一个全地图广播，也就是将收到的移动数据广播给本地图所有的玩家，并将数据保存到MoveComponent组件中，以下是组件的数据定义：

```
class MoveComponent :public Component<MoveComponent>, public IAwake
...
private:
    std::queue<Vector3> _targets;
    MoveVector3 _vector3;
};
```

现在组件有了，那么它是如何运行起来的呢？按照前面的例子，如果玩家要计算出每帧的位移，就需要有一个Update帧函数来实时计算。如果有1000个玩家在地图上，也就是需要调用Update函数1000次。现在统一处理，只需要调用Update一次。处理这个计算的是新系统MoveSystem。

12.3.2 新系统MoveSystem

先来看看MoveSystem的定义，定义文件位于space工程的move_system.h文件中：

```
class MoveSystem : public ISystem<MoveSystem> {
public:
    MoveSystem();
    void Update(EntitySystem* pEntities) override;
private:
    timeutil::Time _lastTime;
    ComponentCollections* _pCollections{ nullptr };
};
```

在libserver基础库中有很多基础系统，现在需要在space进程上做一个特有的新系统，从上面的代码可以看到，新系统也是继承自ISystem，并实现在虚函数Update上，实现代码如下：

```
void MoveSystem::Update(EntitySystem* pEntities) {
    // 每0.5秒刷一次
    const auto curTime = Global::GetInstance()->TimeTick;
    const auto timeElapsed = curTime - _lastTime;
    if (timeElapsed < 500)
        return;

    if (_pCollections == nullptr) {
        _pCollections = pEntities->GetComponentCollections<MoveCom
        if (_pCollections == nullptr)
            return;
    }

    _lastTime = curTime;
    const auto plists = _pCollections->GetAll();
    for (auto iter = plists->begin(); iter != plists->end(); ++ite
        auto pMoveComponent = dynamic_cast<MoveComponent*>(iter->s
        auto pPlayer = pMoveComponent->GetParent<Player>();
        if (pMoveComponent->Update(timeElapsed, pPlayer->GetCompon
            pPlayer->RemoveComponent<MoveComponent>();
        }
    }
}
```

这个函数有3处值得我们注意：

(1) 刷新位置的间隔时间。在服务端输入“efficiency-thread”命令，可以看到整个框架每个线程的执行效率、每一帧花费的时间和一个最大时间。在非极端情况下，每一个线程在1秒钟之内至少可以执行大于500次update函数。如果按帧来计算位移，这个位移就太小了，所以我们将移动的计算时间间隔扩大到0.5秒。

(2) MoveComponent组件有两个Update函数，其中一个用于更新移动路径，另一个用于计算路径。

```
class MoveComponent :public Component<MoveComponent>, public IAwake
public:
    void Update(std::queue<Vector3> targets, Vector3 curPosition);
    bool Update(float timeElapsed, PlayerComponentLastMap* pLastMa
    ...
};
```

具体的算法这里不多说了，感兴趣的读者可以直接查看源代码，这个函数的使用就是随着时间的流逝计算现在玩家所在的位置。玩家下线之后，再次上线进入地图会定位到上次下线时保存的位置，该位置的数据保存在PlayerComponentLastMap组件上，所以这里传入了PlayerComponentLastMap组件的指针。

(3) 移除MoveComponent组件。当玩家走到目标点之后，有一个移除MoveComponent组件的操作。如果一直不停地行走，这个MoveComponent组件的信息就会不断更新，一旦停下来，这个组件就会被移除。这样做的目的在于减少整个MoveSystem的循环量。相对于循环量而言，整个框架创建对象和删除对象没有压力，用空间换取了时间。

12.3.3 加载新系统

新系统定义完成之后就需要运行它了。但是它不是libserver中的基类，所以需要有一个动态加载的过程。这个过程类似于跨线程创建组件。在space.h文件中，InitializeComponentSpace函数初始化了space进程需要的所有组件，同时调用CreateSystem函数创建移动系统，该函数为每个线程增加了一个指定的系统。下面是实现代码：

```
inline void InitializeComponentSpace(ThreadMgr* pThreadMgr) {
    pThreadMgr->CreateComponent<WorldGather>();
    pThreadMgr->CreateComponent<WorldOperatorComponent>();
    ...
    // 新系统
    pThreadMgr->CreateSystem<MoveSystem>();
}
```

创建系统的函数实现如下：

```
template <class T, typename ... TArgs>
void ThreadMgr::CreateSystem(TArgs... args) {
    std::lock_guard<std::mutex> guard(_packet_lock);
    const std::string className = typeid(T).name();
    if (!ObjectFactory<TArgs...>::GetInstance()->IsRegistered(className)
        RegisterObject<T, TArgs...>());
}

Proto::CreateSystem proto;
proto.set_system_name(className.c_str());
auto pCreatePacket = MessageSystemHelp::CreatePacket(Proto::Ms
pCreatePacket->AddComponent<CreateOptionComponent>(true, false
pCreatePacket->SerializeToBuffer(proto);
_cPackets.GetWriterCache()->emplace_back(pCreatePacket);
}
```

在上面的实现函数中，向每个进程发起了一个新协议MI_CreateSystem，这个协议的处理函数位于CreateComponentC组件中，CreateComponentC组件是每个线程中都存在的基础组件，用于创建组件，现在又多了一个创建系统的功能，参考代码如下：

```

void CreateComponentC::Awake() {
    auto pMsgSystem = GetSystemManager()->GetMessageSystem();
    ...
    pMsgSystem->RegisterFunction(this, Proto::MsgId::MI_CreateComp
    pMsgSystem->RegisterFunction(this, Proto::MsgId::MI_CreateSyst
}
void CreateComponentC::HandleCreateSystem(Packet* pPacket) {
    Proto::CreateSystem proto = pPacket->ParseToProto<Proto::Creat
    const std::string systemName = proto.system_name();
    const auto pThread = static_cast<Thread*>(GetSystemManager());
    if (int(pThread->GetThreadType()) != proto.thread_type())
        return;
    GetSystemManager()->AddSystem(systemName);
}
void SystemManager::AddSystem(const std::string& name) {
    const auto pObj = ComponentFactory<>::GetInstance()->Create(nu
    if (pObj == nullptr) {
        LOG_ERROR("failed to create system.");
        return;
    }
    System* pSystem = static_cast<System*>(pObj);
    if (pSystem == nullptr) {
        LOG_ERROR("failed to create system.");
        return;
    }
    _systems.emplace_back(pSystem);
}

```

通过以上代码可以看到，最终这个新创建的系统被放到了SystemManager系统管理类，而SystemManager是一个大容器，它并不关心自己管理的系统有什么功能，只要符合ISystem接口的对象都可以正常运行在这个大容器中。

12.3.4 测试移动

现在打开客户端，验证新编写的这个移动系统。测试之前确保地图上的寻路数据已经被“烘焙”好。打开美术工程，再打开地图，在Navigation界面重新烘焙一下地图，再重新打包。我们可以做3个测试：

第一个测试用来进行简单的登录测试。启动allinone，登录客户端查看行走情况。

登录游戏，进入地图之后，单击地图，角色就会触发行走。这时我们可以看到服务端在不停地打印计算出来的数据：

```
22:01:57 -> [DEBUG] - cur position. x:370.377 y:28.5287 z:500.98
22:01:57 -> [DEBUG] - cur position. x:370.033 y:28.5287 z:501.92
22:01:58 -> [DEBUG] - cur position. x:369.687 y:28.5287 z:502.86
22:01:58 -> [DEBUG] - cur position. x:369.342 y:28.5287 z:503.80
22:01:59 -> [DEBUG] - cur position. x:368.997 y:28.5287 z:504.74
22:01:59 -> [DEBUG] - cur position. x:368.773 y:28.7821 z:505.35
```

差不多0.5秒计算一次，最终角色停在了（368.773，28.7821，505.353）。在客户端增加了一个定时器，每秒打印一次坐标：

```
player position:(370.3, 28.6, 501.1)
player position:(369.6, 28.6, 503.0)
player position:(369.0, 28.8, 504.9)
player position:(368.8, 28.8, 505.4)
```

因为服务端与客户端采用了不同的计算方式，所以存在一些误差，误差在可控范围之内。

第二个测试用来测试同步数据的情况。测试同步数据主要是查看一个玩家在A端移动，B端是否有正常的的数据。要完成同步测试，我们需要有两个客户端。使用Unity生成一个PC端的EXE，同时运行两个客户端。

第三个测试修改engine.yaml文件，修改关于game配置的部分，启

动两个game进程。当有两个game进程时，登录两个玩家，他们会分布在两个game进程中，这时他们的表现会一致吗？读者可以自行测试第三种情况的结果。

总结

本章进入了真正的游戏世界，引入了一个客户端工程用于登录查看。本章扩展了System系统，在不改动底层框架的前提下动态地增加新的系统，还分析了在服务端可能出现的断线问题，断线是分布式框架中非常重要的一个异常处理，当游戏发布上线时，同时有数以千计的物理机，这些物理机每天都可能发生断线重连的情况。

至此，本书关于分布式框架实战的所有示例已全部讲解完成，已经创建了一个完整的、可扩展的、高性能的游戏基础框架。从登录到进入地图，我们足足讲了整本书，所有章节并没有刻意拖沓，甚至为了简洁，在某些章节仅讲解了重点。

一个基础的框架包括的内容实在是太多了。它是整个游戏的基础，一个既可靠又高效率的框架可以让后续的所有功能开发变得高效。这里的高效是双重的，包括高效的性能和高效的编码速度。

通过本书的框架，读者可以创建自己的游戏逻辑，例如背包功能、任务功能、公会功能等。背包和任务都是玩家个人操作，相对简单，只需要建立新的背包组件和任务组件即可。公会是一个比较大的功能，它有人与人之间的交互。如果我们需要在分布式服务器上实现这个功能，该怎么做？公会本质上可以视为一个World，这个World比起主城更简单一些。公会不需要有场景显示，只是一个数据的集合。沿着这个思路，读者可以自行设计下去。如果在阅读本书的过程中有不解或疑问，那么可以加入源代码库讨论群进行咨询，地址在源代码仓库中。

写在最后——如何构建自己的框架

在本书的最后主要聊聊框架。框架意味着编码思维，不过思维是没有定式的。在同一个问题上，你采取了这种方式，而其他人采取了另一种方式，在某些情况下，你的架构可能性能更优，而其他人的架构开发速度更快。

框架就好像是一个人的想法，它没有衡量标准，所以难有对错之分。纵览本书，我们提供的示例从最初的网络层演化到了多线程，又在多线程的基础上一步一步加入了ECS体系，实现了基本的游戏服务端需要提供的功能。

本书最终提供了一个基于ECS的服务器架构思路，也许读者在看这些示例的时候头脑里已经有了自己的想法。沿着这个思路，读者可以开发符合自己游戏需求的框架体系。

在本书的最后讲一个小故事。讲到编程思维，很多人都不禁要问，到底什么是编程思维？笔者的一位前策划同事在一次偶然的机会上，得到修改代码的权限，是为了方便策划而允许他修改一些固定数据，偏偏这位同事可能学过“程序设计”，或者至少学过程序设计语言的语法。问题来了，他修改了大量的代码。笔者发现仅一个函数就有整整1000行。是的，你没有看错，是1000行。笔者认真地看了这1000行代码，这是一个计算技能终结伤害量的函数，他是这样编写的，大量的代码都是如下模式的：

```
if (player.hasPassiveSkill(123) && !player.hasBuff(321)) {  
    player.addBuff(456)  
}
```

这些程序语句的大意就是，如果玩家有一个Id为123的被动技能，并且没有在321号BUFF之下，就给玩家加一个456号BUFF。这类程序语句占据了整整1000行。当然，if的条件不限于这一种，它是多种多样的。相关的数据点包括技能Id、角色Id、BUFF的Id和血量值，其类型也是丰富多样的。

后来，笔者和一位程序员聊起这件事，他分析其原因，是因为这位策划没有学过程序设计，只是学习了语法，他用一种线性的思维方式考虑程序的结构。这大概是初写程序的人的通病。线性的思维方式就是按轨迹寻求问题的解决方案，遇到问题解决问题。但问题往往是有规律性的，程序设计是分析这种规律找到较佳的解决方案。

搜索一下什么是程序设计。所谓程序设计，是给出解决特定问题程序的过程，包括分析、设计、编码、测试、调试（排错）等不同的阶段。

很多人学了程序设计语言的语法后就觉得自己可以编写程序。是的，但大概率可能是用100行就可以完成的工作却用了数倍之多，并且后期还难以维护。在上面的代码中，如果321和456号BUFF的效果发生了改变，这些代码可能就全部偏离了预期的目的。

我们都学过程序设计语言的语法，是什么原因造成一个人与另一个人编写出来的代码大相径庭？你可以说他的经验不够，再进一步讲，他累计编写的代码不够多。笔者个人觉得真正的原因在于他看的源代码不够多，这才造成了他认知的偏差，因为他没有学会其他的处理方式。如果一个人没有3年以上的编码经历，那么他可能难以形成一些常规的程序化思维。

那么，编程思维究竟是一种什么思维？在网上搜索一下，有很多解释。

本质上，这与接触过的代码量是相关的，你看过同一问题的处理方式越多，就会形成一种对于当前问题更优的解决方案。这里说的并不是服务器架构这类大的问题，可以小到“在游戏中杀死一头怪如何通知任务系统、通知周围玩家”这样的问题，你会进一步发现程序的世界中有很多类似这样的问题，它们被统称为“通知机制”的问题。通知机制有很多种处理方式，不同的人有不同的处理方式。

开始时，这种思维可以从设计模式中学到，但终究是纸上谈兵。随着我们的编码经验不断增加，再受到所看的代码的启示，或者研究过一些高质量的源代码，我们会学到一些解决问题的新方式。这些方式不断地在头脑中汇集起来，就会形成我们自己独有的编程思维。

随着看的代码越多，知道的处理方式越多，自己的编程能力也会越强，处理这些问题的方式越趋于专业的思维方式，因为根据自己的这些经验可以提供许多处理这类问题的方案，可以把它们汇总起来，找到一个想要的方式。

笔者曾经遇到过一个比较难以处理的问题。

策划需要实现一个让玩家对战并进行排名的功能，根据不同的排名每天领取不同的礼包。礼包需要在第二天任一时段领取，策划坚持要求由玩家手动领取。但是数据是按前一天21点的数据来结算的，玩家在第二天上线时再次进入战场，排名可能发生了变化。也就是说，不能以内存中当前排名的数据来领取礼包，必须是前一天21点的数据。

这个功能是几乎所有卡牌类手机游戏都有的功能，它们都用了一种方式进行处理，就是在21点发送邮件。服务端在21点整计算玩家对战的排名并生成奖励，奖励以邮件的方式发送给玩家，而且只能是前100、前1000领取。为什么不能让所有玩家都按照排名去领奖励，不能让玩家以点击按钮的方式去领取呢？这是因为服务端后台不能对所有玩家进行排序，过了21点之后这个数据转瞬间就没有了，只能在这个时间点处理。

从程序的角度来说，固定时间发送邮件并不是一个好的处理方案，但是似乎没有更好的处理方案。由玩家去领取当然更好，不过却做不到，对所有玩家实时排序更是不可能。

如果实在要做，有一种变通的方案，就是限定前1000人，服务器在21点时计算并生成数据，保存到历史表中，由玩家上线时自行领取。这样又引发了一个新的问题，如果一个玩家200天前得了第一名，但200天没有上线，是不是需要保存200天的全服数据呢？这显然不是一个好的方法，带来的数据量存储也很大，而且存储数据也要在21点瞬间完成，这对服务器也会造成压力。

难道真的没有其他好的方案了吗？我们一直没有找到很好的处理方

法。直到有一天，有一个玩家来找客服，他的一件极品装备不见了。最后问题转到笔者这里，还好有日志系统，跟踪他的装备时，发现这件极品装备被他用在另一件装备的强化上了。他的日志非常多，浏览数据时，有一个念头在脑海中一闪而过，这些数据都是历史数据，通过这些变化的数据可以得到他在10天或一个月前某一天的状态。笔者似乎找到了“排名问题”的处理方法，那就是记录这些数据的变化，而不是数据本身。

其实归纳起来，这就是一个“如何获取动态数据的历史数据”之类的问题，其实我们早就在使用了，诸如GM系统的日志追踪，只是没有把它和“排名问题”联系起来。如果不停地记录数据的变化，很快DB的容量就会变得庞大，我们做了一些优化，使得它的内容动态地增加或删除。

这种方案很快就体现出了优势，不需要排序，不会给服务端带来计算压力，游戏不会出现一瞬间的卡顿，也不会给邮件系统带来压力，可以在一天的任何时段随机领取。这种解决方案可以在任何指定的时间节点上计算出任何玩家处在哪个具体的排名上。

从本质上来说，我们在处理程序问题的时候还是在考量自己解决问题的能力。编码越多，遇到需要处理的问题就越多，看过的解决方案越多，对于某个问题处理方法的认知就越多，这大概就是大家说的编程思维。

本书完整地讲解了一种构建的思路，给出了一套完整的游戏前后端解决方案。还是本书开篇的那句话，学习的目的是找到答案，更是为了学到思维方式。希望本书能够为你引路，帮助你找到适合自己的高效的游戏架构思路，建立起自己的游戏编程思维。

共勉。

附录A 在CentOS上建立开发环境

A.1 安装CentOS 7

本书使用的Linux版本为CentOS 7。本节将为初次接触Linux的读者阐明如何在Windows下安装一个虚拟的CentOS系统，以方便读者更好地运行本书的Demo。我们需要使用如下两个基本软件包：

1.VirtualBox（Oracle虚拟化开源产品之一）

VirtualBox新版的下载地址为<https://www.virtualbox.org>，下载一个新的Windows版本即可。同时，在下载界面中需要下载扩展工具包VirtualBox 6.0.10 Oracle VM VirtualBox Extension Pack。VirtualBox安装完成之后，可以直接执行扩展包，它会自行安装。

2.CentOS系统

CentOS是一个Red Hat（红帽子）企业级Linux发行版本的社区发行版（同一源码编译，只是换掉了Logo与包管理源地址）。CentOS系统安装时推荐无图像界面，即命令行模式。有关CentOS本书就不再详细介绍了，如果习惯了Windows系统的界面，可能开始时不太习惯，但是长久工作之后，相信你会被这种无窗口、命令行的模式所吸引，因为它的确高效。

CentOS 7系统包可以在阿里的开源网站下载，阿里开源网站的链接为<https://opsx.alibaba.com/mirror>。在主界面的右边单击“下载OS镜像”，这里可以下载CentOS的很多版本。虽然已经有了更高的版本，但是我们还是选择更为稳定的CentOS 7，选择一个新的最小版本，Minimal即为最小版本。

A.1.1 创建虚拟机

安装完VirtualBox，在菜单上选择“控制”→“新建”，弹出创建界面，单击下方的“专家模式”进行设置。需要注意以下几点：

（1）名称可以随意写，方便记住即可。这里我们设置为CentOS Game Dev。

（2）我们选择虚拟机的类型为Linux，版本基于Red Hat 64位系统。

（3）内存大小设置，根据机器的性能划分，如果你的计算机有16GB的内存，可以划分4GB给虚拟机使用，如果你的计算机内存不够大，那么可以选择1GB或2GB。

（4）虚拟硬盘选择为“现在创建虚拟硬盘”。

单击“创建”按钮，弹出新界面，其配置说明如下：

（1）文件大小设置。这个参数相当于正常系统的硬盘大小，根据你的需求可以随意修改，这里我们输入80GB，意味着这个虚拟空间最多可以装载80GB的数据。对于开发用的Linux而言，80GB完全够用了。

（2）虚拟硬盘文件类型，这里默认为VDI，这是VirtualBox的默认格式。

（3）存储方式选择“动态分配”，这意味着之前我们定义的80GB空间不是一次性分配给虚拟系统的，它用多少就分配多少，占用空间的大小是自动增长的。而“固定大小”则是一开始就为我们创建80GB的硬盘空间。这两种分配方式各有利弊，我们选择“动态分配”，能够节约硬盘空间。“固定大小”速度略快于“动态分配”，可以根据你的需求进行选择。

单击“创建”按钮，创建界面消失之后，在VirtualBox主界面的左边会列出新建的虚拟机。在运行虚拟机之前，我们需要先对它进行一些设置。选中虚拟机并右击，从快捷菜单中选择“设置”命令，会弹出设置界面。

(1) 单击“系统”的第二个标签“处理器”，选择一个合适的处理器数量。如果你拥有8个CPU，可以选择2或4 CPU。

(2) 单击“网络”选项。我们为系统设置网络为桥连接。

单击下方的OK按钮，回到主界面上，单击主界面右侧的“启动”按钮。第一次启动时，会先让我们选择启动盘。单击右侧黄色的文件夹图标，选中之前下载的系统安装文件CentOS 7的ISO文件，单击“启动”按钮进行安装。

A.1.2 CentOS的安装参数

启动之后会弹出一个新的窗口，出现60秒等待安装界面，随后是欢迎界面。如果你是第一次使用VirtualBox，在安装系统的过程中，可能会发现鼠标变成了黑色，被禁锢在VirtualBox的小窗口之中，不能移出这个窗口。这时，可以按下键盘右侧的Ctrl键，鼠标会回到Windows系统中。

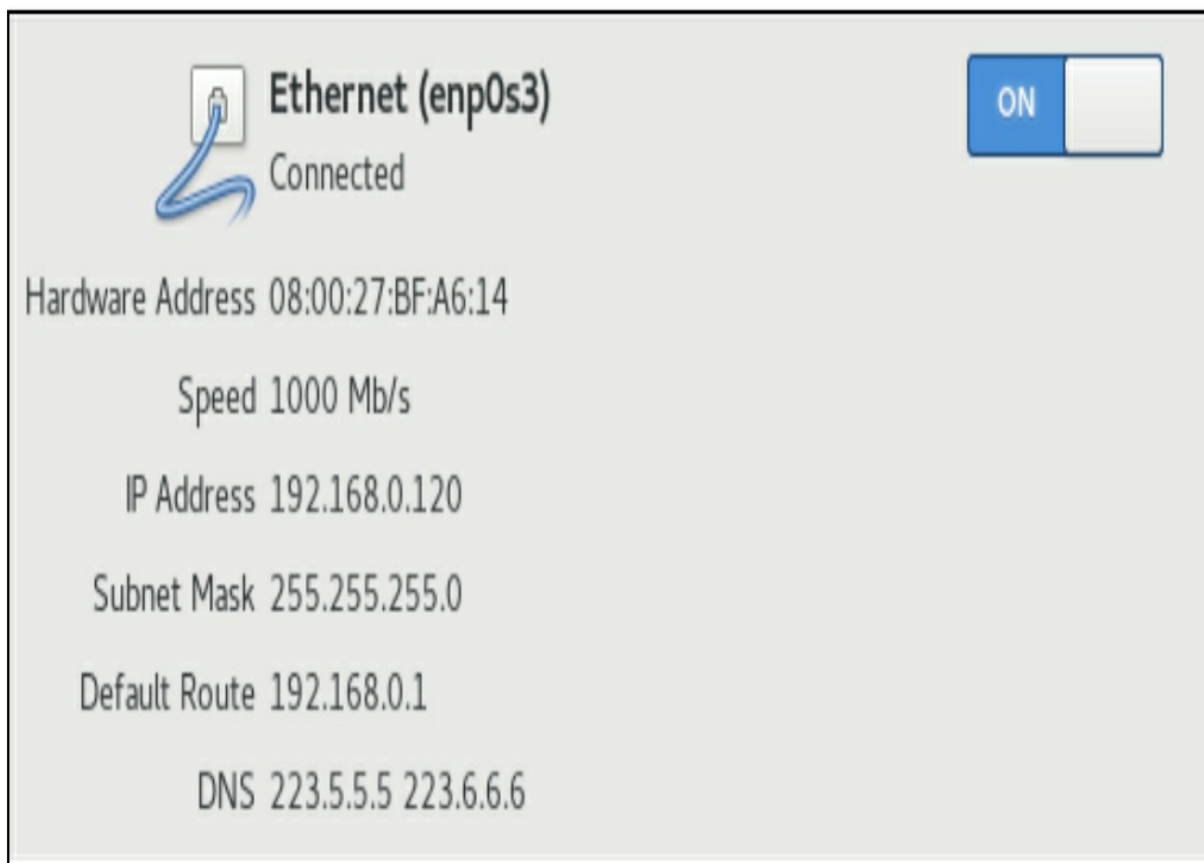
安装时默认的语言是英文，我们在安装中可以选择中文，在左侧下拉列表中滑动鼠标中键选择“中文”。单击“继续”按钮之后，进入安装信息配置。如果选择中文，一定记得在本地化语言支持中增加英语的支持，作为一个后台服务器，我们基本不会用到中文，只会使用英文。下面以英文为例逐项进行配置。

(1) 最顶端本地化选择一共有3项，分别是日期、键盘和语言支持。因为我们选择的是英文界面，所以只需要修改日期和时间，将其修改为Asia/Shanghai时区。

(2) 禁用KDUMP。在系统这一部分禁用了KDUMP，这是一个内核崩溃的反馈报告。因为我们的程序还不会撼动内核，为了加快性能，取消了没有用的服务。

(3) 单击Network&Host Name可以设置网络。首先需要打开右上角的Off/On开关，再单击右下角的Configure。单击IPV4 Settings标签，选择“手动”，单击Add按钮增加一个IP配置，例如输入192.168.0.120，这里Gateway根据当前的网络环境情况可能略有不同。同时，设置DNS为223.5.5.5和223.6.6.6，这是两个公用的DNS。单

击“保存”按钮之后，回到之前的界面，看到的配置如图A-1所示。



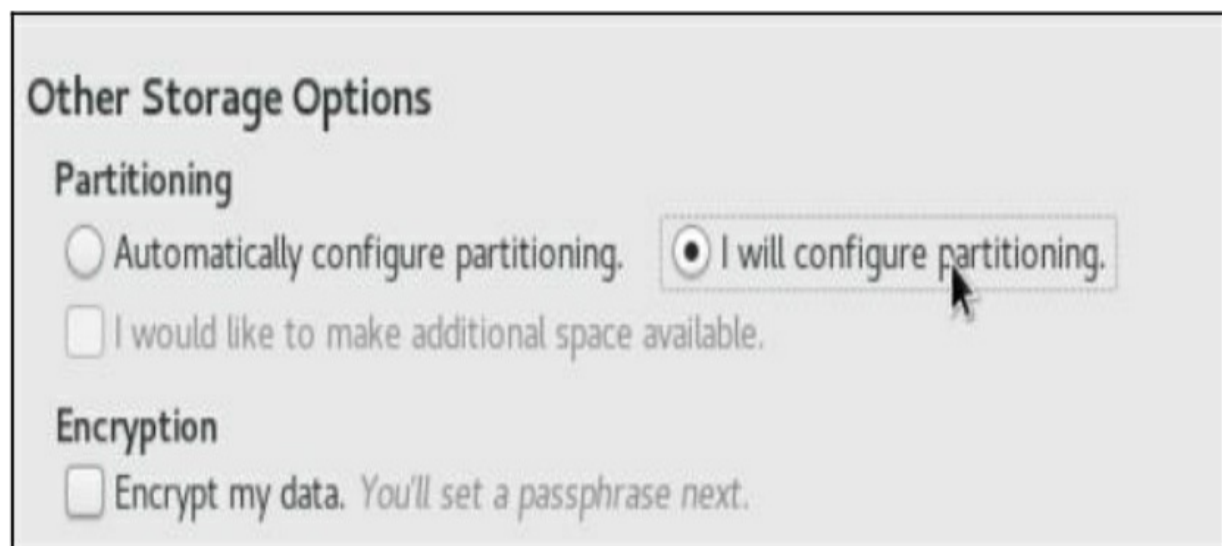
图A-1 网关配置

单击左上角的Done按钮，网络配置成功。这里需要解释一下，为什么必须为虚拟机设置一个IP，而不是默认IP。试想一下，当你想连接服务器时，服务器是一个自动分配的IP，应该去哪里找IP呢？

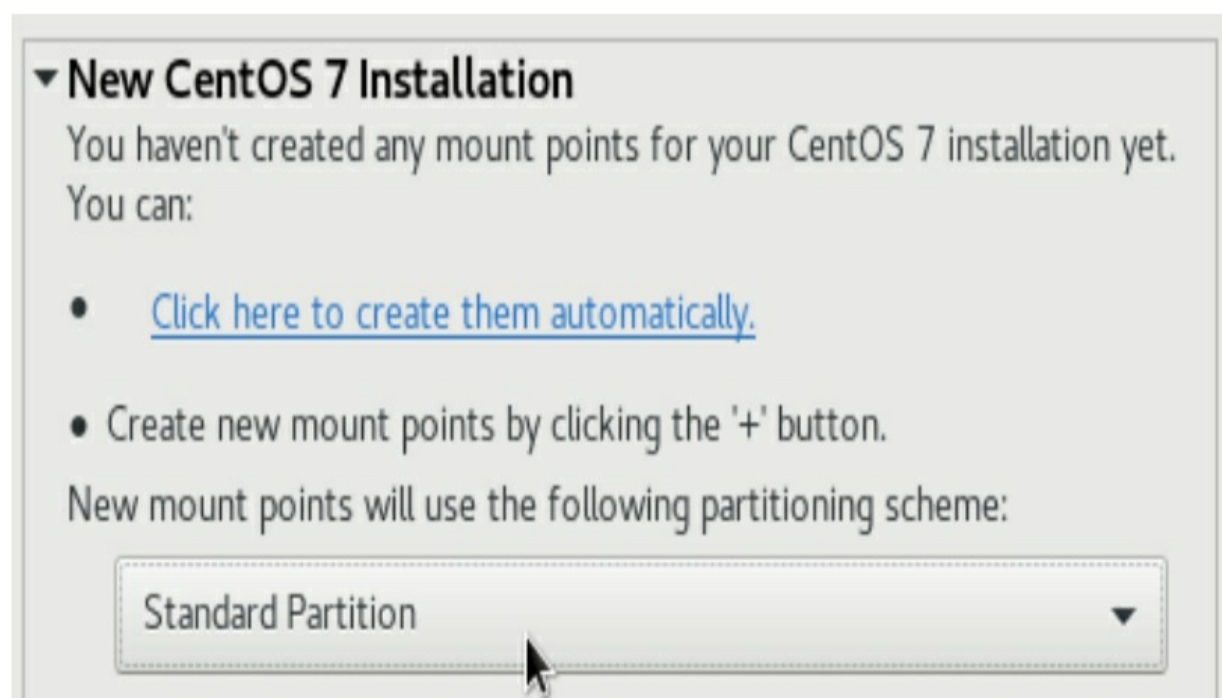
（4）分区配置。基本配置完成之后，重头戏是分区的问题，如果你从来没有接触过Linux系统，可能会有一点不适应。在Windows系统中，我们将一块硬盘分成C盘、D盘、E盘等，其中C盘是系统盘，可以装上系统或一些软件，而D盘或E盘可以存放数据。但Linux不是这样分区的，简单来说，Linux是按功能分区的。下面来做一个简单的分区，单击installation destination，进入分区界面，如图A-2所示。

选择I will configure partitioning（我要配置分区），然后单

击左上角的Done按钮，进入自主分区界面，如图A-3所示。选择Standard Partition（标准分区），单击左下角的“+”按钮进行分区。



图A-2 分区配置



图A-3 标准分区

简单分区规划如下：

① “/boot”：直接在界面上输入“1024MB”，该分区主要用于系统内核镜像文件与启动引导文件，Linux系统内核非常小，没有Windows系统那样庞大，这里我们选择了1024MB。

② “Swap”：8GB。交换区是Linux的虚拟内存，当内存不足时，数据被暂时放在这里，一般设为内存的2倍大小比较合适，交换区对于用户来说是不可操作、不可见的。

③ “/”：默认所有剩余空间。在挂载点选择“/”之后，期望容量不填时就会分配所有余下空间给它。

图A-4展示了一个总的分区图。分区完成之后，单击左上角的Done按钮完成，会弹出提示分区修改的窗口，单击“接受更改”后回到主界面。现在所有配置都完成了，单击Begin Installation开始安装。安装中可以进行Root密码设置。单击ROOT PASSWORD为root账号设置密码。等待安装完成，出现Reboot按钮，单击重启，即可完成安装工作。

SYSTEM		
/boot		1024 MiB
sda1		
/		71 GiB >
sda3		
swap		8192 MiB
sda2		

图A-4 所有分区

A.2 虚拟系统的关闭与备份

因为系统是虚拟的，所以关闭与启动都必须在VirtualBox中完成：

1.启动

在VirtualBox左侧列表中选中要启动的系统，单击右侧上方的“启动”图标即可启动，启动一个虚拟系统之后，会弹出一个新的窗口。CentOS启动相当快，提示输入登录名字，此时已经启动成功了。另外，还可以采用无界面启动，选中要启动的虚拟机并右击，从快捷菜单中选择“启动”→“无界面启动”。

2.关闭

要关闭虚拟机，直接关闭窗口即可，会弹出一个提示窗口，有3个选择：

- 快速休眠：就是保持当前的所有数据，下次启动时，内存数据都不会发生改变。假设你启动了一个服务，再启动时，这个服务还在。
- 正常关闭：就是正常的关机。
- 强制退出：相当于中断所有正在运行中的数据，拔掉电源强制关机。

这3种模式中一般使用“快速休眠”，这样启动和关闭虚拟机都很快。如果采用无界面启动，则关闭时右击，选择快捷菜单中的“退出”，也会出现如上的3种选择。

3.快照备份

VirtualBox是一个很重要的功能，特别是对于新手来说，有一个非常好用的功能——快照备份。快照备份的作用就是建立一个还原点，当我们需要安装一个不是特别确定的软件时，就可以先进行快照备份。当我们完成了一个基础的系统优化时，也可以创建一个快照备份，以便在之后随时切换回最初的状态。

在主界面中选中虚拟机，右侧有一个“当前状态”。单击上面的“生成”按钮，就可以根据当前状态生成一个系统快照备份，可以给它一个简单的名字进行备注。

4.恢复

选中某个备份数据右击，从弹出的快捷菜单中选择“恢复备份”。特别是对于新手来说，在安装一个工具或库之前，如果你拿不准，就可以备份当前状态；如果安装失败，或配置混乱到已经理不清头绪的时候，可以恢复之前的备份，重新再试一次。

A.3 终端工具XShell

每次在VirtualBox中启动CentOS，都会打开一个小小的窗口。一般情况下，我们不会直接在这个窗口上操作，都是通过第三方工具进行操作。第三方工具有很多，例如Putty、SecureCRT，本书中使用到的是XShell，虽然XShell是一个收费软件，但是它有一个免费版，完全够大家日常使用了。

连接Linux端的工具，使用的是XShell的免费版。XShell的下载与安装这里不再详述。安装完成之后，有两个设置需要说明：

(1) 在“工具”菜单中单击“选项”按钮，在“键盘与鼠标”标签中可以看到XShell对于鼠标中间按钮与向右按钮的定义，中间按钮是复制当前选中的内容，而向右按钮是弹出菜单，如图A-5所示。如果你的习惯不相符，那么可以在这里修改。



图A-5 XShell鼠标设置

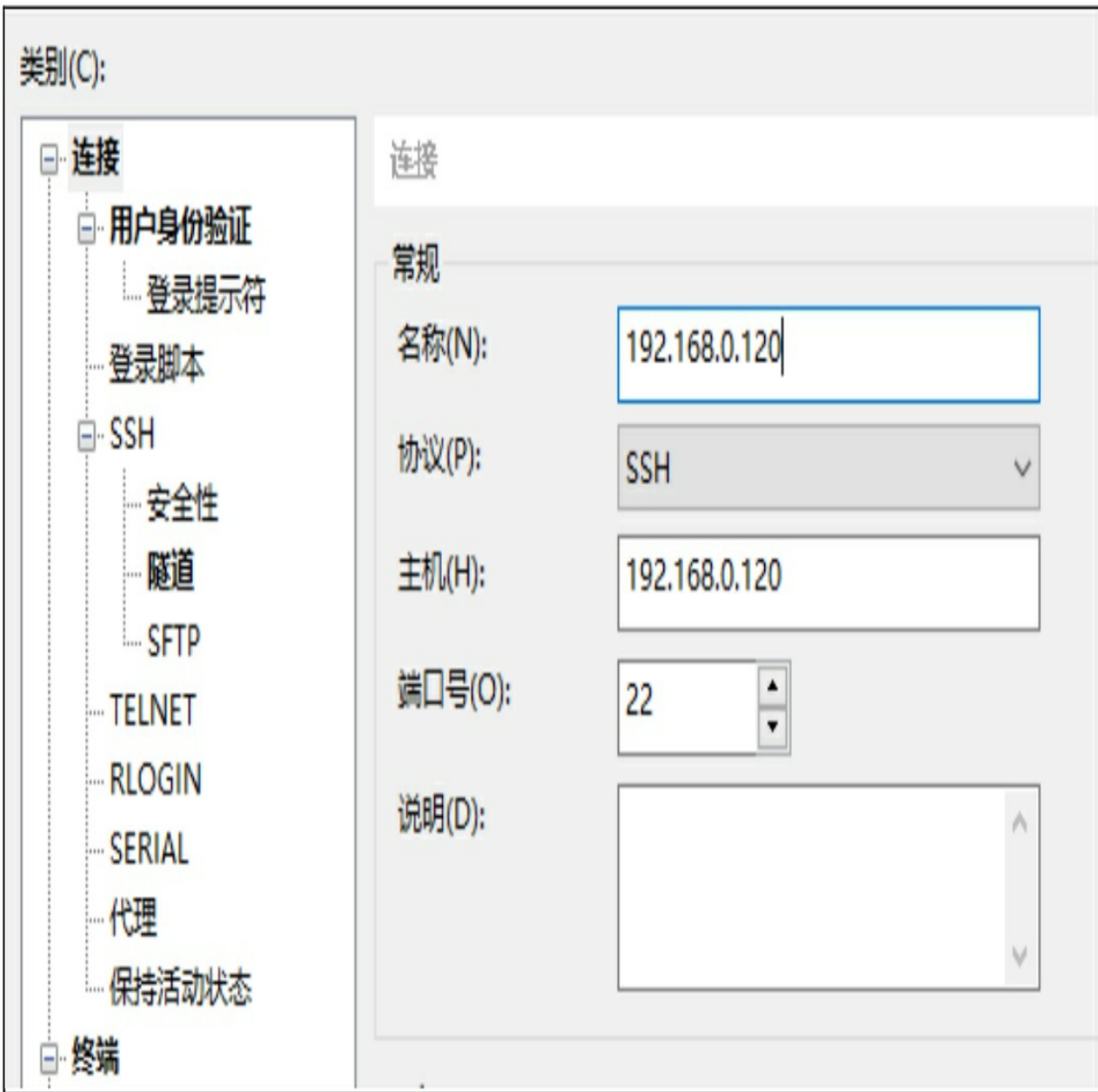
(2) 在“文件”菜单中单击“默认会话属性”，弹出属性设置界面，在“终端”→“键盘”选项中设置，如图A-6所示。



图A-6 XShell键盘设置

当我们用XShell登录进入Linux后，在普通模式下对输入后的字符串进行退格删除操作没有问题。而在运行中，按Delete或Backspace键时会产生^H等乱码问题。这个键盘设置会解决乱码的问题。

配置完成之后，再来看看XShell如何连接CentOS系统。在VirtualBox中启动完成CentOS系统之后，打开XShell。单击“文件”→“新建”菜单，或者单击工具栏的第一个带有“+”的图标，新建一个连接配置，如图A-7所示。



图A-7 XShell连接设置

在“连接”对话框中，填入之前配置的CentOS系统设置的IP，然后单击“确定”关闭“新建会话属性”对话框。第一次连接服务器时会有一个安全提示，选择“接受并保存”即可，随后输入账号root以及之前设置的密码。

```
Connecting to 192.168.0.120:22...
```

```
Connection established.  
To escape to local shell, press 'Ctrl+Alt+]'.  
[root@localhost ~]#
```

出现#提示符时，表示正常登录。至此，我们已经成功登录到了CentOS系统上。输入cat/etc/redhat-release，我们可以查看到当前系统的内核版本为：

```
[root@localhost ~]# cat /etc/redhat-release  
CentOS Linux release 7.7.1908 (Core)
```

有些初次接触Linux系统的读者可能会有疑问，为什么不直接在VirtualBox弹出的窗口中登录，还要绕一圈到XShell中进行登录操作呢？当然，是可以直接登录的，但是原始窗口的操作非常不方便，而XShell这个终端工具比较实用，本书我们使用的是虚拟机，服务器就在本地，而在实际情况下，我们使用的Linux系统可能在互联网上，所以最好是习惯使用终端操作系统。

A.3.1 CentOS命令说明

使用XShell进入系统之后，只有一个cmd提示符。除此之外，什么也没有了。你可能有些不习惯，因为它没有窗口界面。并不是所有系统都需要有界面，CentOS可以安装界面插件，但本书不讨论这个步骤，感兴趣的读者可以从网上搜索此方面的内容。作为服务端使用时，并不需要界面。如果在Windows系统中有过cmd（命令提示符）界面下工作的经历，那么对于命令行应该一点也不陌生，常用命令见表A-1。

表A-1 Windows系统常用命令表

命 令	说 明
dir	查看当前目录下的文件结构
cd xx	进入 xx 目录
cd ..	回到上层目录

这些命令在Linux系统上大同小异，因为没有界面可供操作，所以需要通过大量的命令来达到目的。表A-2列出了一些常用的命令，以方便读者查询。

表A-2 Linux系统常用命令表

命 令	说 明
ls	查看当前目录下的文件结构
cd xx	进入 xx 目录
cd ..	回到上层目录
rm -rf xx	删除 xx 目录
mkdir xx	在当前目录下新建一个名为 xx 的目录

这些命令全是小写的。值得注意的是，在Linux系统上是区分大小写的，也就是说如果我们新建一个test文件，还可以再建一个Test文件，这种做法在Windows系统下是不可行的，但是Linux下完全可以。如果你有查看开源社区源代码的习惯，会发现在服务端的代码，文件名绝大部分都是小写的，这绝对不是巧合。因为如果用大写的文件名，输入或查找时还要按一下Shift键，而你一旦忘记大写的，系统还会告诉你找不到该文件。因此，在服务器编码时，我们绝大部分情况下都是默认使用小写来命名的。更多的命令可以通过help命令查看。

```
[root@localhost ~]# help
```

当我们输入“help”命令时会列出常规的命令以及说明，这些shell命令是内部定义的。在Linux系统下，大部分命令都可以通过help来查看它的使用。例如mkdir命令：

```
[root@localhost ~]# mkdir --help
[root@localhost network-scripts]# mkdir --help
Usage: mkdir [OPTION]... DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.
Mandatory arguments to long options are mandatory for short option
  -m, --mode=MODE      set file mode (as in chmod), not a=rwx - umask
  -p, --parents         no error if existing, make parent directories
.....
```

在说明中指出了mkdir是用来创建文件夹的，它给出了mkdir的格式，还给出了选项参数的说明。初接触系统，我们不需要了解太多命令，用到的时候可以在网页上搜索一下用法。随着使用的频度和时长增加，大部分命令都会慢慢掌握。

A.4 初始化系统

在开始使用之前，我们需要为这个新的系统做一些初始化与优化处理，同时为它安装一些必要的服务。CentOS安装软件比起Windows系统要方便得多，其中有一个安装服务名为yum，其全称为Yellow Dog Update Modified，作用是对软件包进行管理，能够从指定的服务器下自动下载包并安装，如果安装的包有依赖包，那么还会自动安装依赖包。如果你第一次接触到Linux，不太明白也没有关系，本书所有第三方包安装时都给出了相应的步骤。

CentOS自带的仓库对于国内的用户来说可能存在网速上的问题，我们选择阿里仓库镜像。本书源代码库中的Linux目录下有一个名为init_c7.sh的脚本，将它上传到自己的服务器。先安装一个命令：

```
[root@localhost ~]# yum -y install lrzsz
```

将init_c7.sh拖到XShell窗口上，就可以自动执行rz上传命令，上传完成之后，查看一下目录：

```
[root@localhost ~]# ls
anaconda-ks.cfg init_c7.sh
```

输入“ls”查看目录情况，这时就会看到有一个新的文件init_c7.sh。这时，这个sh还是不可执行状态，我们修改它的执行权限：

```
[root@localhost ~]# chmod a+x init_c7.sh
```

有关chmod命令的详细情况，感兴趣的话可以上网查阅一些相关资料。执行init_c7这个文件，输入如下命令后按回车键：

```
[root@localhost ~]# ./init_c7.sh
```

因为在脚本中有一个更新当前系统的操作，所以需要一定的完成时间。在这期间，我们看一下这个脚本的内容：

```
#!/bin/bash
#
#
# 下载阿里云的yum源，替换 /etc/yum.repos.d/CentOS-7.repo 文件
curl -so /etc/yum.repos.d/CentOS-7.repo HTTP://mirrors.aliyun.com/
# 安装需要用到的基础包
yum -y install bash-completion vim lrzsz chrony git yum-utils epel
# 下载阿里云的epel源
curl -o /etc/yum.repos.d/epel.repo HTTP://mirrors.aliyun.com/repo/
# 更新整个系统
yum -y update
# 关闭不用的 seLinux
sed -ri 's/((SELinux=).*\/\1disabled/g' /etc/seLinux/config
# 设置时区为亚洲/上海
\rm /etc/localtime
ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
# 修改系统限制用户的打开文件数与创建进程数
```

```
cat > /etc/security/limits.d/20-nproc.conf <<EOF
# Default limit for number of user's processes to prevent
# accidental fork bombs.
# See rhbz #432903 for reasoning.
*          soft      nproc          unlimited
root       soft      nproc          unlimited
EOF
cat >> /etc/security/limits.conf <<EOF
*          -      nproc      65535
*          -      nofile     655350
EOF
# 关闭所有默认系统服务
systemctl list-unit-files --state enabled|awk '{print $1}'|xargs -
# 开启用到的服务(计划任务、系统日志、ssh远程登录、时间同步)
systemctl enable crond rsyslog sshd chronyd network
```

这个脚本的前三行是关于yum的，当我们需要某个库时，可以直接在这些仓库（源）中查找，然后安装。因为yum默认的仓库地址位于境外，这里把它替换成了阿里云的地址，阿里有一个仓库的镜像，下载起来速度更快。可以访问<https://opsx.alibaba.com/mirror>查看这个源中的内容。

“yum-y update”这一行，是对当前系统中已经安装过的库进行更新。第一次执行这个脚本用的时间会比较久，因为要更新系统与库。随后在这个脚本中，我们更换了一下时区，安装了一个chrony的库，以便同步更新时间，并关闭了一些不用的服务。脚本执行完成之后，可以查看初始化了哪些服务：

```
[root@localhost ~]# systemctl list-unit-files --type service --sta
```

查看当前运行的服务，我们可以用命令systemctl status生成一棵进程树。

如果你初次接触CentOS，可能会有些焦虑，如此多的命令怎么可能记得住。没必要一次性记住，需要使用时可以在网络上搜索一下。更多的Linux命令可以一边使用一边学习，本书的重点在编码上，所以不需要过多担心。我们完成了系统的初始化之后，一定要重启系统，让所有的效果生效。

特别说明，如果在升级系统时遇到有Network网络部分的更新，那么初始化时的IP有可能会失效。这时，需要手动重新修改IP地址。网络配置在“/etc/sysconfig/network-scripts/”目录下：

```
[root@localhost ~]# cd /etc/sysconfig/network-scripts/  
[root@localhost network-scripts]# ls  
ifcfg-enp0s3 .....
```

配置文件在第一个文件ifcfg-enp0s3中，这个enp0s3名字可能会不一样，有时也可能是eth0等其他名字，但文件内容是一致的。使用vim就可以修改IP了，如果你不会用vim，翻到A.5.2节有一个简单的说明。

A.5 CentOS常用库说明

本节将介绍几个在开发过程中常用的基础库，分别是yum、Vim和samba。

A.5.1 yum

在Linux系统上yum无疑是一个重要的命令。前面提到过yum可以为我们安装编码所需要的库，也可以安装一些工具软件，例如gcc、vim、tomcat等。

常用yum命令见表A-3。如果记不住yum的命令，也没有关系。在Linux下工作，我们要学习的第一件事情就是使用help，可以使用yum help。help会列出我们记不住的所有命令，help命令几乎可以覆盖所有Linux下的命令，即使是开源的库也会实现“-help”“-h”或“--help”命令，这几乎是一条不成文的规则。我们可以通过help来看看yum的命令列表。

表A-3 yum常用命令表

命 令	说 明
yum install <keywords>	在系统中安装一个或多个软件包
yum update	更新当前系统所有的包

(续表)

命 令	说 明
yum list installed <keywords>	列出一个或一组软件包。例如：yum list installed grep java 列出了 Java 的安装包。grep 是一个关键字，它是一个文本搜索工具。installed 表示本地已安装。如果我们开发时要用到 Java，这时可以使用该命令，查看本地是否已经安装了 Java
yum search <keywords>	在软件包详细信息中搜索指定字符串
yum info <keywords>	在软件包中查看某个具体的包的信息

```
[root@localhost ~]# yum help
List of Commands:
check          检查 RPM 数据库问题
check-update   检查是否有可用的软件包更新
clean          删除缓存数据
.....
```

因为太长了，这里就不逐一列举了。举一个简单例子来说明如何使用yum安装执行文件或库，假如我们要安装一个名为graphviz的工具，可以采用以下几步：

(1) 用search命令查看是否有该工具：

```
[root@localhost ~]# yum search graphviz
```

```
.....  
graphviz-devel.i686 : Development package for graphviz  
.....  
graphviz-python.x86_64 : Python extension for graphviz  
.....  
graphviz.i686 : Graph Visualization Tools  
.....
```

search命令列出了所有关于graphviz的库或执行文件，graphviz-devel是一个开发包，graphviz.i686是可执行文件。当我们只需要工具，不需要开发包时，可以直接下载可执行文件。

(2) 使用install命令安装：

```
[root@localhost ~]# yum install graphviz
```

执行install命令后会提示是否安装，输入“y”，直至安装完成。

(3) 使用find命令查看安装路径：

```
[root@localhost ~]# find /usr/ -name graphviz  
/usr/lib64/graphviz  
/usr/share/graphviz
```

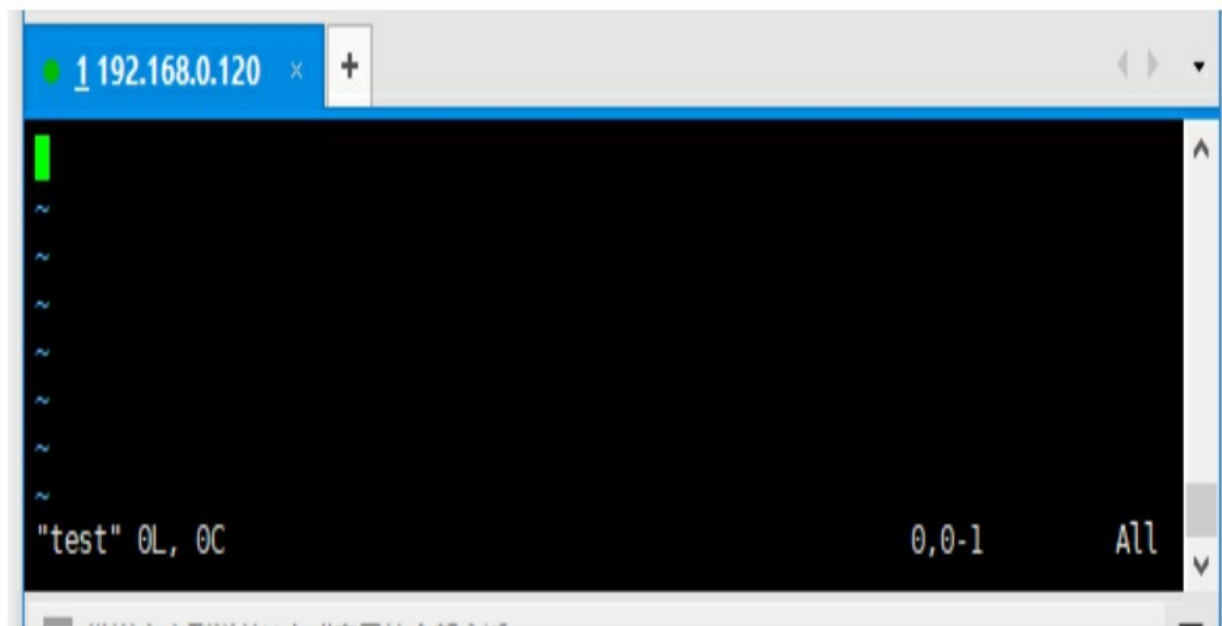
当我们安装完成，想要看看它到底安装在哪里，有哪些安装文件时，可以使用find命令，格式为“find path选项参数”。在上例中，我们使用了-name选项，即为名字查找。更多的选项可以用“find-help”查看。至此，我们已经了解了在CentOS下yum如何安装需要的库了。需要说明的是，并不是所有的Linux系统都有yum，在Ubuntu系统下是采用的是apt-get命令。

A.5.2 Vim

除了yum之外，另一个需要有一定了解的是Vim。Vim是一个著名的编辑器。在初始化系统时已经安装好了Vim。我们以一个例子来简单介绍Vim的使用。

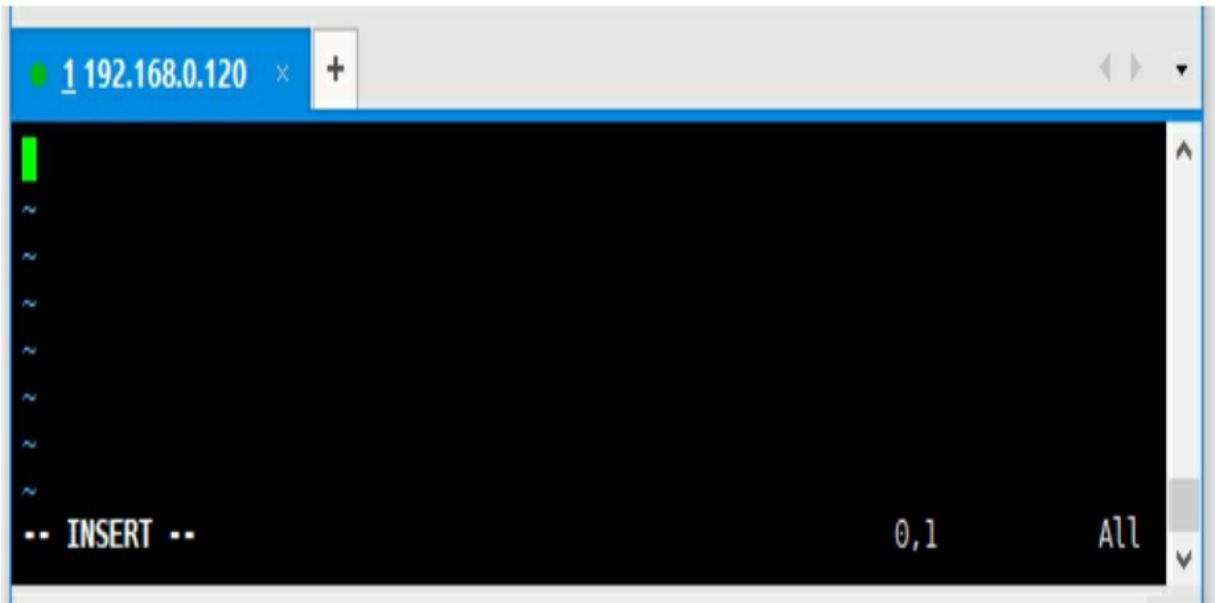
```
[root@localhost ~]# vim test
```

在命令行输入“vim test”，看到的界面如图A-8所示，表示在当前目录打开一个名为test的文件，如果没有该文件会在内存中建立一个缓存文件。



图A-8 Vim空白文档

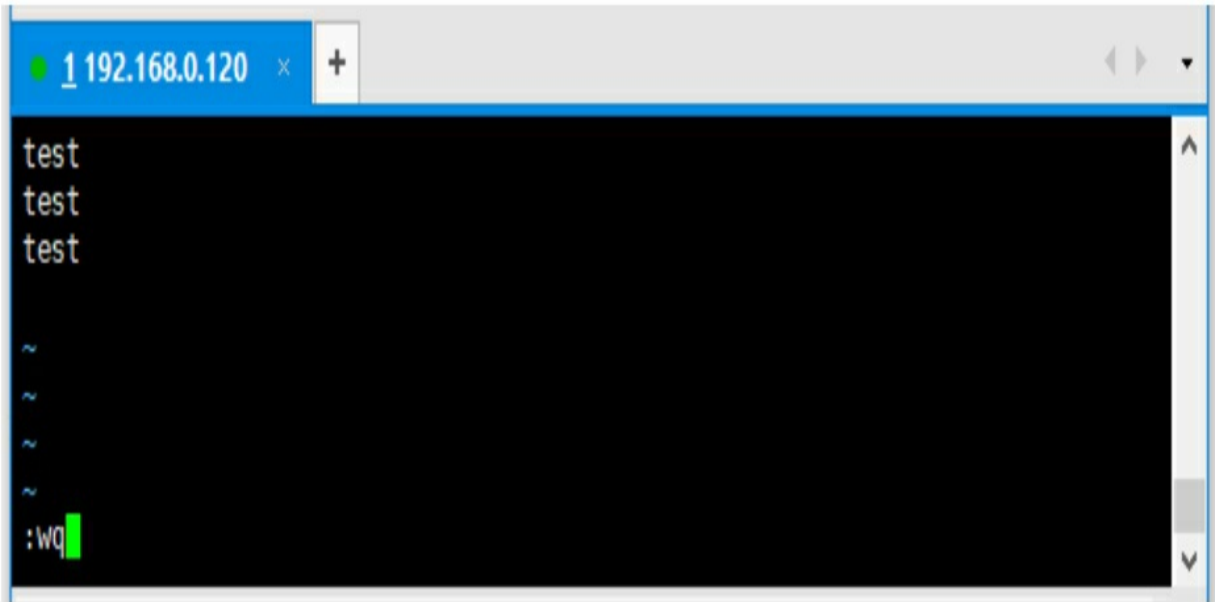
这是一个空文件。在当前这个状态下，我们称为命令行模式，在该模式下是没有办法输入文字的，所以命令行就是等待你下达命令。按下i键，这时界面会发生变化，如图A-9所示。



图A-9 Vim插入状态

注意最下方状态栏的显示信息，这时代表可以在文件中插入数据。在插入数据模式下，上下键可以移动光标，PageUp和PageDown两个键可以翻页。输入完成之后，按下Esc键退出INSERT模式，回到命令行模式。

输入“: wq”，如图A-10所示，在命令行的最下方就会出现输入的命令。wq=quit and write即退出并写入，直接按回车键回到目录界面。使用“ls”命令查看，我们会发现多了一个test文件，它没有后缀，因为我们没有给它加后缀，不过没有关系。再次用“vim test”打开，会看到与退出时一样的内容。



图A-10 Vim保存退出

值得一提的是，在Linux系统下不是靠文件后缀来确定文件类型的，这一点与Windows系统非常不同。在Windows系统下，后缀.exe表示是一个可执行文件，但在Linux系统下，即使没有后缀，它依然是一个可执行文件，但在Windows下如果修改了.exe后缀，它就不能再执行了。表A-4列出了几个常用的命令。

表A-4 Vim常用命令表

命 令	说 明
Esc 键	退出插入模式，切换到命令模式
i 键	进入插入模式
:q	如果无修改，就直接退出
:w	保存文件
:wq	保存，同时退出
:q!	修改过文件，又不想储存，使用!符号强制退出

Vim除了命令之外，还有很多快捷操作方式，其用法多样，需要一段时间的学习。本节简单地阐述了Vim，因为后面需要对Linux的一些文件做出修改，希望这种修改步骤不会成为你执行本书例子时的障碍，所以这里简短说明了Vim的使用。如果你曾经用过Vim，当然更好，如果没有用过Vim，可以适当查看一些资料加以了解，慢慢你就会入门，并且喜爱这种命令行模式的操作，这是一种完全可以脱离鼠标的操作方式，可以提高操作效率。

如果你还不是很了解Vim，也不需要过分担心，本书所有使用Vim进行编辑的地方都会有步骤提示。在Linux上，除了Vim编辑器之外，用于编码的编辑器可以使用Emacs，Emacs上手比较慢，因为它基本靠快捷键操作，熟练使用会加快编码速度。

A.5.3 Samba

为了方便查看CentOS系统上的文件，我们可以安装一个名为Samba的工具软件。Samba的作用是共享目录，它只能使用在局域网中，方便Windows系统与Linux系统交互。直观的功能就是可以在Windows系统上查看文件目录，实现文件共享。我们还是使用yum进行安装，其命令如下：

```
[root@localhost ~]# yum -y install samba
```

安装完成之后，我们需要对Samba进行配置。在配置之前，需要有一个Samba账号，可以使用pdbedit命令来创建Samba账号，在创建账号时，需要注意Samba账号必须先有系统用户的账号。下面将系统的账号root映射到Samba中：

```
[root@localhost ~]# pdbedit -a root
new password:
retype new password:
Unix username:      root
.....
```

提示输入密码之后，账号就创建成功了。现在，启动Samba，使用如下命令：

```
[root@localhost ~]# systemctl start smb
```

在Windows系统的窗口中，在文件资源管理器的地址栏中输入“\\192.168.0.120”，访问192.168.0.120就可以看到root目录。进入root目录就可以看到之前上传的init_c7.sh文件。

执行如下命令，开机自动启动Samba：

```
[root@localhost ~]# systemctl enable smb
Created symlink from /etc/systemd/system/multi-user.target.wants/s
```

现在可以方便地在Windows系统下操作CentOS系统的文件了。

更多的Samba的配置可以在网上查询一下，其配置文件位于“/etc/samba/smb.conf”。

A.5.4 Git

Git是一个开源的分布式版本控制工具，本小节详细说明在Linux上

获取源代码的步骤。如果你已经有相关经验，可跳过此处所讲的内容。

(1) 安装Git，如果已安装，就跳过安装。

```
[root@localhost ~]# yum install -y git
```

(2) 获取源代码库：

```
[root@localhost ~]# git clone git地址 ./game
```

Git的clone命令是将一个指定的Git库获取到本地，我们指定获取目录为“./game”，即当前目录的Game子目录下。

(3) 编译。在CentOS下，每个工程中都有make-all.sh文件，用于批量执行代码编译命令。如在执行过程中遇到“./make-all.sh: 行2: \$'\r': 未找到命令”这样的错误，是因为两个系统下按回车键不同造成的，请使用“dos2unix make-all.sh”修正问题。

(4) 提交代码步骤。Git代码提交步骤和SVN有些不一样，在SVN中，如果执行commit，这些commit会立即上传到源代码服务器上。Git在本地有一个源代码仓库，可以理解为它先在本地克隆了一个一模一样的仓库。执行commit时，修改代码被提交到了本地仓库。如果想要更新源代码仓库，那么还需要使用push命令。

首次进行commit操作，需要先进行全局配置：

```
[root@localhost game]# git config --global user.email "xxx@xxx.com"
[root@localhost game]# git config --global user.name "xxx"
```

该配置将在提交代码时用于显示提交者的信息，只需配置一次。

首次push操作也需要进行全局配置：

```
[root@localhost game]# git config --global push.default simple
```

```
[root@localhost game]# git config --global credential.helper store
[root@localhost game]# git push origin master
```

在执行以上命令时可能会需要输入账号名与密码。“git push origin master”这条命令是将远端的master主分支与本地的origin分支关联起来，方便后面“git push”的使用。如果只是浏览仓库代码，那么可以跳过这一步骤。“credential.helper”会将我们输入的Git凭证保存下来，不用每次push到远端的时候都输入。在执行commit提交之前，可以先查看当前提交的状态：

```
[root@localhost game]# git commit --status
# 位于分支 master
# 尚未暂存以备提交的变更
#   （使用 "git add <file>..." 更新要提交的内容）
#   （使用 "git checkout -- <file>..." 丢弃工作区的改动）
```

该命令会列出当前被修改的所有文件，并做出操作提示，如果有需要提交的文件，使用“git add”命令可以将它加入待提交列表中。“git checkout”命令重新从库中取出文件覆盖当前文件，表示修改被丢弃，最后提交。

```
[root@localhost game]# git commit -m "modify"
```

参数“-m”表示message，提交说明。commit完成之后，你修改的代码已经进入本地origin分支中了。这时，它还没有进入远端的master分支，使用push命令将数据推送到远端：

```
[root@localhost game]# git push
```

并不是每一次commit之后都需要push操作，如果你做一个较大的版本改动需要花费三五天时间，可以每天commit到本地仓库中，最终完成时再进行push操作。

(5) 更多命令。这里在表A-5中列举Git的常用命令，以便随时查

阅。

表A-5 Git常用命令

命 令	说 明
git pull	从远程仓将新的数据拉到本地仓库，类似于 SVN 的升级操作
git clone 地址 目录	命令 clone 就和它的意思一样，克隆一个源代码库到本地目录下
git clean -f	删除当前目录下所有没有进入库管理的文件。如果有些本地文件不在库中，但又不想删除它，可以修改.gitignore 文件，将该文件加入
git clean -n	使用 clean 之后，可能会误删除一些有用的文件，可以使用 clean -n 预览一下将要删除的文件
git status	查看当前文件的变更操作
git checkout	检出一个分支或一个路径到当前工作仓库中。如果我们在使用中不小心误除了文件、目录或分支，可以使用该命令重新获取
git diff	在提交 commit 之前还可以使用 diff 命令查看一下到底修改了哪些内容
git add	将修改过的文件添加到待 commit 列表中。git add -A 可批量添加

Git的命令相当多，一时记不住也没有关系，在本书的例子中使用到它时会逐一进行提示。更多的命令说明可以使用help命令查看。

```
[root@localhost game]# git --help
```

如果我们需要看一个子命令的格式，以commit为例，可以使用以下命令：

```
[root@localhost game]# git commit -help
```

A.6 CentOS编译工具

要在CentOS上搭建一个编译环境，除了之前的一些基础库之外，还需要安装专业的库文件。

A.6.1 GCC

开始进行Linux编码之前，我们需要安装一个编译工具GCC。GCC是目前被大多数Linux操作系统认可的标准编译器，这个工具就相当于Windows端的Visual Studio工具，不同之处在于GCC没有界面，它是一种靠命令完成编译的工具。Visual Studio工具本质上最终也是靠一系列的命令完成编译，只是它将命令部分隐藏，而配置部分是由界面来实现的。

关键点1：下载GCC

查看yum库中自带的GCC版本，还不到5.0：

```
[root@localhost ~]# yum info gcc
```

可安装的软件包：

```
名称: gcc
架构: x86_64
版本: 4.8.5
.....
```

本书中使用的是C++14标准，当前yum仓库中的GCC不适合我们使用，所以手动安装GCC。要下载GCC版本，先安装一个下载工具wget。wget是Linux上的一个下载工具，支持断点续传。我们需要用它来下载GCC的安装包。

```
[root@localhost ~]# yum install -y wget
```

wget安装完成之后，我们需要从GCC官方下载安装包。打开[HTTP: //ftp. gnu. org/gnu/gcc/](http://ftp.gnu.org/gnu/gcc/)这个网页，可以看到当前官方的所有版本，选择一个比较新的版本gcc-9.1.0，使用wget开始下载。

```
[root@localhost ~]# wget -c HTTP://ftp.gnu.org/gnu/gcc/gcc-9.1.0/g
```

若输入命令时出现：“正在解析主机ftp. gnu. org (ftp. gnu. org) ... 失败：未知的名称或服务”类似的信息，则可能是之前配置的DNS无法连接，使用vim工具修改/etc/resolv. conf文件，将223. 5. 5. 5和223. 6. 6. 6替换为8. 8. 8. 8和8. 8. 4. 4这组DNS服务器。下载完成之后，使用“ls”命令查看，多了一个gcc-9.1.0. tar. gz文件。

```
[root@localhost ~]# ls
anaconda-ks.cfg  gcc-9.1.0.tar.gz  init_c7.sh
```

gcc-9.1.0. tar. gz是一个压缩文件，在Windows系统下，我们习惯使用“.rar”或“.zip”后缀的压缩文件，tar是Linux系统上常用的压缩工具。

关键点2：依赖库

下载好GCC之后，还需要安装两个依赖工具。

(1) 低版本GCC

下载完成之后，从yum中安装一个低版本GCC来进行编译。因为我们下载的GCC9的版本也是源代码，源代码是需要编译的，所以下载低版本的GCC来编译它。

```
[root@localhost build]# yum install -y gcc gcc-C++
```

```
[root@localhost build]# gcc -v
.....
GCC版本 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC)
```

GCC提供了标准C的编译，GCC-C++提供了C++标准的编译。安装完成之后，我们看看GCC的版本。

(2) bzip解压

在编译GCC之前，我们还需要安装bzip工具。

```
[root@localhost ~]# yum -y install bzip2
```

关键点3: 配置编译

一切都准备好之后，开始对下载的版本进行编译。分成以下几个步骤：

```
[root@localhost ~]# tar xf gcc-9.1.0.tar.gz
[root@localhost ~]# ls
anaconda-ks.cfg  gcc-9.1.0  gcc-9.1.0.tar.gz  init_c7.sh
```

解压之后，当前目录多了一个gcc-9.1.0子目录。进入子目录后，下载gcc需要的依赖包：

```
[root@localhost ~]# cd gcc-9.1.0/
[root@localhost gcc-9.1.0]# ./contrib/download_prerequisites
```

下载依赖包的时间是由本地的网络决定的，GCC的下载和编译的时间相当长，需要耐心等待。执行完成之后会显示All prerequisites downloaded successfully。接下来，我们在GCC目录下新建一个名为build的子目录，进入该目录中，执行configure配置。

```
[root@localhost build]# mkdir build; cd build
[root@localhost build]# ../configure --prefix=/usr/local/gcc-9.1.0
```

我们将安装目录定为“--prefix=/usr/local/gcc-9.1.0”。 “--disable-multilib”是指定我们仅编译当前的64位版本，不产生多版本，这样可以节约编译时间。

这一步执行完成之后，Makefile文件已经被创建出来了。在Linux系统下，我们要对C++文件进行编译，重要的配置文件就是Makefile。后续会进一步介绍如何生成Makefile，现在只需要编译执行它就可以了。输入以下命令：

```
[root@localhost build]# make && make install
```

gcc编译的时长相当长，大约需要30分钟到1小时不等，或许更长。完成之后，查看一下版本号：

```
[root@localhost ~]# /usr/local/gcc-9.1.0/bin/gcc -v
```

显示版本号为9.1.0（GCC），即安装成功。

关键点4：环境变量

现在我们的系统中安装了两个GCC。使用默认的gcc命令，得到的版本依然是低版本。但是我们并不想使用这个低版本的GCC，所以还需要在用户环境中配置高版本的GCC的路径。

```
[root@localhost ~]# vim ~/.bash_profile
```

打开文件之后，按i键，进入插入模式，修改文件如下：

```
PATH=/usr/local/gcc-9.1.0/bin/:$PATH:$HOME/bin
export PATH
```



```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/gcc-9.1.0/lib64/  
export LD_LIBRARY_PATH
```

文件“.bash_profile”修改了当前账户的环境配置，我们将GCC-9.1.0的bin目录加入到path路径中。在Windows系统中增加环境变量时，用的分隔符为；（分号），在Linux下，用的分隔符为：（冒号）。PATH是执行路径的环境变量名，LD_LIBRARY_PATH是动态库的环境变量名，当有程序需要动态库时，就会在LD_LIBRARY_PATH指定的路径中查找。我们在这个变量中增加了GCC-9.1.0的新的库目录。而当我们执行某个程序时，则在PATH这个目录下查找。配置完成之后，退出并重新进入XShell终端，查看版本：

```
[root@localhost ~]# gcc -v
```

现在用的GCC已经变成了新安装的GCC高版本了。

A.6.2 CMake3

除了基本的编译器GCC之外，在Linux下编译C++文件还涉及Makefile文件。Makefile文件的作用就是把源码编译成中间码，再编译为可执行程序。Windows下使用的Visual Studio也是同样的编译步骤，Visual Studio把中间过程工具化了，写代码的人不清楚这个中间过程。

Linux下没有类似Visual Studio这种可视化工具，所以需要自己写Makefile文件。Makefile文件对于初学者来说比较烧脑，所以用工具CMake来自动生成Makefile。

```
[root@localhost ~]# yum install -y cmake3  
[root@localhost ~]# cmake3 --version  
cmake3 version 3.13.5
```

执行如上命令，安装CMake3。下面我们创建一个简单的任务，查看在Linux下，CMake3和GCC是如何进行代码编译的。打开本书源代码库中的00_01_gcc目录，该目录中有CMakeLists.txt和main.cpp两个文件。

main.cpp的代码如下：

```
#include <stdio>
int main( int argc, char *argv[] ) {
    printf("Hello, world!\n");
    return 0;
}
```

代码非常简单，打印一句“Hello world! ”。使用CMake编译源代码需要如下步骤：

(1) 进入源代码目录：

```
[root@localhost ~]# cd ./game/00_01_gcc/
[root@localhost 00_01_gcc]# ls
CMakeLists.txt  main.cpp
```

查看源代码目录，可以看到有两个文件：一个是用于CMake的CMakeLists.txt；另一个是源代码文件。

(2) 生成Makefile文件：

```
[root@localhost 00_01_gcc]# cmake3 ./
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
.....
```

执行cmake3时，提示了CXX的编译器为系统默认的4.8版本，要指定安装高版本的GCC，需要进行指定。

```
[root@localhost 00_01_gcc]# vim ~/.bash_profile
```

按i键，进入插入模式，增加两行代码：

```
export CC=/usr/local/gcc-9.1.0/bin/gcc
export CXX=/usr/local/gcc-9.1.0/bin/g++
```

编辑完成之后，按Esc键，退出编辑模式，输入“: wq”保存退出。注意是半角符号。关闭XShell连接，再次打开。使用“rm-f Makefile”命令删除已生成的Makefile文件。再次执行cmake3命令：

```
[root@localhost 00_01_gcc]# cmake3 ./
-- The C compiler identification is GNU 9.1.0
-- The CXX compiler identification is GNU 9.1.0
.....
[root@localhost 00_01_gcc]# ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.txt  m
```

完成之后，再次使用ls命令浏览一下目录，会发现多了一些临时文件，其中就有GCC需要的Makefile。“cmake3 ./”命令指定了要CMake的目录是当前目录，如果需要指定的目录位于上层目录，则使用“cmake3 ../xxx”。

在这些生成的中间文件中，其中重要的就是Makefile，Makefile是GCC编译的基础，Makefile文件可以手动写出来，它定义了一系列的规则来指定哪些文件需要编译、编译的前后顺序等，在网上可以搜索到很多教你写Makefile文件的教程。如果对此感兴趣，可以查看资料，这里不再详述。总之，我们使用CMake生成了Makefile。

(3) 使用make命令生成执行文件：

```
[root@localhost 00_01_gcc]# make
Scanning dependencies of target test
[ 50%] Building CXX object CMakeFiles/test.dir/main.cpp.o
[100%] Linking CXX executable test
[100%] Built target test
```

生成可执行文件需要用到make命令，使用make命令会去查找当前目录中的Makefile文件，并执行Makefile中的每一条命令，最终生成目标文件。执行完成之后，多了一个名为test的文件，它没有后缀。前面已经讲过，在Linux系统下不是依照文件后缀来确定文件类型的。现在来

执行一下看看效果。

(4) 运行：

```
[root@localhost 00_01_gcc]# ./test
Hello, world!
```

运行结果打印了“Hello, world!”。假设我们在CMakeLists.txt中把最后一行修改成了`add_executable(test.exe ${SRCS})`，使用“`rm -f CMakeCache.txt`”删除CMake的中间文件，CMake将会重新执行，再次执行make将会重新生成一个test.exe文件。或者将它修改为“`add_executable(test.test ${SRCS})`”，则会生成一个带.test后缀的文件，它依然是一个可执行文件。

```
[root@localhost 01_testgcc]# ./test.test
Hello, world!
```

以上步骤完成了一个项目的配置与编译，打开CMakeLists.txt，我们简单分析一下这个文件每一行的作用，如表A-6所示。

表A-6 CMakeLists常用命令表

命 令	说 明
<code>cmake_minimum_required (VERSION 2.8)</code>	因为每个 CMake 版本不同，可以提供的功能也不同，这里限制了 CMake 的最低版本不能低于 2.8 版本
<code>project(test)</code>	指定工程名字
<code>aux_source_directory(. SRCS)</code>	将当前目录“.”保存在 SRCS 变量中，作为源码目录。如果有多个源码目录，可以多次声明。这个过程与在 VS 界面上把文件拖到工程中是一样的
<code>include_directories(/usr/include)</code>	指定 include 目录。因为本例很简单，没有我们自己的 include 目录，所地这里使用了系统的 include 目录
<code>set(CMAKE_CXX_FLAGS "-Wall -std =C++14")</code>	基础编译选项，这里使用了 C++14 标准。C++11 与 C++14、C++17 都有不同的语法
<code>set(CMAKE_CXX_FLAGS_DEBUG "-D_DEBUG -O0 -g -gdb")</code>	在 DEBUG 模式下的编译选项。这里增加了 gdb，在 DEBUG 模式下可调试代码。 -O0 不要优化代码，方便调试。注意第一个是字母 O，第二个才是 Zero(0)
<code>set(CMAKE_CXX_FLAGS_RELEASE "-O3 -DNDEBUG")</code>	在 RELEASE 模式下的编译选项，去掉了多余的 DEBUG 信息。其中“-O3”是最高程度优化代码
<code>link_directories(/usr/local/lib)</code>	指定 link 的文件目录，即在这个目录下寻找库文件，如果有多个库文件目录，就可以多次声明。本例很简单，只指定了系统的库目录
<code>add_executable(test \${SRCS})</code>	生成一个名为 test 的执行文件，指定源代码目录变量在 SRCS 中

在CMakeList.txt文件中，我们使用到诸如aux_source_directory、link_directories、set这样的命令，这些命令都是CMake提供的，通过使用CMake，我们可以方便地在Linux中编译出想要的可执行文件。CMake中有一系列的变量，都可以通过set来赋值。简单说明一下CMake中可能使用到的其他命令：

(1) CMAKE_RUNTIME_OUTPUT_DIRECTORY：输出可执行文件的目录。

类似这样的输出目录在CMAKE中有3个，分别是：

- CMAKE_RUNTIME_OUTPUT_DIRECTORY，表示输出可执行文件的目录。
- CMAKE_LIBRARY_OUTPUT_DIRECTORY，表示输出动态库目录。
- CMAKE_ARCHIVE_OUTPUT_DIRECTORY，表示输出静态库目录。

例如，set (CMAKE_ARCHIVE_OUTPUT_DIRECTORY "../libs") 表示要把静态文件放到上一层目录的libs目录中。当然，也可以不指定输出目录，这时输出的文件在当前目录下。

(2) CMAKE_BUILD_TYPE：编译类型，常规的值是Debug、Release。

例如set (CMAKE_BUILD_TYPE Debug) 表示要编译的是一个Debug版本。

(3) CMAKE_CXX_FLAGS：因为我们编的是C++，C++的编译选项名在CMake中为CMAKE_CXX_FLAGS。编译参数这个值在CMake中的模板为CMAKE_<LANG>_FLAGS，如果这里是纯C语言，就是CMAKE_C_FLAGS。这个参数用于指定一些编译参数。

除此之外，还有一些扩展的属性，例如CMAKE_CXX_FLAGS_DEBUG，表示Debug版本下的参数；CMAKE_CXX_FLAGS_RELEASE则表示Release版本下的参数。使用Debug还是使用Release是由指定的CMAKE_BUILD_TYPE

来决定的。

因为我们使用的是C++，所以特别说明一下C++的编译选项，即CMAKE_CXX_FLAGS中的值。

不论我们使用CMake还是使用Makefile，在GNU编译器集合（GNU Compiler Collection, GCC）中都有两个环境变量，分别是CFLAGS与CXXFLAGS。CFLAGS用于C代码，CXXFLAGS用于C++代码。在使用CMake时，环境变量被缓存到CMAKE_CXX_FLAGS/CMAKE_C_FLAGS中，如果我们对CMAKE_CXX_FLAGS/CMAKE_C_FLAGS做了定义，那么环境变量就会被忽视。在上面的例子中，我们没有使用环境变量，而直接对CMAKE_CXX_FLAGS进行赋值，即这一句：set (CMAKE_CXX_FLAGS"-Wall-std=C++11")；如果要使用环境变量，可以这样：set (CMAKE_CXX_FLAGS"\$ENV{CXXFLAGS}-Wall-std=C++11")，\$ENV{CXXFLAGS}取得环境变量。更多的选项见表A-7。

表A-7 编译选项表

编译选项	说 明
-Wall	在编译时加上可以让编译器对你的代码提出尽可能多的警告
-std=C++14	以 C++14 的标准来编译源代码。C++14 与 C++11 或更早的 C++ 标准语法不同
-O0、-O3	优化程度调到最高为-O3，最低为-O0，即不做优化。注意，-O 是 OPT 的 O，而不是 0，-O0 第二个字符才是 0
-DNDEBUG	在 Release 时使用，表示不进行调试编译，生成的可执行文件没有调试信息，可以加快运行速度
-g -ggdb	生成 gdb 需要的调试信息
-D_DEBUG	这是自定义的一个预编译宏。在 Windows 系统的 Debug 模式下定义了_DEBUG 宏，以完成一些 DEBUG 的调试功能。为了统一性，我们在 Linux 下也定义一个相同的宏，一般来说在自定义宏前加-D，GCC 就可以识别这个宏

关于CMake更多的资料可以在<https://cmake.org/documentation/>上查看。如果是第一次接触CMake，这一章的信息量可能过大，我们再总结下一个基本的CMakeLists.txt需要做什么：

- (1) 指定源码目录，可以是一个或多个目录。
- (2) 指定include目录，可以是一个或多个目录。
- (3) 指定lib库目录，可以是一个或多个目录。
- (4) 指定输出目录，不同类型输出目录不相同，也可以不指定。
- (5) 指定编译类型。
- (6) 指定编译选项、预处理值等。

(7) 执行生成命令，生成可执行文件、静态库或动态库。

(8) 库连接。

细心的读者会发现，其实CMake和我们在Visual Studio下做的事情相差无几，Visual Studio是一个可视化的界面，而CMake是命令行。

本节用一个简单的例子来测试安装的GCC和CMake是否可以正常工作，并对CMake有了一个大致的了解。如果还不是太明白，也无须担心，本书的所有源代码已经写好了CMakeList.txt文件，结合本节所给出的知识点完全可以看懂每一个CMakeList.txt文件。在有特殊用法时，使用一些其他的配置也会根据上下文加以说明。

A.6.3 gdb

现在可以用CMake生成可执行文件了。有一个新的问题，没有Visual Studio的可视界面，我们怎么在Linux下调试代码呢？有一些纯服务端开发的程序，只在Linux上运行，它有很多Linux的特性，根本无法使用界面工具进行调试。即使源代码兼容Windows或Linux，因为其运行环境的不同、编辑器的不同，即使是相同的代码，有时也会有一点细微的差别，所以对于Linux系统下的调试是作为服务端程序的一个必备条件。如果还不熟悉，本小节准备了一些简单的例子供初学者翻阅；如果你已经熟悉gdb操作，可以跳过本小节。

关键点1：安装gdb

在Linux下使用gdb来进行调试，gdb是Linux上的标配调试器。可以使用命令进行安装：

```
[root@localhost ~]# yum install -y gdb
```

在调试代码时一般会定位到某个文件的某一行，但是由于Windows和Linux的换行符不一致，因此会造成行偏差。我们安装一个小工具doc2unit。

```
[root@localhost ~]# yum install -y dos2unix
```

如果在调试时发现某个文件（例如network.cpp）的换行符不对，可以执行如下命令，将换行符转换为Linux下的换行符。

```
[root@localhost network]# dos2unix network.cpp
```

关键点2：调试代码步骤

我们使用本书源代码00_01_gcc目录中的例子来说明如何用gdb调试代码。在执行文件前加上gdb的关键字，就可以开始一个可执行文件的调试：

```
[root@localhost 00_01_gcc]# gdb test
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-114.el7
.....
Reading symbols from /root/game/00_01_gcc/test...(no debugging sym
(gdb)
```

注意第一个提示（no debugging symbols found），原因是test中没有Debug调试信息，这是什么原因？我们生成test时没有指定Debug模式。gdb必须在Debug模式下才能良好运行。使用vim编辑CMakeLists.txt文件，命令为：vim CMakeLists.txt，在第三行处增加设置：set (CMAKE_BUILD_TYPE Debug)，表示以Debug来build工程。删除CMakeCache.txt文件，重新编译。

```
[root@localhost 00_01_gcc]# rm -rf CMakeCache.txt
[root@localhost 00_01_gcc]# cmake3 ./
[root@localhost 00_01_gcc]# make
```

在（gdb）提示符之后输入“r”，出现了“auto-load safe-path”警告，每次gdb启动时都会执行“.gdbinit”文件，我们创建一个放到root用户的根目录下。

```
[root@localhost 00_01_gcc]# vim /root/.gdbinit
```

认真看一下提示，.gdbinit的文件位置在不同的机器上可能不同。在“.gdbinit”文件中加入一行“add-auto-load-safe-path/usr/local/gcc-9.1.0/lib64/libstdC++.so.6.0.26-gdb.py”，保存退出。

再次使用“gdb test”命令，前面列出的是gdb相关的信息，在提示符（gdb）之后输入“r”命令，意味着让它继续运行Run。

```
(gdb) r
Starting program: /root/game/00_01_gcc/test
Hello, world!
[Inferior 1 (process 2113) exited normally]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-260
```

因为没有断点，所以程序直接运行，输出了“Hello, world!”，随后有一个debuginfos缺失的提示。我们按照以下步骤修改debuginfo缺失问题：

步骤01 使用Vim修改“/etc/yum.repos.d/CentOS-Debuginfo.repo”文件使enable=1。完整命令如下：

```
[root@localhost 00_01_gcc]# vim /etc/yum.repos.d/CentOS-Debuginfo.
```

步骤02 执行“yum install yum-utils”。在初始化CentOS的脚本中，已经安装过这个包，如果你没有执行过初始化脚本，就安装一次，如果本地已安装，执行时就会跳过。完整命令如下：

```
[root@localhost 00_01_gcc]# yum install yum-utils
```

步骤03 执行上面提示中use：后面的部分“debuginfo-install

glibc-2.17-260.el7_6.6.x86_64”。完整命令如下：

```
[root@localhost 00_01_gcc]# debuginfo-install glibc-2.17-260.el7_6
```

注意，这个命令不是绝对的，根据不同的版本，提示信息不一样，需要根据自己机器的提示信息来执行命令。安装完成之后，再次执行gdb调试：

```
[root@localhost 00_01_gcc]# gdb test
.....
Reading symbols from /root/game/00_01_gcc/test...done.
(gdb) r
Starting program: /root/game/00_01_gcc/test
Hello, world!
[Inferior 1 (process 1589) exited normally]
(gdb)
```

可以看到，这一次打印输出了“Hello, world!”之后，打印了进程退出的消息，没有什么异常打印，也没有警告。好了，我们在gdb中完成了一次完整的程序执行。下面介绍怎么打断点与查看数据。

关键点3：调试命令

程序运行到断点时，可以用gdb来查看栈上的信息。如果我们正在用gdb运行一个可执行程序，程序遇到宕机时，gdb会主动在宕机处停下来，查看栈信息也可以获得更多的宕机信息。首先列出gdb调试时常用的命令，见表A-8。

表A-8 gdb命令列表

命 令	说 明
l	查看当前行后面 10 行的代码
l 行号	查看行号前后 10 行的代码
l 起始行号 结束行号	查看起始到结束行的代码
break 文件名:行号	在给定的行号上打一个断点
break 函数名	为给定的函数打一个断点
r	开始调试
n (=next)	单步调试
s	单步调试，进入函数
c (= continue)	继续
clear 文件名:行号	清除一个 break
p 变量	查看变量
backtrace	打印全部栈的调用信息，按 Ctrl+C 组合键可终止打印
info local	显示当前栈局部变量的值

我们还是以00_01_gcc为例，使用gdb来调试。首先用gdb启动test：

```
[root@localhost 00_01_gcc]# gdb test
.....
Reading symbols from /root/game/00_01_gcc/test...done.
```

步骤01 我们先用l命令查看一下代码。

```
(gdb) l
1      #include <stdio>
2
3      int main( int argc, char *argv[] ) {
4          printf("Hello, world!\n");
5          return 0;
6      }
```

步骤02 在第4行加一个断点。

```
(gdb) break 4
Breakpoint 1 at 0x4006d1: file /root/game/00_01_gcc/main.cpp, line
```

步骤03 输入“r”开始调试，程序运行到第4行时会中断。

```
(gdb) r
Starting program: /root/game/00_01_gcc/test
Breakpoint 1, main (argc=1, argv=0x7fffffffe578) at /root/game/00_
4      printf("Hello, world!\n");
(gdb)
```

步骤04 向下单步执行“n”。

```
(gdb) n
Hello, world!
5      return 0;
```

在单步跟踪时，执行了第4行代码，同时打印出了第5行代码的源代码。

步骤05 如果我们不想再跟踪，或者需要程序直接运行到下一个断点，这时执行“c”继续向下执行：

```
(gdb) c
Continuing.
[Inferior 1 (process 1460) exited normally]
(gdb)
```

本例中后续没有再设置断点，程序就完成退出了，输入“q”退出gdb，一个简单的调试就完成了。

关键点4：跟踪变量

我们修改一下代码跟踪变量。使用vim修改main.cpp文件如下：

```
[root@localhost 01_testgcc]# vim main.cpp
#include <stdio>
int main( int argc, char *argv[] ) {
    int i = 1;
    printf("Hello, world!\n");
    i = 2;
    return 0;
}
```

修改完成之后直接make，因为没有增加或减少文件数量，Makefile不需要重新生成。GCC会给出一个警告，虽然定义了一个i变量，但是没有使用它。再来看看gdb的表现：

步骤01 输入l命令查看代码。

```
(gdb) l
1      #include <stdio>
2
3      int main( int argc, char *argv[] ) {
4          int i = 1;
5          printf("Hello, world!\n");
6          i = 2;
7          return 0;
8      }
```

步骤02 在第5行、第7行打断点。

```
(gdb) break 5
Breakpoint 1 at 0x4006d8: file /root/game/00_01_gcc/main.cpp, line
(gdb) break 7
Breakpoint 2 at 0x4006e9: file /root/game/00_01_gcc/main.cpp, line
```

步骤03 输入r命令执行程序，在第5行时打断点。

```
(gdb) r
Starting program: /root/game/00_01_gcc/test
Breakpoint 1, main (argc=1, argv=0x7fffffff578) at /root/game/00_
5   printf("Hello, world!\n");
```

步骤04 查看变量。

```
(gdb) info local
i = 1
(gdb) p i
$1 = 1
```

我们可以使用info或p命令。info local显示所有本地变量。p则指定变量名。此时，输入“info local”查看当前local的情况，i被赋值为1了。输入“c”继续运行到下一个断点处：

```
(gdb) c
Continuing.
Hello, world!
Breakpoint 2, main (argc=1, argv=0x7fffffff578) at /root/game/00_
7   return 0;
(gdb) info local
i = 2
```

在第7行断点处，再次输入“info local”，发现i被赋值为2了。

这个过程和在Visual Studio窗口界面上查看变量的值的过程是一样的，只是Visual Studio有界面，我们使用的是命令。

关键点5：堆栈

再次修改源代码查看堆栈的情况。如果你实在不习惯vim式的修改，可以在samba共享的目录中打开main.cpp，在Windows系统下修改文件。

```
[root@localhost 01_testgcc]# vim main.cpp
#include <stdio>
void fun() {
    printf("Hello, world!\n");
}
int main( int argc, char *argv[] ) {
    int i = 1;
    fun();
    i = 2;
    return 0;
}
```

在源代码中增加了一个fun函数，保存后再次执行make编译，进入gdb调试：

步骤01 查看源代码文件。

```
(gdb) l
1      #include <stdio>
2      void fun() {
3          printf("Hello, world!\n");
4      }
5      int main( int argc, char *argv[] ) {
6          int i = 1;
7          fun();
8          i = 2;
9          return 0;
10     }
```

步骤02 在函数内加入一个断点。

```
(gdb) break 3
Breakpoint 1 at 0x4006f6: file /root/game/00_01_gcc/main.cpp, line
```

步骤03 执行到断点位置。

```
(gdb) r
Starting program: /root/game/00_01_gcc/test
Breakpoint 1, fun () at /root/game/00_01_gcc/main.cpp:3
3   printf("Hello, world!\n");
```

步骤04 查看堆栈。

断点停下来之后，我们执行backtrace命令，即可看到当前的堆栈信息。当前只有两层堆栈，我们在当前堆栈下查看局部变量，因为没有局部变量，显示为“No locals”。

```
(gdb) backtrace
#0  fun () at /root/game/00_01_gcc/main.cpp:3
#1  0x000000000040071e in main (argc=1, argv=0x7fffffff578) at /r
(gdb) info local
No locals.
```

同时可以使用up n、down n命令来移动堆栈，其中n表示前进或后退的层数。退回到上一层。

```
(gdb) up 1
#1  0x000000000040071e in main (argc=1, argv=0x7fffffff578) at /r
7       fun();
(gdb) info local
i = 1
```

可以看到上一层的调用位于main.cpp的第7行，再打印“info local”的时候，变量i的值被打印出来了。继续执行输入“c”，退出输入“q”。好了，到目前为止，我们使用了几个简单的小例子，初步探索了gdb的使用。

除了上面这种调试方式外，gdb还可以对一个正在运行中的程序进行调试，简单可描述为两步：先找到这个程序的PID，再执行gdb attach PID。对一个正在运行中的程序调试是我们经常会遇到的事情，刚开始并没有以gdb的方式启动，但是程序在中途出现了错误，这时也需要调试。我们用一个例子来说明这个情况：

步骤01 修改main.cpp文件如下：

```
[root@localhost 00_01_gcc]# vim main.cpp
#include <stdio>
int main( int argc, char *argv[] ) {
    int i = 0;
    while(true){
        i++;
    }
    return 0;
}
```

退出vim，执行make生成新的test文件。执行test文件：

```
[root@localhost 00_01_gcc]# ./test
```

按回车键之后，就会发现test一直没有返回，当然不会返回，我们写了一个死循环。

步骤02 在XShell上再打开一个终端连接到虚拟机上。

```
[root@localhost ~]# ps -ef|grep test
root      1948   1881  99 18:06 pts/0    00:02:42 ./test
root      1976   1952   0 18:09 pts/1    00:00:00 grep --color=auto
```

此时输入“ps-ef|grep test”命令，查看有没有名为test的进程，ps命令是Process Status的缩写。第一个就是刚才运行的test进程。显示数据的前3项比较重要，分别是UID、PID和PPID。UID是账号ID，PID是进程ID，而PPID是该进程的父进程ID。现在找到了test的PID为1948。

步骤03 执行gdb。

```
[root@localhost ~]# gdb attach 1948
.....
Attaching to process 1948
.....
0x0000000000400698 in main (argc=1, argv=0x7fff1c0173c8) at /root/
5      i++;
(gdb) info local
i = 20783
```

当我们跟进进程时，它正好执行到了main.cpp的第5行。这时这个循环已经执行了2万多次了。在使用attach命令退出时，需要使用detach命令将进程与gdb剥离。这时可以使用各种命令查看线程或进程信息，这里不再逐一介绍了。

关键点6: Core

在某些情况下，在Linux系统中运行程序是直接执行的，没有用到gdb，但是程序发生了宕机，这时可以查看内核生成的Core文件来查看宕机时的信息。首先确认是否打开了Core文件的开关。

```
[root@localhost ~]# ulimit -c
0
```

命令“ulimit-c”，如果结果为0，就表示关闭了Core生成。

执行“ulimit-c unlimited”，打开Core文件生成，生成的Core文件大小无限制。如果需要限制“ulimit-c size”，size的单位是Kb。

```
[root@localhost ~]# ulimit -c unlimited
[root@localhost ~]# ulimit -c
unlimited
```

在终端执行上面的命令，仅在本次会话中有效果。如果要长期打开Core，就需要修改profile文件。执行如下命令：

```
[root@localhost ~]# echo "ulimit -c unlimited" >> /etc/profile
[root@localhost ~]# source /etc/profile
[root@localhost ~]# ulimit -c
unlimited
```

以上3个命令实现的功能是，将命令写入profile文件中重新加载，让它立即生效。下面还用了一个例子来说明Core文件的用途。使用的工程为本书源代码库中的00_02_gdb工程，用以下两步进行编译：

```
[root@localhost 00_02_gdb]# cmake3 ./
[root@localhost 00_02_gdb]# make
```

在执行make命令时，GCC会抛出一个警告，忽视这个警告信息。Make之后，生成可执行文件core_null_d。此处的“_d”是故意加上去的，以标志该执行文件为debug模式。执行文件生成之后，开始查看Core文件的步骤：

步骤01 执行core_null_d，会出现一个段错误。

```
[root@localhost 00_02_gdb]# ./core_null_d
段错误(吐核)
[root@localhost 00_02_gdb]# ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.txt  c
```

使用ls查看，在本地生成一个core.XXXX的文件，XXXX部分是PID，所以每次可能不同。

步骤02 使用gdb命令，其格式为：gdb可执行文件core文件。

```
[root@localhost 00_02_gdb]# gdb core_null_d core.2485
.....
Program terminated with signal 11, Segmentation fault.
#0  0x00000000004006ed in main () at /root/game/00_02_gdb/main.cpp
6      *pChar = '0';
```

可以看到gdb信息中列出了宕机的代码位置，就在main.cpp的第6行。

步骤03 如果上一步不够详细，可以使用bt（backtrace）命令列出堆栈：

```
(gdb) bt
#0  0x00000000004006ed in main () at /root/game/00_02_gdb/main.cpp
```

这一行就是出错的行，对应代码是*pChar = '0'。因为pChar指针是一个空指针，所以这里抛出了异常。

CMake、gdb都是Linux编程中必备的知识点，由于篇幅有限，有一个大致的印象即可，需要深入学习时可以阅读一些相关的资料。

A.6.4 安装数据库环境

本书要完成一个完整的服务端框架体系，我们需要用到数据库。在CentOS的仓库中使用的是MariaDB。MariaDB是MySQL的一个分支，这个分支形成的原因是MySQL被Oracle收购之后，可能面临着闭源。MariaDB就是MySQL闭源时的替代品，它依然是一个免费开源的数据库，由MySQL原作者开发维护。MariaDB的命令与MySQL基本是一致的，在性能上官方认为MariaDB可能更优于原MySQL。执行安装与启动命令：

```
[root@localhost ~]# yum install -y mariadb-server mariadb mariadb-
[root@localhost ~]# systemctl start mariadb // 启动服务
```

```
[root@localhost ~]# mysql // 执行
Server version: 5.5.60-mariadb mariadb Server
mariadb [(none)]>
.....
```

出现提示符说明已经安装成功了。我们在MariaDB提示符输入“exit”，命令退回到Shell界面下，对MySQL进行配置。

关键点1：配置

在使用MariaDB之前，需要对MariaDB进行配置。执行如下命令：

```
[root@localhost ~]# mysqlL_secure_installation
```

(1) Enter current password for root (enter for none)：输入当前密码，当前默认的密码为空，所以这里直接按回车键。

(2) Set root password? [y/n]：是否设置root密码，输入“y”，然后输入密码。注意，输入密码时在Linux端是看不到任何表达的，也就是说不会有****这样的字符，输入完成后按回车键再次确认。

(3) Remove anonymous users? [y/n]：是否删除匿名用户，输入“y”。

(4) Disallow root login remotely? [y/n]：禁止root远程登录，输入“n”。在例子中需要远程登录，所以这里选择了“n”。

(5) Remove test database and access to it? [y/n]：是否删除测试数据库以及权限，输入“y”。

(6) Reload privilege tables now? [y/n]：是否重新加载权限表，输入“y”。

执行了以上6步，Shell就会提示你成功完成。

关键点2：开放端口

我们还需要将3306端口开放出来，并重新加载防火墙。如果没有防火墙，那么可以省略这一步。

```
[root@localhost ~]# firewall-cmd --zone=public --add-port=3306/TCP
```

输入firewall-cmd命令，如果提示如下：

(1) FirewallD is not running: 表示没有防火墙，后续步骤可省略。

(2) success: 需要重启防火墙。命令如下：

```
[root@localhost ~]# firewall-cmd --reload
```

关键点3：权限

再次进入MySQL，这次因为有了密码，所以需要输入账号与密码：

```
[root@localhost ~]# mysql -u root -p
Enter password:
```

输入密码之后，为root用户授权，分为两步：

```
mariadb [(none)]> grant all on *.* to 'root'@'%' identified by '123456'
Query OK, 0 rows affected (0.00 sec)
mariadb [(none)]> flush privileges;
Query OK, 0 rows affected (0.00 sec)
```

第一条命令是授权，其中123456为用户密码；第二条命令让授权生效。现在，我们可以在Windows端用navicat这类的软件连接一下DB，测

试连接是否能够成功。

关键点4：字符集

使用数据库时，字符集是一个比较敏感的问题，我们查看一下默认的字符集：

```
mariadb [(none)]> show variables like 'ch%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| character_set_client | utf8 |
| character_set_connection | utf8 |
| character_set_database | latin1 |
| character_set_filesystem | binary |
| character_set_results | utf8 |
| character_set_server | latin1 |
| character_set_system | utf8 |
| character_sets_dir | /usr/share/MySQL/charsets/ |
+-----+-----+
8 rows in set (0.00 sec)
```

数据库默认的database和server的字符集都是latin，但是在程序中需要utf8字符集。在创建数据库时需要注意，后面会在程序中将数据库修改为utf8。再来看一下collation：

```
mariadb [(none)]> show variables like 'collation%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| collation_connection | utf8_general_ci |
| collation_database | latin1_swedish_ci |
| collation_server | latin1_swedish_ci |
+-----+-----+
3 rows in set (0.00 sec)
```

ci为case insensitive的缩写，即对大小写不敏感。也就是说，Test账号与test账号是同一个账号。

关键点5：开机启动

一切配置完成之后，我们将数据库服务加入开机启动服务中：

```
[root@localhost ~]# systemctl enable mariadb
Created symlink from /etc/systemd/system/multi-user.target.wants/
```

如果不确定是否已经开机列表中，那么可以查看一下：

```
[root@localhost ~]# systemctl list-unit-files --type service --sta
UNIT FILE          STATE
.....
mariadb.service enabled
```

Enabled表示MariaDB的服务已经在开机列表中了。