

RoboMaster

视觉从入门到入土

Misaka21

Turn drawings of imagination into racetrack realization.

June 25, 2025

Thank you for using this book ❤️,
I hope you like it 😊

目录

1. 基础篇	6
1.1. 初识 Linux	6
1.1.1. Linux 的前世今生	6
1.1.2. 为什么 RoboMaster 开发选择 Linux	9
1.1.3. Linux 与 Windows 的差异	9
1.1.4. Linux 发行版选择	9
1.1.5. 安装 Linux 的方式	9
1.2. 环境准备	9
1.3. Linux & Ubuntu 操作	9
1.3.1. 图形界面初探	9
1.3.2. 终端与 Shell 基础	9
1.3.3. 文件系统与目录结构	9
1.3.4. 文件与目录操作	9
1.3.5. 文本编辑与处理	9
1.3.6. 用户与权限管理	9
1.3.7. 软件包管理	9
1.3.8. 进程与系统监控	9
1.3.9. 网络与远程访问	9
1.3.10. Shell 脚本入门	9
1.3.11. 开发环境配置	9
1.4. 计算机系统基础	9
1.4.1. 为什么需要了解计算机系统	9
1.4.2. 程序的执行：从源码到运行	9
1.4.3. 进程：程序的运行实例	13
1.4.4. 线程：轻量级执行单元	22
1.4.5. 内存层次结构	30
1.4.6. 虚拟内存	30
1.4.7. 进程的内存布局	30
1.4.8. 文件系统	30
1.4.9. 网络通信基础	30
1.4.10. 并发与同步	30
1.4.11. 操作系统的角色	30
1.4.12. 性能分析与调试工具	30
1.5. C++ 语言基础	30
1.5.1. C++ 基本程序结构	30
1.5.2. 编译一个 C++ 程序	32
1.5.3. 变量与基本数据类型	33
1.5.4. 运算符与表达式	38

1.5.5. 控制语句	43
1.5.6. 函数基础	51
1.5.7. 数组与字符串	58
1.5.8. 指针基础	65
1.5.9. 引用	72
1.5.10. 结构体	78
1.5.11. 类与对象	84
1.5.12. 构造函数与析构函数	95
1.5.13. 拷贝控制	108
1.5.14. 类的继承	118
1.5.15. 多态与虚函数	130
1.5.16. 运算符重载	142
1.5.17. STL 容器	156
1.5.18. STL 算法与迭代器	168
1.5.19. 模板基础	181
1.5.20. 智能指针	193
1.5.21. 右值引用与移动语义	208
1.5.22. Lambda 表达式	220
1.5.23. 多线程基础	231
1.5.24. 文件操作	245
1.5.25. Eigen 矩阵库	264
1.5.26. Ceres 非线性优化库	277
1.5.27. OpenCV 基础	293
1.5.28. 串口通信	293
1.6. CMake 与构建系统	293
1.7. 软件工程基础	293
1.8. 软件工程基础	293
1.8.1. 为什么需要软件工程	293
1.8.2. 代码规范与风格	293
1.8.3. 设计模式的起源	304
1.8.4. 设计模式的基本原则	305
1.8.5. 常用设计模式详解	306
1.8.6. 单元测试	306
1.8.7. 调试技巧	306
1.8.8. 性能分析与优化	306
1.8.9. 文档编写	306
1.8.10. 版本控制与协作（可选/扩展）	320
1.9. Git 版本控制	320
1.10. ROS/ROS2 入门	320
2. 数学理论篇	321

2.1. 计算机视觉基础	321
2.2. 传统视觉算法	321
2.3. 深度学习和神经网络	321
2.4. 相机模型	321
2.5. 坐标变换	321
2.6. 卡尔曼滤波	321
2.6.1. 什么是卡尔曼滤波?	321
2.6.2. 如何准确地测量体重	321
2.6.3. 从体重秤到赛车: 静态估计的局限性	323
2.6.4. 最优增益的推导	326
2.6.5. 多维卡尔曼滤波	331
2.6.6. 扩展卡尔曼滤波	336
2.6.7. 无迹卡尔曼滤波	342
3. 实战技术篇	349
3.1. 通信协议设定	349
3.2. 相机标定与手眼标定	349
3.3. 时间戳对齐	349
3.4. 弹道解算	349
4. RoboMaster 应用篇	350
4.1. 工业相机	350
4.2. 装甲板识别	350
4.3. 装甲板的跟踪	350
4.4. 能量机关识别	350
4.5. 自瞄算法设计	350
4.6. 串口模块	350
5. 进阶篇	351
5.1. 雷达站视觉方案	351
5.2. 哨兵决策视觉	351
5.3. 性能优化技巧	351
5.4. 实战经验与坑点总结	351
6. 项目分析	352
6.1. rm_vision	352
6.2. rm.cv.fans	352
6.3. 同济自瞄	352
Index of Tables	353
Index of Listings	353

1. 基础篇

本书主要针对基础篇的内容。

1.1. 初识 Linux

1.1.1. Linux 的前世今生

在开始学习 Linux 的具体操作之前，了解它的历史背景会帮助你更好地理解这个系统的设计理念。Linux 不是凭空出现的，它继承了 Unix 数十年的智慧，融合了自由软件运动的理想，最终在全球开发者的协作下成长为今天这个无处不在的操作系统。这段历史不仅仅是技术的演进，更是一群人对“软件应该如何被创造和分享”这个问题的回答。

1.1.1.1. Unix：一切的起源

故事要从 1969 年说起。那一年，人类首次登上月球，而在美国新泽西州的贝尔实验室，两位计算机科学家 Ken Thompson 和 Dennis Ritchie 正在做一件同样具有开创性的事情——他们在一台被闲置的 PDP-7 小型机上，开发一个全新的操作系统。这个系统后来被命名为 Unix。

Unix 的诞生带有几分偶然。在此之前，贝尔实验室参与了一个名为 Multics 的大型操作系统项目，但这个项目因为过于复杂而进展缓慢，贝尔实验室最终选择退出。Thompson 和 Ritchie 对 Multics 的一些理念仍然着迷，于是决定自己动手，用更简洁的方式实现类似的功能。Unix 这个名字本身就是对 Multics 的一种调侃——Multics 意为“多路复用”，而 Unix 则暗示“单一”，讽刺 Multics 的过度复杂。

Unix 的设计体现了一种独特的哲学，这种哲学至今仍深刻影响着软件开发的方式。其核心理念可以概括为几个原则：每个程序只做一件事，并把它做好；程序之间通过文本流协作，一个程序的输出可以作为另一个程序的输入；尽早编写原型，快速迭代改进。这种“小而美”的设计思想与当时流行的“大而全”的系统形成鲜明对比。当你在 Linux 终端中用管道符 | 将多个命令串联起来时，你正在使用的就是 Unix 哲学的直接体现。

1972 年，Dennis Ritchie 完成了另一项影响深远的工作——他发明了 C 语言，并用 C 语言重写了 Unix。这使得 Unix 成为第一个主要用高级语言编写的操作系统，大大提高了它的可移植性。在此之前，操作系统通常用汇编语言编写，与特定硬件紧密绑定。C 语言的出现让 Unix 可以相对容易地移植到不同的计算机上，这为它后来的广泛传播奠定了基础。

贝尔实验室隶属于 AT&T 公司，当时 AT&T 受到反垄断法规的限制，不能进入计算机业务，因此以极低的成本将 Unix 授权给大学和研究机构。这使得 Unix 在学术界迅速流行起来。加州大学伯克利分校的学生和研究人员在 Unix 的基础上进行了大量改进和扩展，形成了著名的 BSD (Berkeley Software Distribution) 版本。许多我们今天仍在使用的网络功能，包括 TCP/IP 协议栈的实现，都源自 BSD。

然而，好景不长。1984 年，AT&T 被拆分后获准进入计算机市场，Unix 开始商业化，授权费用大幅上涨。与此同时，各个公司和机构开发的 Unix 变体开始分化，互不兼容。这个时期出现了许多商业 Unix 系统：Sun 的 Solaris、IBM 的 AIX、HP 的 HP-UX 等。Unix 世界陷入了碎片化和专利诉讼的泥潭，曾经开放共享的精神逐渐消失。

1.1.1.2. GNU 与自由软件运动

在 Unix 走向封闭的同时，一位 MIT 人工智能实验室的程序员决定反抗这一趋势。他的名字是 Richard Stallman，人们通常称他为 RMS。

Stallman 是一位极具原则性的人。在他看来，软件应该是自由的——用户应该有权运行、研究、修改和分发软件。商业公司将软件封闭起来、禁止用户查看和修改源代码的做法，在他眼中是对用户自由的侵犯。他常常讲述一个故事：实验室的打印机经常卡纸，如果能看到驱动程序的源代码，他可以轻松添加一个卡纸通知功能；但新打印机的驱动程序是闭源的，他无能为力。这种“被软件控制而无法控制软件”的无力感，促使他下定决心做出改变。

1983 年，Stallman 发起了 GNU 计划。GNU 是“GNU’s Not Unix”的递归缩写——一个典型的黑客式幽默。这个计划的目标是创建一个完全自由的、与 Unix 兼容的操作系统。所谓“自由”，Stallman 定义了四项基本自由：自由运行程序、自由研究程序如何工作（需要源代码）、自由分发程序的副本、自由改进程序并发布改进版本。注意，这里的“自由”（free as in freedom）不是“免费”（free as in free beer）的意思，尽管自由软件通常也是免费的。

1985 年，Stallman 创立了自由软件基金会（Free Software Foundation, FSF），并发明了 GPL（GNU General Public License）许可证。GPL 的核心条款是“copyleft”——如果你分发基于 GPL 软件的衍生作品，你必须以同样的 GPL 条款发布，包括提供源代码。这确保了自由软件的自由能够传递下去，不会被人拿去变成专有软件。

在接下来的几年里，GNU 计划开发了大量高质量的软件工具：GCC 编译器、GDB 调试器、Emacs 编辑器、GNU Make、Bash Shell，以及众多的命令行工具（ls、cat、grep 等）。这些工具至今仍是 Linux 系统的核心组成部分，也是每个 RoboMaster 开发者每天都在使用的工具。你用 g++ 编译代码、用 make 构建项目、用 bash 编写脚本，使用的都是 GNU 软件。

然而，到了 1990 年代初，GNU 计划仍然缺少一个关键组件——操作系统内核。Stallman 团队正在开发一个名为 GNU Hurd 的内核，但由于采用了过于前卫的微内核架构，开发进度非常缓慢（事实上，Hurd 至今仍未完成）。GNU 拥有了构建完整操作系统所需的几乎所有工具，却唯独缺少让这些工具运行起来的内核。这就像拥有了所有的汽车零件，却没有发动机。

1.1.1.3. Linux 内核的诞生

1991 年 8 月 25 日，芬兰赫尔辛基大学的一名 21 岁学生在 Usenet 新闻组发布了一条消息：

> Hello everybody out there using minix - > I’m doing a (free) operating system (just a hobby,
won’t be big and professional like gnu) for 386(486) AT clones.

这个学生的名字是 Linus Torvalds，他正在开发的这个“业余爱好项目”，就是后来的 Linux 内核。

Torvalds 开发 Linux 的初衷很简单：他想在自己新买的 386 电脑上运行 Unix，但商业 Unix 太贵，适合教学的 Minix 系统功能又太弱。于是他决定自己写一个。与 Stallman 的理想主义不同，Torvalds 的动机更加实用——他只是想要一个好用的系统。

Linux 的开发速度令人惊讶。Torvalds 将源代码发布在网上，邀请其他人参与改进。来自世界各地的程序员通过互联网协作，报告 bug、提交补丁、添加新功能。这种分布式协作开发的模式在当时是前所未有的。Torvalds 后来将 Linux 内核以 GPL 许可证发布，正式加入了自由软件的阵营。

Linux 内核与 GNU 工具的结合，终于形成了一个完整的自由操作系统。严格来说，我们日常使用的“Linux”应该称为“GNU/Linux”——它使用 Linux 内核，但用户空间的大部分工具来自 GNU 计划。Stallman 一直坚持这个称呼，以强调 GNU 项目的贡献，但在日常使用中，人们通常简称为“Linux”。

Linux 的成功有多重因素。技术上，它采用了成熟的单内核 (monolithic kernel) 架构，性能优异且稳定。许可证上，GPL 保证了代码的开放性，任何人都可以自由使用和修改。时机上，互联网的兴起让全球协作成为可能，也创造了对可靠服务器操作系统的巨大需求。社区上，Torvalds 的务实态度和技术领导力吸引了大量优秀的贡献者。

1.1.1.4. 从爱好项目到主导世界

如果 1991 年的 Torvalds 能够预见 Linux 的未来，他一定会惊讶不已。三十多年后的今天，Linux 已经成为世界上最成功的操作系统之一：

全球大多数的服务器运行着 Linux，包括支撑互联网运转的网站、云服务和数据中心。当你使用搜索引擎、观看在线视频或者发送消息时，你的请求几乎必然是由某个 Linux 服务器处理的。

世界上最快的 500 台超级计算机，100% 运行 Linux。科学研究、气象预测、人工智能训练，这些需要巨大计算能力的任务，都依赖于 Linux。

Android 系统基于 Linux 内核，这意味着全球数十亿部智能手机都在运行 Linux 的变体。

嵌入式领域同样是 Linux 的天下。从路由器、智能电视到汽车信息娱乐系统，从树莓派到 NVIDIA Jetson，Linux 无处不在。RoboMaster 机器人上常用的 Jetson 系列计算平台，运行的正是基于 Ubuntu 的 Linux 系统。

如今，Linux 内核的开发已经成为人类历史上最大规模的协作项目之一。数以千计的开发者来自数百家公司，包括 Google、Microsoft、Intel、Red Hat、华为等，共同贡献代码。曾经对 Linux 嗤之以鼻的微软，现在不仅是内核的主要贡献者之一，还在 Windows 中集成了 WSL (Windows Subsystem for Linux)，让用户可以在 Windows 上直接运行 Linux 程序。

Linux 的成功证明了开源协作模式的力量。一个芬兰大学生的业余项目，在全球开发者的共同努力下，成长为支撑现代数字世界的基础设施。当你开始学习 Linux，你不仅是在学习一个操作系统的使用方法，更是在加入一个拥有数十年历史、影响深远的技术传统。Unix 的设计哲学、GNU 的自由精神、全球社区的协作文化——这些都将在你的学习和实践中逐渐体现。

对于 RoboMaster 开发者来说，理解这段历史还有另一层意义。机器人开发所依赖的几乎所有核心工具——ROS、OpenCV、GCC、CMake、Git——都是开源软件，都植根于 Linux 和自由软件的传统。当你使用这些工具时，你是这个庞大开源生态的受益者；当你有朝一日贡献自己的代码时，你也将成为这个传统的一部分。

1.1.2. 为什么 RoboMaster 开发选择 Linux

1.1.3. Linux 与 Windows 的差异

1.1.4. Linux 发行版选择

1.1.5. 安装 Linux 的方式

1.2. 环境准备

1.3. Linux & Ubuntu 操作

1.3.1. 图形界面初探

1.3.2. 终端与 Shell 基础

1.3.3. 文件系统与目录结构

1.3.4. 文件与目录操作

1.3.5. 文本编辑与处理

1.3.6. 用户与权限管理

1.3.7. 软件包管理

1.3.8. 进程与系统监控

1.3.9. 网络与远程访问

1.3.10. Shell 脚本入门

1.3.11. 开发环境配置

1.4. 计算机系统基础

1.4.1. 为什么需要了解计算机系统

1.4.2. 程序的执行：从源码到运行

当你在终端输入 `./my_robot` 并按下回车，机器人程序便开始运行。但在这简单的操作背后，隐藏着一段漫长的旅程——从你编写的 C++ 源代码到 CPU 能够执行的机器指令，中间经历了多个精密的转换阶段。理解这个过程，不仅能帮助你解读编译器的错误信息、解决链接问题，还能让你明白为什么某些优化手段有效、为什么 ROS 节点启动需要时间。

1.4.2.1. 从源代码到可执行文件

我们编写的 C++ 代码是人类可读的文本，但计算机只能执行二进制的机器指令。将源代码转换为可执行文件的过程称为构建 (build)，它由四个阶段组成：预处理、编译、汇编和链接。虽然我们通常用一条 `g++ main.cpp -o main` 命令完成整个过程，但理解每个阶段的作用对于排查问题至关重要。

预处理是构建的第一步。预处理器处理所有以 # 开头的指令：#include 将头文件的内容直接插入到源文件中，#define 进行文本替换，#ifdef 等条件编译指令决定哪些代码参与后续编译。预处理的结果仍然是 C++ 代码，只是所有的宏都被展开、所有的头文件都被包含进来。你可以用 g++ -E main.cpp -o main.i 查看预处理的结果，会发现一个简单的包含了 <iostream> 的程序会膨胀到数万行——这就是标准库头文件展开后的样子。

```
// 原始代码
#include <iostream>
#define PI 3.14159

int main() {
    std::cout << PI << std::endl;
    return 0;
}

// 预处理后（简化示意）
// ... 数万行 iostream 的内容 ...

int main() {
    std::cout << 3.14159 << std::endl; // PI 被替换
    return 0;
}
```

编译阶段将预处理后的 C++ 代码转换为汇编代码。这是最复杂的阶段，编译器需要进行词法分析、语法分析、语义分析，构建抽象语法树，然后进行各种优化，最终生成目标平台的汇编指令。编译器在这个阶段检查语法错误和类型错误，这就是为什么“编译错误”通常指的是代码本身的问题。使用 g++ -S main.cpp -o main.s 可以查看生成的汇编代码。

```
; 编译生成的汇编代码片段 (x86-64)
main:
    push    rbp
    mov     rbp, rsp
    mov     edi, OFFSET FLAT:_ZSt4cout
    mov     esi, 3
    call    _ZNStolsEi          ; 调用 operator<<
; ...
```

汇编阶段将汇编代码转换为机器码，生成目标文件 (object file，扩展名通常是 .o)。汇编器的工作相对简单，基本上是汇编指令到机器指令的一一对应转换。目标文件包含了机器码，但还不能直接执行，因为其中的函数调用和全局变量引用还没有确定最终地址。

链接是构建的最后一步，也是最容易出问题的一步。链接器将多个目标文件以及需要的库文件合并成一个可执行文件。它的核心工作是符号解析 (symbol resolution) 和重定位 (relocation)。符号解析负责将代码中的函数调用和变量引用与它们的实际定义匹配起来；重定位则是计算并填入这些符号的最终内存地址。

```
# 分步构建过程
g++ -E main.cpp -o main.i      # 预处理
g++ -S main.i -o main.s        # 编译
g++ -c main.s -o main.o        # 汇编
g++ main.o -o main             # 链接
```

当链接器找不到某个符号的定义时，就会报“undefined reference”错误。这通常意味着你忘记链接某个库、忘记编译某个源文件，或者函数声明与定义不匹配。例如，在 RoboMaster 项目中，

如果你使用了 OpenCV 但忘记在 CMakeLists.txt 中链接 \${OpenCV_LIBS}，就会在链接阶段遇到大量的 undefined reference 错误。

1.4.2.2. 可执行文件的结构

链接完成后，我们得到一个可执行文件。在 Linux 上，可执行文件通常采用 ELF (Executable and Linkable Format) 格式。这个文件不是简单的机器指令堆砌，而是有着精心设计的结构，包含多个段 (section)，每个段存储不同类型的数据。

代码段 (text segment, 也叫 .text) 存储编译后的机器指令。这部分内容是只读的，因为程序执行时不应该修改自己的指令。将代码段设为只读还有安全上的考虑——它可以防止某些类型的攻击修改程序逻辑。

数据段 (data segment, .data) 存储已初始化的全局变量和静态变量。例如，`int globalVar = 42;` 这样的定义，变量的初始值 42 就存储在数据段中。程序启动时，这些值会被加载到内存的相应位置。

BSS 段 (.bss, Block Started by Symbol) 存储未初始化的全局变量和静态变量。例如 `int uninitializedVar;` 这样的定义。与数据段不同，BSS 段在可执行文件中并不占用实际空间——文件只记录这些变量的大小，程序加载时操作系统会分配相应的内存并初始化为零。这是一个聪明的优化：如果你定义了一个很大的未初始化数组，可执行文件不会因此变大。

```
int initialized = 100;           // 存储在 .data 段
int uninitialized;             // 存储在 .bss 段（加载时清零）
const char* message = "Hello";  // 指针在 .data, 字符串在 .rodata

void function() {               // 代码在 .text 段
    static int counter = 0;     // 存储在 .data 段
    int localVar = 5;          // 运行时存储在栈上，不在可执行文件中
}
```

除了这些主要的段，可执行文件还包含符号表（记录函数和变量的名称与地址）、重定位信息、调试信息等。使用 `readelf -S executable` 可以查看可执行文件的所有段，使用 `objdump -d executable` 可以反汇编查看机器码。

理解这些段的区别有助于优化程序大小。如果你的嵌入式程序太大，可以检查是否有过多的已初始化全局数据；如果内存不足，可以检查是否有过大的静态数组。在 RoboMaster 的嵌入式开发中，Flash 存储代码和初始化数据，RAM 存储 BSS 和运行时数据，了解这些对于资源受限的单片机编程尤为重要。

1.4.2.3. 程序的加载与执行

可执行文件存储在磁盘上，要运行它，操作系统需要将其加载到内存中。这个过程由操作系统的加载器 (loader) 完成，它远比简单的“复制到内存”要复杂。

当你执行一个程序时，操作系统首先为它创建一个新的进程，并建立虚拟地址空间。虚拟地址空间让每个进程都以为自己独占整个内存，实际上操作系统通过页表将虚拟地址映射到物理内存。接着，加载器读取可执行文件的头部信息，了解各个段应该加载到虚拟地址空间的什么位置。

现代操作系统通常采用“按需加载” (demand paging) 策略。加载器并不会立即将整个程序复制到内存，而是建立好内存映射关系，当程序实际访问某个页面时，如果该页面还不在内存中，

就会触发“缺页中断”（page fault），操作系统此时才从磁盘读取相应的内容。这种懒加载策略使得程序启动更快，也更节省内存——对于大型程序，可能只有一部分代码会被实际执行。

加载完成后，操作系统将 CPU 的程序计数器设置为程序的入口点（entry point），程序开始执行。在执行 `main` 函数之前，运行时库的启动代码会先执行，完成一些初始化工作：设置栈、初始化堆、调用全局对象的构造函数、设置 C++ 异常处理机制等。当 `main` 函数返回后，运行时库还会调用全局对象的析构函数，然后通过系统调用告知操作系统进程结束。

程序加载流程：

1. 创建进程，建立虚拟地址空间
2. 解析 ELF 头，映射各段到虚拟地址
3. 如果是动态链接程序，加载动态链接器
4. 动态链接器加载所需的共享库
5. 执行初始化代码（全局构造函数等）
6. 跳转到 `main` 函数开始执行

1.4.2.4. 静态链接与动态链接

当程序使用库（如标准库、OpenCV、Eigen）时，链接器需要将库中的代码与你的代码合并。根据合并的方式，链接分为静态链接和动态链接两种。

静态链接在构建时将库的代码直接复制到可执行文件中。生成的可执行文件是自包含的，不依赖外部库文件，可以直接复制到其他机器上运行。静态库在 Linux 上扩展名为 `.a`，在 Windows 上为 `.lib`。静态链接的缺点是可执行文件体积较大，而且如果多个程序使用同一个库，每个程序都会包含一份库的副本，浪费磁盘空间。更重要的是，当库需要更新（如修复安全漏洞）时，所有静态链接的程序都需要重新编译。

动态链接将库的代码保留在独立的共享库文件中（Linux 上是 `.so` 文件，Windows 上是 `.dll`），程序运行时才加载这些库。可执行文件只记录需要哪些共享库以及需要哪些符号，实际的库代码在运行时由动态链接器加载。多个程序可以共享同一个库文件，既节省磁盘空间，又节省内存（操作系统可以让多个进程共享同一份库代码的物理内存）。库更新时，只需替换共享库文件，无需重新编译程序。

```
# 查看程序依赖的共享库
ldd ./my_robot

# 输出示例:
# linux-vdso.so.1
# libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6
# libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
# libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
# libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
# libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
```

动态链接也有代价。程序启动时，动态链接器需要定位并加载所有依赖的共享库，解析符号引用，这会增加启动时间。对于依赖大量共享库的程序，这个启动开销可能相当可观。此外，运行时动态链接还需要通过间接跳转表（PLT/GOT）调用库函数，有轻微的性能损失，不过现代系统通过延迟绑定等优化技术已将这种损失降到很低。

在 CMake 中，可以控制生成静态库还是共享库：

```
# 生成静态库
add_library(mylib STATIC src1.cpp src2.cpp)
```

```
# 生成共享库  
add_library(mylib SHARED src1.cpp src2.cpp)  
  
# 链接到可执行文件  
target_link_libraries(my_program mylib)
```

1.4.2.5. 为什么 ROS 节点启动需要时间

了解了程序的加载过程，我们就能理解为什么 ROS 节点启动往往需要几百毫秒甚至更长时间。一个典型的 ROS 节点依赖大量的共享库：ROS 核心库、消息序列化库、通信库、OpenCV、Eigen、PCL 等。程序启动时，动态链接器需要依次加载这些库，解析成千上万个符号引用。

你可以用 `time` 命令测量程序的启动时间，用 `LD_DEBUG=statistics` 环境变量查看动态链接的详细耗时：

```
# 测量启动时间  
time ./my_ros_node  
  
# 查看动态链接统计  
LD_DEBUG=statistics ./my_ros_node
```

除了动态链接，ROS 节点启动还涉及其他耗时操作：连接到 ROS Master、初始化发布者和订阅者、建立网络连接等。对于需要快速启动的实时系统，可以考虑以下优化策略：

减少不必要的依赖是最直接的方法。检查你的程序是否链接了实际并未使用的库，这在大型项目中很常见。每个额外的共享库都会增加加载时间。

使用预链接（prelink）工具可以提前完成部分链接工作，减少启动时的动态链接开销。不过这需要系统级的配置，在 RoboMaster 比赛环境中可能不太适用。

对于嵌入式系统或对启动时间极度敏感的场景，可以考虑静态链接。虽然可执行文件会变大，但省去了动态链接的开销，启动会快得多。

另一个思路是将不需要立即执行的初始化推迟到后台进行。例如，某些模块可以在程序启动后的空闲时间再初始化，而不是阻塞在 `main` 函数的开头。

理解从源码到执行的完整过程，能帮助你更好地组织代码、排查链接问题、优化程序性能。当你下次遇到“undefined reference”错误或程序启动缓慢时，希望这些知识能为你指明排查方向。

1.4.3. 进程：程序的运行实例

上一节我们了解了程序从源代码到可执行文件的构建过程。可执行文件只是静静地躺在磁盘上的一堆字节，它本身并不做任何事情。当操作系统将其加载到内存并开始执行时，一个“进程”便诞生了。进程是操作系统中最核心的概念之一，理解进程对于编写多任务程序、调试性能问题、理解 ROS 架构都至关重要。

1.4.3.1. 什么是进程

简单地说，进程是程序的运行实例。程序是静态的，是存储在磁盘上的指令和数据的集合；进程是动态的，是程序在内存中的执行过程。同一个程序可以同时运行多次，产生多个独立的进程。例如，你可以同时打开多个终端窗口，每个窗口运行一个 bash 进程，它们都来自同一个 `/bin/bash` 程序，但各自独立、互不干扰。

进程不仅仅是内存中的代码和数据，它还包含了执行所需的全部上下文信息：CPU 寄存器的当前值、程序计数器指向的位置、打开的文件列表、网络连接、当前工作目录、环境变量等。这些信息共同定义了进程在某一时刻的完整状态。当操作系统需要暂停一个进程去执行另一个进程时，必须保存这些信息；当这个进程恢复执行时，再将这些信息恢复，进程才能从中断的地方继续运行，仿佛什么都没有发生过。

每个进程都有一个唯一的标识符，称为进程 ID (PID)。PID 是一个正整数，在进程的生命周期内保持不变。操作系统通过 PID 来管理和区分不同的进程。在 Linux 中，有几个特殊的 PID 值值得注意：PID 0 是调度进程 (swapper)，负责进程调度；PID 1 是 init 进程（现代系统中通常是 systemd），是所有用户进程的祖先；PID 2 是内核线程的父进程。

进程之间存在父子关系。除了 init 进程外，每个进程都由另一个进程创建，创建者称为父进程，被创建的称为子进程。当你在终端中运行 `./my_robot` 时，bash 进程创建了一个子进程来执行你的程序。使用 `pstree` 命令可以查看系统中进程的树状结构：

```
$ pstree -p
systemd(1)── ModemManager(865)
              └── NetworkManager(802)
                ├── gdm(1205) ── gdm-session-wor(1245) ── gnome-session-b(1403) ── ...
                └── sshd(922) ── sshd(12045) ── bash(12047) ── my_robot(12089)
...
...
```

1.4.3.2. 进程的状态

进程在其生命周期中会经历不同的状态。虽然不同的教科书对状态的划分略有差异，但最基本的状态模型包含以下几种。

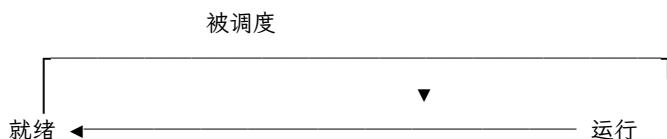
创建状态是进程刚被创建时的短暂状态。操作系统正在为进程分配资源、初始化进程控制块，但进程还没有准备好运行。

就绪状态表示进程已经准备好运行，正在等待 CPU 时间。就绪状态的进程拥有运行所需的一切资源，唯独缺少 CPU。在任何时刻，可能有多个进程处于就绪状态，它们在就绪队列中排队，等待调度器分配 CPU。

运行状态表示进程正在 CPU 上执行指令。在单核系统中，任何时刻只有一个进程处于运行状态；在多核系统中，最多有 N 个进程同时运行（N 是 CPU 核心数）。进程不会一直运行下去——当它用完分配的时间片，或者需要等待某个事件时，调度器会将 CPU 分配给其他进程。

阻塞状态（也称等待状态）表示进程正在等待某个事件的发生，无法继续执行。这个事件可能是等待 I/O 操作完成（如读取磁盘文件）、等待用户输入、等待网络数据到达、等待信号量或互斥锁、等待子进程结束等。处于阻塞状态的进程不会竞争 CPU，即使 CPU 空闲也不会分配给它。当等待的事件发生后，进程会转入就绪状态，重新排队等待 CPU。

终止状态表示进程已经结束执行。进程可能是正常退出（`main` 函数返回或调用 `exit`），也可能是异常终止（如收到致命信号、访问非法内存）。进程终止后，它占用的大部分资源会被操作系统回收，但进程的退出状态会保留，直到父进程读取。如果父进程不读取子进程的退出状态，子进程就会变成“僵尸进程”（zombie），在进程表中占用一个位置。





在 RoboMaster 开发中，理解进程状态有助于分析性能问题。如果你的机器人程序响应迟缓，可能是因为进程经常处于阻塞状态，等待 I/O 或锁；如果 CPU 占用率很高但程序仍然卡顿，可能是计算过于密集，或者有太多就绪进程在竞争 CPU。

1.4.3.3. 进程控制块

操作系统如何管理如此多的进程信息呢？答案是进程控制块（Process Control Block, PCB）。PCB 是操作系统为每个进程维护的数据结构，包含了进程的所有管理信息。

PCB 中最基本的信息包括进程标识（PID、父进程 PID）、进程状态、以及程序计数器（下一条要执行的指令地址）。除此之外，PCB 还记录 CPU 寄存器的值，这些值在进程被切换出去时保存，在进程恢复执行时恢复。内存管理信息也是 PCB 的重要组成部分，包括页表基址、内存限制、段表等，这些信息决定了进程能访问哪些内存区域。

PCB 还包含 I/O 状态信息：进程打开了哪些文件、每个文件的读写位置、分配了哪些 I/O 设备。记账信息记录了进程使用 CPU 的时间、实际运行时间、时间限制等，用于系统监控和计费。调度信息包括进程优先级、调度队列指针、所属调度类等，调度器根据这些信息决定下一个运行哪个进程。

在 Linux 中，内核使用 `task_struct` 结构体表示进程（在 Linux 中更准确的术语是“任务”），这是一个非常庞大的结构，包含数百个字段。你可以在内核源码的 `include/linux/sched.h` 中找到它的定义。

当发生进程切换时，操作系统执行上下文切换（context switch）：保存当前进程的状态到它的 PCB，从下一个进程的 PCB 恢复状态。上下文切换是有开销的——需要保存和恢复寄存器、刷新 TLB（页表缓存）、可能还要刷新 CPU 缓存。虽然现代 CPU 的上下文切换只需要几微秒，但在高频切换的情况下，这个开销会累积成可观的性能损失。这就是为什么在实时系统中，我们要尽量减少不必要的进程切换。

1.4.3.4. 进程调度

在多任务操作系统中，通常有多个进程同时处于就绪状态，但 CPU 资源是有限的。调度器（scheduler）负责决定哪个进程获得 CPU 时间、获得多长时间。调度策略直接影响系统的响应性、吞吐量和公平性。

最简单的调度算法是先来先服务（First-Come, First-Served, FCFS）：按照进程到达就绪队列的顺序分配 CPU。这种方法简单公平，但一个长时间运行的进程会阻塞后面的所有进程，导致平均等待时间很长。

时间片轮转（Round-Robin, RR）是更常用的方法：每个进程获得一个固定长度的时间片（通常是几毫秒到几十毫秒），用完后无论是否执行完毕，都必须让出 CPU 给下一个进程。这样可以保证所有进程都能得到响应，但时间片太短会导致频繁切换带来的开销，太长则会影响响应性。

优先级调度允许不同进程拥有不同的优先级，高优先级进程优先获得 CPU。这对于实时系统很重要——紧急任务需要比普通任务更快得到响应。但纯粹的优先级调度可能导致低优先级进

程“饿死”（永远得不到 CPU），因此实际系统通常会动态调整优先级，或者将优先级调度与时间片轮转结合。

Linux 使用的是完全公平调度器（Completely Fair Scheduler, CFS），它试图给每个进程公平的 CPU 时间份额。CFS 使用红黑树组织就绪进程，根据每个进程的“虚拟运行时间”来决定调度顺序——运行时间少的进程优先获得 CPU。对于需要实时性的进程，Linux 还提供 SCHED_FIFO 和 SCHED_RR 策略，这些进程的优先级高于普通进程。

在 RoboMaster 的 Linux 系统上，你可以使用 nice 命令调整进程的优先级（nice 值越低，优先级越高）：

```
# 以更高优先级运行程序（需要 root 权限）
sudo nice -n -10 ./my_robot
```

```
# 查看进程的 nice 值
ps -o pid,ni,comm
```

对于需要硬实时保证的场景，普通 Linux 可能不够用，需要使用 RT-Preempt 补丁或实时操作系统。RoboMaster 机器人的嵌入式端通常使用 FreeRTOS 等实时操作系统，保证控制循环的精确时序。

1.4.3.5. 查看和管理进程

Linux 提供了丰富的工具来查看和管理系统中的进程。掌握这些工具对于调试和性能分析非常有帮助。

ps 是最基本的进程查看命令。不带参数时，它只显示当前终端的进程。加上参数可以查看更多信息：

```
# 查看所有进程的详细信息
ps aux

# 输出说明：
# USER: 进程所有者
# PID: 进程 ID
# %CPU: CPU 使用率
# %MEM: 内存使用率
# VSZ: 虚拟内存大小 (KB)
# RSS: 实际使用的物理内存 (KB)
# TTY: 关联的终端
# STAT: 进程状态 (R=运行, S=睡眠, D=不可中断睡眠, Z=僵尸, T=停止)
# START: 启动时间
# TIME: 累计 CPU 时间
# COMMAND: 命令行

# 查看特定进程
ps aux | grep my_robot

# 查看进程树
ps axjf
```

top 提供实时的进程监控界面，每隔几秒刷新一次。它显示系统整体的 CPU、内存使用情况，以及各个进程的资源占用。在 top 界面中，你可以按 P 键按 CPU 排序，按 M 键按内存排序，按 k 键杀死进程，按 q 键退出。

`htop` 是 `top` 的增强版，提供更友好的界面、彩色显示、鼠标支持，还可以直接上下选择进程进行操作。如果系统没有安装，可以用 `sudo apt install htop` 安装。

```
# 实时监控所有进程  
top
```

```
# 更友好的界面  
htop
```

```
# 只监控特定进程  
top -p $(pgrep my_robot)
```

`/proc` 文件系统是一个虚拟文件系统，提供了内核和进程信息的接口。每个进程在 `/proc` 下都有一个以 PID 命名的目录，包含该进程的各种信息：

```
# 查看进程 12345 的各种信息  
ls /proc/12345/
```

```
# 常用文件：  
cat /proc/12345/status      # 进程状态  
cat /proc/12345/cmdline     # 命令行参数  
cat /proc/12345/environ     # 环境变量  
cat /proc/12345/fd/          # 打开的文件描述符  
cat /proc/12345/maps         # 内存映射
```

控制进程的基本命令包括：

```
# 终止进程（发送 SIGTERM，允许进程清理）  
kill 12345
```

```
# 强制终止（发送 SIGKILL，立即杀死，不可捕获）  
kill -9 12345
```

```
# 按名称杀死进程  
pkill my_robot  
killall my_robot
```

```
# 暂停进程  
kill -STOP 12345
```

```
# 恢复进程  
kill -CONT 12345
```

在调试 RoboMaster 程序时，这些工具非常有用。如果机器人行为异常，可以用 `htop` 查看 CPU 和内存使用情况；如果程序卡死，可以用 `kill -3` 发送 SIGQUIT 信号让程序打印堆栈信息；如果需要分析程序打开了哪些文件，可以查看 `/proc/PID/fd/` 目录。

1.4.3.6. 进程的创建与终止

在 Linux 中，创建新进程使用 `fork()` 系统调用。`fork()` 创建一个几乎完全相同的子进程副本，子进程继承父进程的代码、数据、打开的文件、环境变量等。`fork()` 的巧妙之处在于它“调用一次，返回两次”——在父进程中返回子进程的 PID，在子进程中返回 0，程序可以据此判断自己是父进程还是子进程。

```
#include <unistd.h>  
#include <iostream>
```

```

int main() {
    std::cout << "Before fork, PID = " << getpid() << std::endl;

    pid_t pid = fork();

    if (pid < 0) {
        // fork 失败
        std::cerr << "Fork failed!" << std::endl;
        return 1;
    } else if (pid == 0) {
        // 子进程
        std::cout << "Child process, PID = " << getpid()
            << ", Parent PID = " << getppid() << std::endl;
    } else {
        // 父进程
        std::cout << "Parent process, PID = " << getpid()
            << ", Child PID = " << pid << std::endl;
    }

    return 0;
}

```

fork() 之后，子进程通常会调用 exec() 族函数来加载并执行一个新程序，替换掉从父进程继承的代码。这就是 shell 运行命令的基本原理：shell 进程 fork 出子进程，子进程 exec 要运行的程序，父进程等待子进程结束。

```

#include <unistd.h>
#include <sys/wait.h>
#include <iostream>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // 子进程：执行 ls 命令
        execvp("ls", {"ls", "-la", nullptr});
        // 如果 exec 成功，下面的代码不会执行
        std::cerr << "Exec failed!" << std::endl;
        return 1;
    } else {
        // 父进程：等待子进程结束
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status)) {
            std::cout << "Child exited with code " << WEXITSTATUS(status) <<
std::endl;
        }
    }

    return 0;
}

```

现代 Linux 提供了 clone() 系统调用，它比 fork() 更灵活，可以精细控制子进程与父进程共享哪些资源。线程实际上就是通过 clone() 创建的，共享地址空间但拥有独立的栈。

进程终止时，操作系统会执行以下清理工作：关闭所有打开的文件描述符、释放分配的内存、释放其他资源（如信号量、共享内存）。子进程终止后会变成僵尸状态，直到父进程调用 `wait()` 或 `waitpid()` 读取其退出状态。如果父进程比子进程先终止，子进程会被 `init` 进程“收养”，由 `init` 负责回收其资源。

1.4.3.7. 进程间通信

进程是相互隔离的——每个进程有自己独立的地址空间，一个进程不能直接访问另一个进程的内存。这种隔离是操作系统提供的重要保护机制，防止进程之间相互干扰。但有时候进程之间需要交换数据、协调工作，这就需要进程间通信（Inter-Process Communication, IPC）机制。

管道（pipe）是最简单的 IPC 机制。管道是一个单向的字节流通道，一端写入，另一端读出。当你在 shell 中使用 `|` 符号时，就是在使用管道：

```
# ls 的输出通过管道传给 grep
ls -la | grep ".cpp"
```

匿名管道只能用于有亲缘关系的进程（如父子进程）之间通信。命名管道（FIFO）则可以用于任意进程之间通信，它在文件系统中有一个名字，进程通过打开这个文件来读写管道。

```
#include <unistd.h>
#include <iostream>

int main() {
    int pipefd[2]; // pipefd[0] 读端, pipefd[1] 写端
    pipe(pipefd);

    pid_t pid = fork();

    if (pid == 0) {
        // 子进程：从管道读取
        close(pipefd[1]); // 关闭写端
        char buffer[256];
        read(pipefd[0], buffer, sizeof(buffer));
        std::cout << "Child received: " << buffer << std::endl;
        close(pipefd[0]);
    } else {
        // 父进程：向管道写入
        close(pipefd[0]); // 关闭读端
        const char* message = "Hello from parent!";
        write(pipefd[1], message, strlen(message) + 1);
        close(pipefd[1]);
        wait(nullptr);
    }

    return 0;
}
```

共享内存是最快的 IPC 方式，因为它允许多个进程直接访问同一块物理内存，数据不需要在内核和用户空间之间复制。使用共享内存时，多个进程需要协调对共享区域的访问，通常结合信号量或互斥锁使用。

```
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
```

```

#include <cstring>

int main() {
    // 创建共享内存对象
    int fd = shm_open("/my_shm", O_CREAT | O_RDWR, 0666);
    ftruncate(fd, 4096); // 设置大小

    // 映射到进程地址空间
    void* ptr = mmap(nullptr, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    // 现在可以像普通内存一样读写 ptr
    strcpy((char*)ptr, "Hello, shared memory!");

    // 其他进程可以用相同的名称打开并映射这块内存

    // 清理
    munmap(ptr, 4096);
    shm_unlink("/my_shm");

    return 0;
}

```

消息队列提供了一种结构化的消息传递方式。发送者将消息放入队列，接收者从队列取出消息。与管道不同，消息队列中的消息有明确的边界，接收者一次接收一条完整的消息。消息还可以有类型，接收者可以选择接收特定类型的消息。

信号 (signal) 是一种异步通知机制，用于通知进程发生了某个事件。当进程收到信号时，会中断当前执行的操作，转而执行信号处理函数。常见的信号包括：

SIGINT (2) - 终端中断，通常由 Ctrl+C 触发

SIGTERM (15) - 终止请求，可以被捕获和处理

SIGKILL (9) - 强制终止，不能被捕获或忽略

SIGSEGV (11) - 段错误，访问非法内存

SIGCHLD (17) - 子进程状态改变

SIGUSR1/2 - 用户自定义信号

```
#include <signal.h>
```

```
#include <iostream>
```

```
volatile sig_atomic_t running = 1;
```

```
void signalHandler(int signum) {
    std::cout << "Received signal " << signum << std::endl;
    running = 0;
}
```

```
int main() {
    // 注册信号处理函数
    signal(SIGINT, signalHandler);
    signal(SIGTERM, signalHandler);
```

```
    std::cout << "Running... Press Ctrl+C to stop." << std::endl;
```

```
    while (running) {
        // 主循环
        sleep(1);
    }
}
```

```

    std::cout << "Shutting down gracefully." << std::endl;
    return 0;
}

```

套接字 (socket) 最初是为网络通信设计的，但 Unix 域套接字也可以用于同一机器上的进程间通信。套接字提供了灵活的双向通信能力，支持流式（类似 TCP）和数据报（类似 UDP）两种模式。ROS 的节点间通信底层就大量使用套接字。

1.4.3.8. ROS 节点与进程

了解了进程的概念，我们可以更好地理解 ROS 的架构。在 ROS 中，每个节点 (node) 本质上就是一个进程。当你运行 `rosrun my_package my_node` 时，操作系统创建一个新进程来执行你的节点程序。

ROS Master 是一个特殊的节点，负责节点的注册和发现。当节点启动时，它首先连接到 Master，注册自己的名称以及它发布和订阅的话题。Master 并不参与实际的消息传递，它只是一个“介绍人”——当订阅者想要接收某个话题时，Master 告诉它发布者的地址，然后订阅者直接与发布者建立连接。

节点之间的话题通信使用的是前面提到的 IPC 机制。对于同一机器上的节点，ROS 1 默认使用 TCP 套接字 (TCPROS)，也可以配置使用 UDP 或共享内存。ROS 2 使用 DDS (Data Distribution Service) 作为通信中间件，它会根据节点是否在同一机器上自动选择最优的通信方式——同机器可能使用共享内存，跨机器使用网络。

```

# 查看 ROS 节点对应的进程
rosnode list
rosnode info /my_node

# 使用 ps 查看 ROS 节点进程
ps aux | grep ros

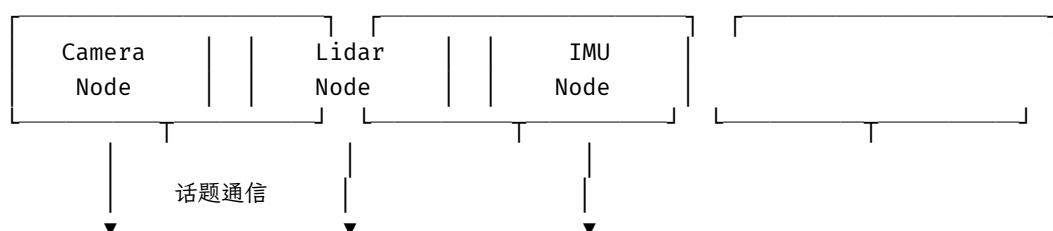
# 查看节点进程打开的网络连接
lsof -i -p $(pgrep my_node)

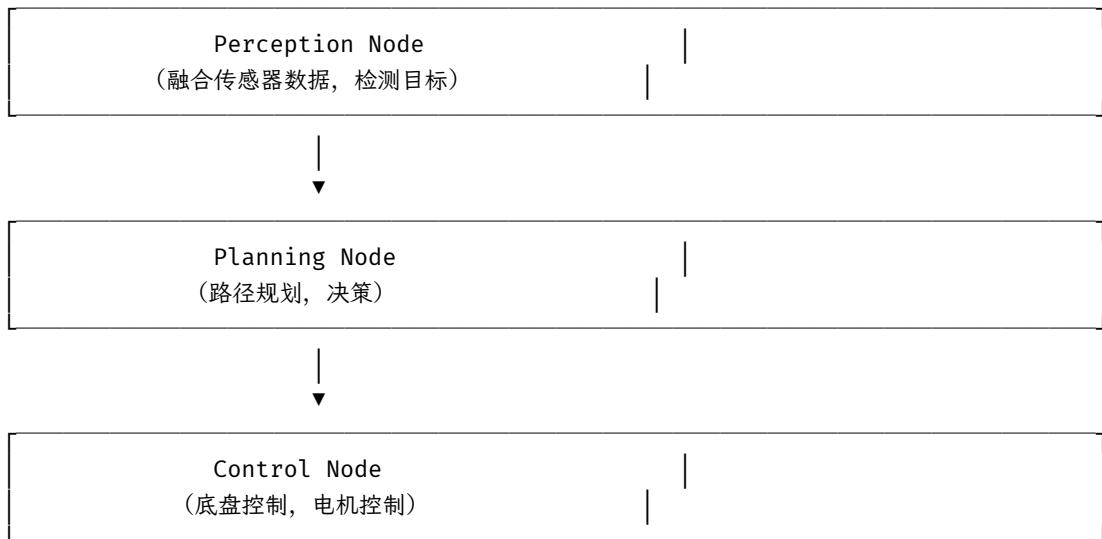
```

理解 ROS 节点是进程，有助于解释一些常见问题。为什么节点之间不能直接共享变量？因为它们是不同的进程，有独立的地址空间。为什么一个节点崩溃不会影响其他节点？因为进程是相互隔离的。为什么启动大量节点会消耗很多资源？因为每个进程都有自己的内存开销。

在设计 ROS 系统时，需要权衡节点的粒度。把功能拆分到多个节点可以提高模块化和容错性，但节点间通信会带来延迟和开销。对于实时性要求高的功能（如控制循环），通常放在同一个节点内用多线程实现；对于相对独立的功能（如感知、规划、用户界面），可以拆分为不同的节点。

ROS 系统的典型架构：





进程是操作系统管理程序执行的基本单位。理解进程的概念、状态、调度以及进程间通信机制，是深入理解计算机系统的基础。在 RoboMaster 开发中，无论是调试多节点系统、分析性能问题，还是设计系统架构，进程知识都会派上用场。下一节我们将讨论线程——一种更轻量级的执行单元，它在进程内部提供并发执行的能力。

1.4.4. 线程：轻量级执行单元

上一节我们了解了进程的概念。进程提供了程序执行的独立环境，但这种隔离性也带来了代价：创建进程开销大，进程间通信复杂，进程切换成本高。当一个程序内部需要同时执行多个任务时——比如机器人程序需要同时采集图像、处理数据、执行控制——如果为每个任务创建一个进程，不仅资源消耗大，任务间的数据共享也很麻烦。线程（thread）应运而生，它是一种更轻量级的执行单元，在保持并发能力的同时，大大降低了开销。

1.4.4.1. 从进程到线程

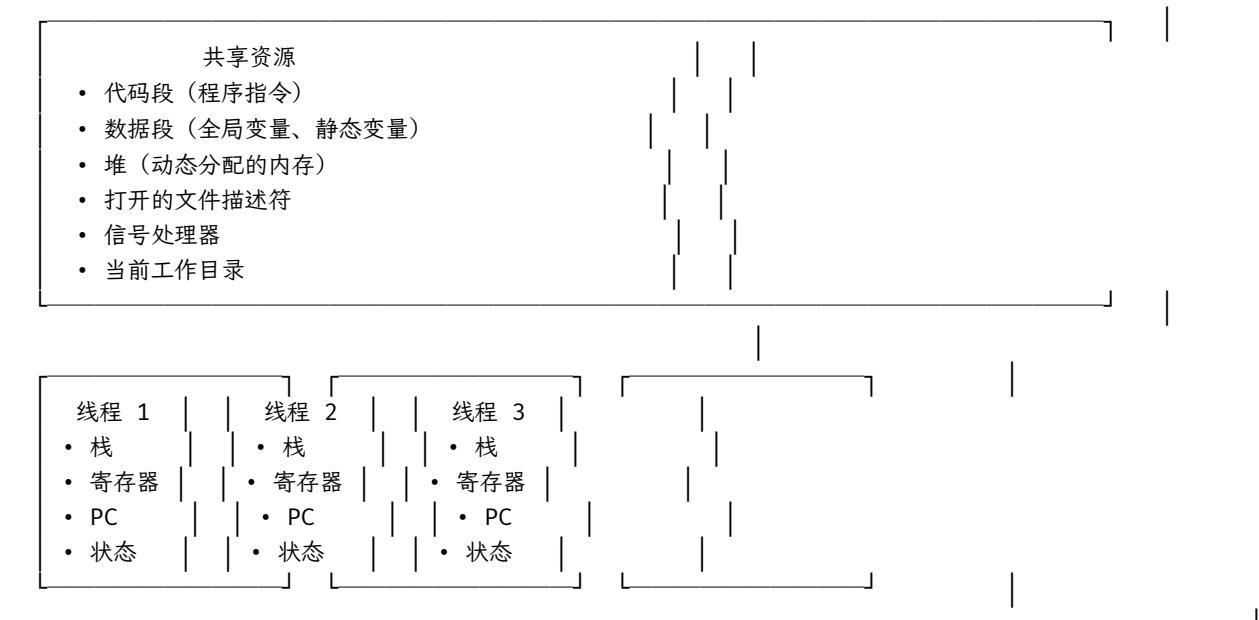
在早期的操作系统中，进程是 CPU 调度的基本单位。一个进程在任何时刻只能执行一个任务，即使计算机有多个 CPU 核心，单个进程也只能使用其中一个。如果程序需要并发执行多个任务，就必须创建多个进程，但进程之间是隔离的，数据共享需要通过 IPC 机制，既复杂又有开销。

线程的出现改变了这一局面。线程是进程内部的执行流，一个进程可以包含多个线程，这些线程共享进程的地址空间和资源，但各自拥有独立的执行上下文。你可以把进程想象成一个工厂，线程则是工厂里的工人。工人们共享工厂的设备、原材料和产品仓库（共享地址空间），但每个工人有自己的工作台和工具（独立的栈和寄存器），可以独立完成各自的任务。

线程与进程的关键区别在于资源共享的程度。进程之间是完全隔离的，每个进程有独立的地址空间，一个进程不能直接访问另一个进程的内存。而同一进程内的线程共享代码段、数据段、堆内存、打开的文件、信号处理器等资源，唯独栈是每个线程私有的。这种共享带来了两个重要的好处：一是线程间的数据交换极其简单，直接读写共享变量即可；二是线程的创建和切换开销远小于进程，因为不需要复制整个地址空间。

进程与线程的资源关系：





每个线程都有自己的栈空间，用于存储局部变量和函数调用信息。这意味着线程函数中的局部变量是线程私有的，不会与其他线程冲突。但全局变量、静态变量以及堆上分配的内存是所有线程共享的，这是多线程编程中需要特别注意的地方——共享意味着可能产生冲突。

1.4.4.2. 为什么需要多线程

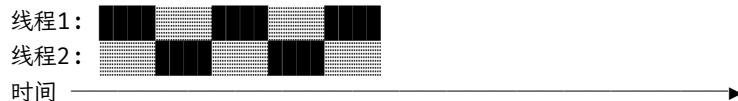
多线程编程的核心目标是并发 (concurrency) 和并行 (parallelism)。这两个概念经常被混淆，但它们有本质区别。

并发是指多个任务在同一时间段内交替执行，宏观上看它们是“同时”进行的，但在任一微观时刻，可能只有一个任务在执行。单核 CPU 上的多线程就是并发：操作系统快速地在线程之间切换，每个线程执行一小段时间，给人以同时执行的错觉。并发的主要目的是提高程序的响应性和资源利用率——当一个线程在等待 I/O 时，CPU 可以切换到另一个线程继续执行，而不是空等。

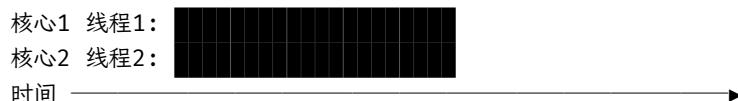
并行是指多个任务真正地同时执行，这需要多核 CPU 或多个处理器的支持。在 4 核 CPU 上，最多可以有 4 个线程真正同时执行。并行的主要目的是提高计算吞吐量——把一个大任务分解成多个子任务，让多个核心同时处理，缩短总执行时间。

并发 vs 并行：

并发（单核）：



并行（多核）：



在 RoboMaster 机器人开发中，多线程几乎是必需的。考虑一个典型的视觉瞄准系统：相机以 60 FPS 的速度采集图像，图像处理算法需要检测目标，控制算法需要计算瞄准角度并发送给云台。如果这些任务顺序执行，假设图像处理需要 20ms，那么从看到目标到做出响应至少需要 20ms 以上，而且在处理图像的这段时间里，新的图像无法采集，控制输出也会停滞。

使用多线程，这些任务可以并发甚至并行执行：一个线程专门负责图像采集，一个线程负责图像处理，一个线程负责控制输出。图像采集线程不断地将新图像放入缓冲区，图像处理线程从缓冲区取出图像进行处理，控制线程以固定的频率输出控制指令。三个任务互不阻塞，系统的响应性大大提高。

```
// 多线程视觉系统的概念示意
class VisionSystem {
    std::queue<Frame> frameBuffer;
    std::mutex bufferMutex;
    Target latestTarget;
    std::atomic<bool> running{true};

    void captureThread() {
        while (running) {
            Frame frame = camera.capture();
            {
                std::lock_guard<std::mutex> lock(bufferMutex);
                frameBuffer.push(frame);
            }
        }
    }

    void processThread() {
        while (running) {
            Frame frame;
            {
                std::lock_guard<std::mutex> lock(bufferMutex);
                if (!frameBuffer.empty()) {
                    frame = frameBuffer.front();
                    frameBuffer.pop();
                }
            }
            if (frame.valid()) {
                latestTarget = detector.detect(frame);
            }
        }
    }

    void controlThread() {
        while (running) {
            auto angles = calculateAim(latestTarget);
            gimbal.setAngles(angles);
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
        }
    }
};
```

多线程还有一个重要的应用场景是保持程序的响应性。在图形界面程序中，如果在主线程中执行耗时操作，界面就会“卡住”，无法响应用户输入。正确的做法是将耗时操作放到后台线程执行，主线程只负责处理用户界面。在机器人系统中道理相同：如果主控制循环被某个耗时操作阻塞，机器人就会失去响应，这在比赛中是致命的。

1.4.4.3. 用户线程与内核线程

线程的实现可以在用户空间，也可以在内核空间，或者两者结合。这三种模型各有优缺点。

用户级线程完全在用户空间实现，操作系统内核对线程的存在一无所知，它只看到一个普通的进程。线程的创建、调度、同步都由用户空间的线程库完成。用户级线程的优点是创建和切换速度极快（不需要陷入内核），可以在不支持线程的操作系统上运行，而且调度策略可以由应用程序自定义。但缺点也很明显：当一个用户级线程执行阻塞的系统调用（如读文件）时，整个进程都会被阻塞，因为内核不知道进程中还有其他可运行的线程；另外，用户级线程无法利用多核并行，因为内核只会把整个进程调度到一个核心上。

内核级线程由操作系统内核直接管理，内核知道每个线程的存在，并负责它们的调度。当一个线程阻塞时，内核可以调度同一进程的其他线程继续执行。内核级线程可以真正地并行运行在多个CPU核心上。缺点是线程的创建和切换需要陷入内核，开销比用户级线程大。Linux从2.6版本开始使用NPTL（Native POSIX Thread Library），它实现了高效的内核级线程，是目前Linux上的标准线程实现。

现代系统通常采用混合模型：用户空间的线程库与内核级线程配合使用。例如，Go语言的goroutine就是用户级的轻量级线程，由Go运行时调度到少量的内核线程上执行，兼顾了轻量级和并行能力。

在Linux中，线程实际上通过与进程相同的task_struct结构表示的，只是创建时通过clone()系统调用指定了共享地址空间等标志。这就是为什么有时说“Linux不区分进程和线程”——从内核调度器的角度看，它们都是“任务”，只是共享资源的程度不同。

```
# 查看进程的线程  
ps -T -p <PID>  
  
# 使用 top 查看线程（按 H 键切换到线程视图）  
top -H -p <PID>  
  
# 在 /proc 中查看线程  
ls /proc/<PID>/task/
```

1.4.4.4. 线程的调度与切换

线程调度与进程调度的原理相同，只是调度的粒度更细。在支持内核级线程的系统中，调度器直接调度线程而非进程。就绪线程排队等待CPU，调度器根据优先级、时间片等策略选择下一个执行的线程。

线程上下文切换比进程切换轻量，但仍有开销。切换时需要保存和恢复的上下文包括：程序计数器(PC)、栈指针(SP)、通用寄存器、浮点寄存器、状态寄存器等。由于同一进程的线程共享地址空间，线程切换不需要切换页表，也不需要刷新TLB(页表缓存)，这是线程切换比进程切换快的主要原因。

尽管如此，频繁的线程切换仍然会带来可观的开销。每次切换都需要陷入内核、保存恢复寄存器、可能刷新CPU缓存（因为新线程访问的数据可能不在缓存中）。如果线程数量远多于CPU核心数，线程之间竞争激烈，大量时间会花在切换上而非实际计算。

在RoboMaster开发中，线程数量的选择需要谨慎。一个常见的误区是“线程越多越好”——实际上，对于CPU密集型任务，线程数量接近CPU核心数是最优的；创建过多线程反而会因为频

繁切换而降低性能。对于 I/O 密集型任务（如网络通信、文件读写），可以使用更多线程，因为线程大部分时间在等待，切换开销相对较小。

```
// 获取 CPU 核心数，作为线程数的参考
unsigned int numCores = std::thread::hardware_concurrency();
std::cout << "CPU 核心数: " << numCores << std::endl;
```

Linux 允许设置线程的调度策略和优先级。普通线程使用 SCHED_OTHER 策略，由 CFS 调度器公平调度。对于实时性要求高的线程，可以使用 SCHED_FIFO 或 SCHED_RR 策略，这些线程的优先级高于普通线程，会被优先调度。不过，使用实时调度策略需要 root 权限，而且不当使用可能导致系统失去响应（高优先级线程独占 CPU）。

```
#include <pthread.h>
#include <sched.h>

void setRealtimePriority() {
    struct sched_param param;
    param.sched_priority = 80; // 实时优先级 1-99

    if (pthread_setschedparam(pthread_self(), SCHED_FIFO, &param) != 0) {
        std::cerr << "设置实时优先级失败（需要 root 权限）" << std::endl;
    }
}
```

1.4.4.5. 多线程的挑战

多线程编程的复杂性不在于创建和管理线程，而在于协调线程之间的交互。当多个线程访问共享数据时，如果没有适当的同步机制，就会产生各种难以调试的问题。

竞态条件（race condition）是最常见的多线程 bug。当程序的行为取决于多个线程执行的相对顺序时，就存在竞态条件。考虑一个简单的例子：两个线程同时对一个计数器加一。

```
int counter = 0;

void increment() {
    for (int i = 0; i < 100000; i++) {
        counter++; // 这不是原子操作!
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();

    // 期望 200000，实际结果可能小于这个值
    std::cout << "Counter: " << counter << std::endl;
    return 0;
}
```

counter++ 看似简单，实际上包含三个步骤：读取 counter 的值到寄存器、在寄存器中加一、将结果写回 counter。如果两个线程的这三个步骤交错执行，就可能丢失更新。假设 counter 当前值为 100，线程 1 读取到 100，还没来得及写回，线程 2 也读取到 100，然后两个线程都写回 101，本应增加 2 却只增加了 1。

数据竞争 (data race) 是竞态条件的一种特殊形式：当两个线程同时访问同一内存位置，且至少一个是写操作，且没有同步机制时，就发生了数据竞争。C++ 标准规定数据竞争是未定义行为，编译器和 CPU 可以假设程序中不存在数据竞争，并据此进行优化，这可能导致出乎意料的结果。

死锁 (deadlock) 是另一个经典问题。当两个或多个线程相互等待对方持有的资源时，就形成了死锁，所有相关线程都无法继续执行。最简单的死锁场景是两个线程、两把锁：

```
std::mutex mutexA, mutexB;

void thread1() {
    std::lock_guard<std::mutex> lockA(mutexA);
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::lock_guard<std::mutex> lockB(mutexB); // 等待 mutexB
    // ...
}

void thread2() {
    std::lock_guard<std::mutex> lockB(mutexB);
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::lock_guard<std::mutex> lockA(mutexA); // 等待 mutexA, 死锁!
    // ...
}
```

线程 1 持有 mutexA，等待 mutexB；线程 2 持有 mutexB，等待 mutexA。两个线程都在等待对方释放资源，形成循环等待，程序永久卡死。死锁的四个必要条件是：互斥（资源不能共享）、持有并等待（持有资源的同时等待其他资源）、非抢占（不能强制释放他人的资源）、循环等待（形成等待环路）。预防死锁的常见方法是打破循环等待——规定所有线程必须按相同的顺序获取锁。

活锁 (livelock) 与死锁类似，但线程不是静止等待，而是不断地改变状态却无法取得进展。想象两个人在走廊相遇，都想给对方让路，结果两人同时向左、又同时向右，不断重复却无法通过。

优先级反转 (priority inversion) 是实时系统中的问题。当高优先级线程等待低优先级线程持有的锁时，高优先级线程被阻塞。如果此时有中等优先级的线程运行，低优先级线程得不到执行机会，锁无法释放，高优先级线程就被无限期阻塞。这个问题曾导致火星探路者号 (Mars Pathfinder) 探测器频繁重启。解决方案包括优先级继承（临时提升持有锁的低优先级线程的优先级）和优先级天花板（锁的优先级等于可能使用它的最高优先级线程）。

内存可见性问题更加隐蔽。现代 CPU 使用多级缓存和乱序执行来提高性能，一个线程对内存的修改可能不会立即对其他线程可见。在没有同步机制的情况下，线程可能读到“过时”的数据。C++ 的内存模型定义了多线程环境下内存操作的可见性保证，`std::atomic` 和同步原语提供了必要的内存屏障。

```
bool ready = false;
int data = 0;

void producer() {
    data = 42;
    ready = true; // 可能被重排到 data = 42 之前!
}

void consumer() {
    while (!ready); // 可能永远看不到 ready == true
```

```
    std::cout << data << std::endl; // 可能输出 0
}
```

这些问题的共同特点是不确定性和难以复现。程序可能运行一千次都正常，但在比赛关键时刻出现问题。多线程 bug 往往与时序相关，调试时由于执行速度变慢，bug 反而消失了。这就是为什么多线程编程需要系统性的方法，而不能依赖“测试通过”。

1.4.4.6. 同步机制概览

为了解决上述问题，操作系统和编程语言提供了各种同步机制。这里简要介绍主要的同步原语，后续的 C++ 多线程章节会详细讲解它们的使用。

互斥锁（mutex）是最基本的同步机制。它确保同一时刻只有一个线程可以进入临界区（访问共享资源的代码段）。线程进入临界区前必须获取锁，离开时释放锁。如果锁已被其他线程持有，试图获取的线程会被阻塞。

```
std::mutex mtx;
int sharedData = 0;

void safeIncrement() {
    mtx.lock();      // 获取锁
    sharedData++;   // 临界区
    mtx.unlock();   // 释放锁
}
```

读写锁（read-write lock）优化了“读多写少”的场景。它允许多个线程同时读取共享数据，但写入时必须独占。这提高了读取的并发性。

条件变量（condition variable）允许线程等待某个条件成立。它通常与互斥锁配合使用，线程可以在条件不满足时进入等待状态并释放锁，当其他线程改变条件并发出通知时被唤醒。条件变量是实现生产者-消费者模式的基础。

信号量（semaphore）是一个计数器，可以用来限制同时访问某个资源的线程数量。互斥锁可以看作是初始值为 1 的信号量。

原子操作（atomic operation）是不可分割的操作，执行过程中不会被其他线程打断。对于简单的数值操作，使用原子类型比使用互斥锁更高效。

```
std::atomic<int> atomicCounter(0);

void safeAtomicIncrement() {
    atomicCounter++; // 原子操作，无需锁
}
```

选择合适的同步机制需要权衡正确性、性能和复杂性。过度同步会降低并发性能，同步不足会导致正确性问题。一个好的原则是：尽量减少共享数据，必须共享时才加锁，加锁时尽量缩小临界区范围。

1.4.4.7. 从系统到语言

理解了线程的底层概念后，我们就为学习 C++ 多线程编程打下了基础。C++11 引入了标准的线程支持库，包括 `std::thread`、`std::mutex`、`std::condition_variable`、`std::atomic` 等。这些组件是对操作系统原生线程 API（如 POSIX 的 `pthread`）的跨平台封装，让你可以写出可移植的多线程代码。

```
#include <thread>
#include <mutex>
#include <iostream>

std::mutex coutMutex;

void printMessage(int id) {
    std::lock_guard<std::mutex> lock(coutMutex);
    std::cout << "Hello from thread " << id << std::endl;
}

int main() {
    std::thread t1(printMessage, 1);
    std::thread t2(printMessage, 2);

    t1.join();
    t2.join();

    return 0;
}
```

在后续的 C++ 多线程章节中，我们将详细学习如何使用这些组件：创建和管理线程、使用互斥锁保护共享数据、用条件变量实现线程间协调、用原子操作进行无锁编程。我们还会讨论 RAII 风格的锁管理（`lock_guard`、`unique_lock`）、如何避免常见的多线程陷阱，以及如何在 RoboMaster 项目中设计线程安全的系统。

线程是并发编程的基础，也是现代软件开发的必备技能。机器人系统天然需要同时处理多个任务，掌握多线程编程可以让你的程序充分利用多核处理器的能力，构建响应迅速、性能优异的机器人控制软件。但与此同时，多线程编程的复杂性也不容小觑——竞态条件、死锁等问题可能潜伏很久才暴露，一旦出现往往难以排查。下一节我们将讨论内存的层次结构，理解为什么有些程序比其他程序快得多，以及如何编写对缓存友好的代码。

1.4.5. 内存层次结构

1.4.6. 虚拟内存

1.4.7. 进程的内存布局

1.4.8. 文件系统

1.4.9. 网络通信基础

1.4.10. 并发与同步

1.4.11. 操作系统的角色

1.4.12. 性能分析与调试工具

1.5. C++ 语言基础

1.5.1. C++ 基本程序结构

本章节介绍 C++ 程序的基本组成形式，并说明源代码从编写到生成可执行文件的完整流程。内容包括程序结构、编译器 g++ 的使用方法以及构建系统 CMake 的基础用法。

一个最小的 C++ 程序通常由源文件 *.cpp 和可选的头文件 *.h 或 *.hpp 组成。程序的入口由 main() 函数定义，它是操作系统加载程序后第一个执行的函数。

下面是一个简单的 C++ 程序示例：

```
#include <iostream> // 引入输入输出库
using namespace std;
int main() {
    cout << "Hello, RoboMaster!" << endl; // 输出文本
    return 0; // 返回状态码
}
```

如果读者此前已经接触过 C 语言，可以发现 C++ 与 C 语言的相似之处，但必须说明的是：C++ 和 C 语言是两种不同的语言，C++ 既不是 C 语言的扩展，C 语言也不是 C++ 的简化。这一点将在后续的章节中越来越多地体现。对比 C++ 与 C 语言的 Hello RoboMaster 代码就能看出明显的区别，这是 C 语言的 Hello World 代码，能和上方代码输出相同的内容：

```
#include <stdio.h> // 引入标准输入输出库
int main() {
    printf("Hello, RoboMaster!\n"); // 输出文本
    return 0; // 返回状态码
}
```

了解了 C 语言与 C++ 的不同之后，现在让我们来解释一下这段程序：

- **#include <iostream>:**

C++ 提供了一组标准头文件 (header files)，这些头文件包含了常用的类型、函数和类，程序只需包含相应头文件即可直接使用这些工具。就是其中用于“输入/输出”操作的头文件：I/O (Input/Output) 表示输入输出，stream (流) 表示数据按序列化地在程序内部或程序与外部设备之间传输的抽象。

在本程序中，我们需要借助标准输出能力将文本打印到终端，`<iostream>`中正好提供了 `cout`、`cin`、`cerr` 等对象来完成这些工作。因此用 `#include <iostream>` 将相关声明引入到翻译单元中，编译器在预处理阶段会把头文件内容插入到源文件。

- **using namespace std;**

C++ 使用命名空间（namespace）来组织符号，避免不同库之间的名称冲突。标准库中的大部分符号（包括 `cout`、`endl`、`vector` 等）都被放在 `std` 命名空间内。

`using namespace std;` 这行告诉编译器在当前作用域中可以不写 `std::` 前缀直接访问 `std` 下的符号。它方便了书写，但在较大型工程或头文件中滥用会增加命名冲突的风险。因此更推荐的做法是在需要时显式使用 `std::`：

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

这样更清晰，也更安全。

- **int main():**

每个可独立运行的 C++ 程序都必须包含一个名为 `main()` 的函数：操作系统在启动程序时会调用它。`main` 的返回值类型通常为 `int`，代表程序的退出状态；返回 `0` 一般表示成功，非零值表示出现了错误或异常终止。

函数体由一对大括号 {} 包围，内部是语句块（statement block），包含程序逻辑。

- **注释：// 与 /* ... */**

注释用于向阅读代码的人说明意图，编译器会忽略注释内容。C++ 支持两种注释形式：
- // 单行注释：// 之后至行尾为注释。
- /* */ 多行注释：可跨行书写。

良好的注释能提高代码可读性，但注释内容应简洁、相关，避免重复说明代码本身能表达的事实。

- **cout << "Hello World" << endl;** —— 输出与“流”的概念

这一行展示了 C++ 标准库 I/O 的核心风格：流（stream）操作。`cout`（Console Output）是一个输出流对象，负责将数据发送到标准输出（通常是终端）。`<<` 是插入运算符（insertion operator），表示“把右侧的值插入到左侧的流中”，可把多个项连续插入同一个流。`"Hello World"` 是一个字符串字面量，将被写入 `cout`。`endl` 是一个操作符，表示输出换行并刷新流缓冲区（flush），常用于确保输出及时显示。流操作的链式写法使得多个输出项可以按顺序组合，例如：

```
cout << "Hello" << " " << "World" << endl;
```

可以把这一过程理解为：字符串和 `endl` 按顺序“流向”`cout`，最终由 `cout` 将它们输出到终端。这个“流动”的抽象比直接调用格式化函数更面向对象，也更易于扩展（例如重载 `operator<<` 来支持自定义类型）。需要注意的一点是：`endl` 会强制刷新缓冲区，这在频繁输出的小段落中可能影响性能。若只需要换行而不刷新的话，可以使用 '`\n`'。

- **return 0;**

`return 0;` 表示 `main` 正常结束并向操作系统返回成功状态。不同平台和约定会使用特定的非零值标识各种错误或异常退出原因，但通常情况下返回 `0` 即视为成功。

1.5.2. 编译一个 C++ 程序

在理解了程序结构之后，下一步需要了解：如何将源代码转换为计算机能够执行的程序。C++ 属于编译型语言，源代码必须经过编译器处理才能运行。本节将以实际示例为主，从一个文件的编译开始，逐步扩展到多文件项目和构建系统，帮助读者建立对工具链的完整认识。下面假设我们有一个最基本的程序 `main.cpp`：

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, RoboMaster!" << endl;
    return 0;
}
```

要将这个源文件编译成可执行文件，可以使用 GNU 编译器集合（GCC）中的 `g++` 命令行工具。假设读者已经在系统中安装了 `g++`，可以通过以下命令进行编译：

```
g++ main.cpp -o main
```

这里，`g++` 是调用 C++ 编译器的命令，`main.cpp` 是源文件名，`-o main` 指定输出文件名为 `main`（在 Windows 上会生成 `main.exe`）。执行该命令后，如果源代码没有语法错误，编译器会生成一个可执行文件 `main`。要运行生成的程序，可以在终端中输入：

```
./main      # 在类 Unix 系统 (Linux、macOS) 上
```

至此，我们完成了一次最基础的 C++ 编译。

尽管我们只敲了一条命令，但编译器内部实际经历了四个阶段：

1. 预处理 Preprocessing

预处理器会处理以 `#` 开头的预处理指令，如 `#include`、`#define` 等。它会将包含的头文件内容插入到源文件中，展开宏定义等。预处理后的代码会生成一个临时文件，通常以 `.i` 结尾。我们可以用以下命令仅执行预处理阶段：

```
g++ -E main.cpp -o main.i
```

这时，`main.i` 文件中包含了展开后的代码。

2. 编译 Compilation

编译器将预处理后的代码转换为汇编代码。这个阶段会进行语法分析、语义检查和优化等。生成的汇编代码通常以 `.s` 结尾。可以用以下命令仅执行编译阶段：

```
g++ -S main.i -o main.s
```

输出文件 `main.s` 内是汇编语言。这时，我们可以查看汇编代码，了解编译器如何将 C++ 代码转换为底层指令。

3. 汇编 Assembly

汇编器将汇编代码转换为机器码，生成目标文件（object file），通常以 `.o` 结尾。目标文件包含了机器指令，但还不能独立运行。可以用以下命令仅执行汇编阶段：

```
g++ -c main.s -o main.o
```

这时，`main.o` 文件中包含了机器码，但还不能执行，因为它可能还依赖其他目标文件或库。

4. 链接 Linking

链接器将一个或多个目标文件和所需的库文件合并，生成最终的可执行文件。在这个阶段，链接器会解析符号引用，确保所有函数和变量都能正确连接。可以用以下命令仅执行链接阶段：

```
g++ main.o -o main
```

最终生成的 `main` 文件就是可执行程序，可以直接运行。

1.5.3. 变量与基本数据类型

程序的本质是处理数据。无论是计算电机转速、追踪目标位置，还是判断装甲板颜色，都需要在内存中存储和操作各种数据。C++ 通过变量来管理这些数据，而数据类型则决定了变量能够存储什么样的值、占用多少内存空间，以及可以进行哪些运算。

1.5.3.1. 变量的声明与初始化

变量是程序中用于存储数据的命名内存区域。在 C++ 中，使用变量之前必须先声明它的类型和名称。声明告诉编译器两件事：为这个变量分配多大的内存空间，以及如何解释这块内存中的二进制数据。

最基本的变量声明形式是“类型名 变量名”：

```
int score;           // 声明一个整数变量  
double temperature; // 声明一个双精度浮点数变量
```

仅仅声明变量而不赋值是危险的做法。未初始化的变量包含的是内存中的随机数据，使用它们会导致不可预测的行为。因此，声明变量时应当同时进行初始化：

```
int score = 0;  
double temperature = 36.5;
```

C++11 引入了统一初始化语法，使用花括号进行初始化。这种方式不仅适用于基本类型，也适用于复杂类型，并且能够防止某些隐式类型转换带来的精度损失：

```
int count{10};  
double pi{3.14159};  
int dangerous{3.14}; // 编译器会警告：浮点数转整数会丢失精度
```

当初始值的类型与变量类型不完全匹配时，花括号初始化会进行更严格的检查，这有助于在编译阶段发现潜在的错误。

1.5.3.2. 整数类型

整数是最常用的数据类型之一。C++ 提供了多种整数类型，它们的区别在于能够表示的数值范围和占用的内存大小。

`int` 是最常用的整数类型，在现代 64 位系统上通常占用 4 个字节（32 位），能够表示大约 ± 21 亿的范围。对于大多数日常计算，`int` 已经足够：

```
int motorSpeed = 3000;      // 电机转速 RPM  
int bulletCount = 42;        // 弹丸数量  
int targetDistance = 5000; // 目标距离 mm
```

当需要更大或更小的数值范围时，可以使用其他整数类型。`short` 通常占用 2 个字节，适合存储较小的数值以节省内存；`long` 和 `long long` 则提供更大的数值范围，后者保证至少 64 位，能够表示约 ± 922 亿的范围。

```
short sensorReading = 1024;          // 传感器读数  
long long timestamp = 1703491200000; // 毫秒级时间戳
```

整数类型还分为有符号 (signed) 和无符号 (unsigned) 两种。有符号类型可以表示负数，无符号类型只能表示非负数，但正数的范围扩大一倍。默认情况下整数类型是有符号的，如果明确知道某个值不会为负（如数组索引、计数器），可以使用无符号类型：

```
unsigned int frameCount = 0; // 帧计数，不会为负  
size_t arraySize = 100; // size_t 是无符号类型，用于表示大小
```

需要注意的是，无符号整数在减法运算中可能产生意外结果。例如， $0u - 1$ 的结果不是 -1 ，而是一个非常大的正数（无符号整数的最大值）。在 RoboMaster 开发中，涉及可能为负的计算时应当使用有符号类型。

1.5.3.3. 浮点类型

浮点类型用于表示带有小数部分的数值，在机器人控制中随处可见：角度、速度、PID 参数、坐标位置等都需要用浮点数表示。

C++ 提供三种浮点类型：`float`（单精度）、`double`（双精度）和 `long double`（扩展精度）。它们的区别在于精度和范围：

```
float angle = 45.0f; // 单精度，约 7 位有效数字  
double position = 1234.56789; // 双精度，约 15 位有效数字
```

`float` 占用 4 个字节，`double` 占用 8 个字节。虽然 `float` 节省内存，但其精度有限。在连续计算中，精度损失会逐渐累积，导致结果偏差。因此，除非有明确的内存或性能限制，一般推荐使用 `double`。

浮点数字面量默认是 `double` 类型。如果要表示 `float` 类型的字面量，需要在数字后加上 `f` 或 `F` 后缀：

```
float f1 = 3.14; // 3.14 是 double，隐式转换为 float，可能有精度损失  
float f2 = 3.14f; // 3.14f 是 float，无需转换
```

浮点数的一个重要特性是它无法精确表示所有十进制小数。例如， 0.1 在二进制浮点数中是一个无限循环小数，存储时会有微小的舍入误差。这意味着直接比较两个浮点数是否相等往往不可靠的：

```
double a = 0.1 + 0.2;  
double b = 0.3;  
if (a == b) { // 危险！可能不相等  
    // ...  
}
```

正确的做法是判断两个浮点数的差值是否在可接受的误差范围内：

```
const double epsilon = 1e-9;  
if (std::abs(a - b) < epsilon) {  
    // 认为相等  
}
```

1.5.3.4. 字符类型

`char` 类型用于存储单个字符，占用 1 个字节。字符字面量用单引号括起：

```
char grade = 'A';  
char newline = '\n';  
char tab = '\t';
```

在 C++ 中，`char` 本质上是一个小整数，存储的是字符的编码值。在 ASCII 编码中，'A' 的值是 65，'0' 的值是 48。这一特性使得字符可以参与算术运算：

```
char c = 'A';
c = c + 1; // c 现在是 'B'

int digit = '7' - '0'; // 将字符 '7' 转换为整数 7
```

反斜杠 \ 开头的是转义字符，用于表示一些特殊字符：`\n` 表示换行，`\t` 表示制表符，`\\"` 表示反斜杠本身，`\'` 表示单引号。

对于需要处理中文等非 ASCII 字符的场景，`char` 类型不再适用。C++11 引入了 `char16_t` 和 `char32_t` 用于 Unicode 字符，但更常见的做法是使用 `std::string` 配合 UTF-8 编码处理多语言文本。

1.5.3.5. 布尔类型

`bool` 类型用于表示逻辑值，只有两个可能的取值：`true`（真）和 `false`（假）。布尔类型在条件判断和逻辑运算中广泛使用：

```
bool isTargetDetected = true;
bool isAutoAimEnabled = false;
bool shouldShoot = isTargetDetected && isAutoAimEnabled;
```

虽然 `bool` 在逻辑上只需要 1 位就能表示，但由于内存对齐的原因，它通常占用 1 个字节。

在 C++ 中，布尔值可以隐式转换为整数（`false` 转为 0，`true` 转为 1），整数也可以隐式转换为布尔值（0 转为 `false`，非零值转为 `true`）。这种转换在条件语句中很常见：

```
int errorCode = someFunction();
if (errorCode) { // 非零值被视为 true
    std::cout << "发生错误" << std::endl;
}
```

尽管这种隐式转换很方便，但过度依赖它会降低代码的可读性。在表达逻辑意图时，显式使用布尔表达式更为清晰。

1.5.3.6. auto 关键字

C++11 引入的 `auto` 关键字允许编译器根据初始值自动推断变量的类型。这在类型名称冗长或复杂时特别有用：

```
auto count = 10;           // 推断为 int
auto pi = 3.14159;         // 推断为 double
auto message = "Hello";   // 推断为 const char*
```

`auto` 的价值在复杂类型中体现得更为明显。例如，在使用标准库容器时，迭代器的类型往往非常冗长：

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// 不使用 auto
std::vector<int>::iterator it1 = numbers.begin();

// 使用 auto, 简洁得多
auto it2 = numbers.begin();
```

使用 `auto` 时需要注意几点。首先，`auto` 声明的变量必须在声明时初始化，否则编译器无法推断类型。其次，`auto` 推断的类型可能与预期不同，特别是涉及引用和常量时。例如：

```
const int& ref = someValue;
auto a = ref; // a 的类型是 int, 不是 const int&
```

如果需要保留引用或常量属性，应当显式声明：

```
const auto& b = ref; // b 的类型是 const int&
```

在 RoboMaster 开发中，`auto` 常用于 lambda 表达式、迭代器和复杂模板类型的声明，能够显著提高代码的简洁性。但对于基本类型，显式声明往往更清晰——`int count = 10` 比 `auto count = 10` 更直观地表达了变量的用途。

1.5.3.7. 类型转换

不同类型的数据在运算时可能需要相互转换。C++ 支持隐式转换和显式转换两种方式。

隐式转换由编译器自动完成，通常发生在混合类型运算中。基本原则是“向更宽的类型转换”，以避免数据丢失：

```
int a = 10;
double b = 3.14;
double c = a + b; // a 被隐式转换为 double, 结果为 13.14
```

然而，隐式转换也可能导致意外的结果。例如，将浮点数赋值给整数变量时，小数部分会被截断：

```
int x = 3.99; // x 的值是 3, 不是 4
```

当需要明确进行类型转换时，应当使用显式转换。C++ 提供了四种转换运算符，其中最常用的是 `static_cast`：

```
double pi = 3.14159;
int truncated = static_cast<int>(pi); // 显式截断为 3

int dividend = 7;
int divisor = 2;
double result = static_cast<double>(dividend) / divisor; // 结果为 3.5
```

`static_cast` 用于编译期可确定的安全转换，如数值类型之间的转换、向上转型等。其他三种转换运算符——`dynamic_cast`、`const_cast` 和 `reinterpret_cast`——用于更特殊的场景，将在后续章节介绍。

C 语言风格的强制转换（如 `(int)3.14`）在 C++ 中仍然有效，但不推荐使用。C++ 的转换运算符更加明确，也更容易在代码中搜索和审查。

1.5.3.8. 类型大小与 `sizeof` 运算符

不同平台上基本类型的大小可能有所不同。C++ 标准只规定了最小要求（如 `int` 至少 16 位），具体大小由编译器和平台决定。使用 `sizeof` 运算符可以查询类型或变量所占的字节数：

```
#include <iostream>
using namespace std;

int main() {
    cout << "char: " << sizeof(char) << " 字节" << endl;
    cout << "short: " << sizeof(short) << " 字节" << endl;
    cout << "int: " << sizeof(int) << " 字节" << endl;
```

```

        cout << "long: " << sizeof(long) << " 字节" << endl;
        cout << "long long: " << sizeof(long long) << " 字节" << endl;
        cout << "float: " << sizeof(float) << " 字节" << endl;
        cout << "double: " << sizeof(double) << " 字节" << endl;
        cout << "bool: " << sizeof(bool) << " 字节" << endl;
    }
}

```

在典型的 64 位 Linux 系统上，输出通常为：

```

char: 1 字节
short: 2 字节
int: 4 字节
long: 8 字节
long long: 8 字节
float: 4 字节
double: 8 字节
bool: 1 字节

```

如果需要确切大小的整数类型，可以使用 `<cstdint>` 头文件中定义的定宽类型：

```

#include <cstdint>

int8_t a; // 精确 8 位有符号整数
int16_t b; // 精确 16 位有符号整数
int32_t c; // 精确 32 位有符号整数
int64_t d; // 精确 64 位有符号整数

uint8_t e; // 精确 8 位无符号整数
uint16_t f; // 精确 16 位无符号整数
uint32_t g; // 精确 32 位无符号整数
uint64_t h; // 精确 64 位无符号整数

```

在 RoboMaster 开发中，与下位机通信时经常需要使用定宽类型来确保数据格式的一致性。例如，串口协议中的帧头、数据长度、校验码等字段通常会明确规定使用多少字节，此时定宽类型就显得尤为重要。

1.5.3.9. 常量与 `const` 关键字

在程序中，有些值一旦确定就不应该被修改，如数学常数、配置参数等。C++ 提供 `const` 关键字来声明常量：

```

const double PI = 3.14159265358979;
const int MAX_BULLET_SPEED = 30; // 单位 m/s

```

使用 `const` 声明的变量必须在声明时初始化，之后任何试图修改它的操作都会导致编译错误。这种编译期检查能够有效防止意外修改重要数据。

C++11 引入了 `constexpr` 关键字，用于声明编译期常量。与 `const` 不同，`constexpr` 要求值必须在编译时就能确定：

```

constexpr double PI = 3.14159265358979;
constexpr int ARRAY_SIZE = 100;

int arr[ARRAY_SIZE]; // 合法，ARRAY_SIZE 在编译期已知

```

对于简单的常量值，`const` 和 `constexpr` 的效果相似。但 `constexpr` 的语义更加明确——它强调这个值在编译时就已经确定，可以用于模板参数、数组大小等需要编译期常量的场合。

相比于使用宏定义常量 (`#define PI 3.14159`)，`const` 和 `constexpr` 具有类型安全性，也遵循作用域规则，是现代 C++ 推荐的做法。

1.5.4. 运算符与表达式

掌握了变量和数据类型之后，下一步是学习如何操作这些数据。运算符是执行特定操作的符号，表达式则是由运算符和操作数组合而成的计算式。C++ 提供了丰富的运算符，从基本的算术运算到复杂的位操作，它们是构建程序逻辑的基础工具。

1.5.4.1. 算术运算符

算术运算符用于执行数学计算，是最直观的一类运算符。C++ 支持加 (+)、减 (-)、乘 (*)、除 (/) 和取模 (%) 五种基本算术运算：

```
int a = 17, b = 5;

int sum = a + b;           // 22
int diff = a - b;          // 12
int prod = a * b;          // 85
int quot = a / b;          // 3 (整数除法，截断小数)
int rem = a % b;           // 2 (取模，即余数)
```

整数除法是初学者常遇到的陷阱。当两个整数相除时，结果仍然是整数，小数部分被直接截断而非四舍五入。如果需要保留小数，至少要将其中一个操作数转换为浮点类型：

```
int dividend = 7;
int divisor = 2;

int result1 = dividend / divisor;                      // 结果为 3
double result2 = dividend / divisor;                   // 结果仍为 3.0，因为除法在
赋值前已完成
double result3 = static_cast<double>(dividend) / divisor; // 结果为 3.5
double result4 = dividend / 2.0;                        // 结果为 3.5, 2.0
是 double
```

取模运算符 % 只能用于整数类型，返回除法的余数。它在许多场景下很有用，例如判断奇偶性、实现循环计数器、或者将数值限制在某个范围内：

```
bool isEven = (number % 2 == 0);                     // 判断偶数
int arrayIndex = counter % arraySize;                // 循环索引
int angleInRange = angle % 360;                      // 将角度限制在 0-359
```

C++ 还提供了自增 (++) 和自减 (--) 运算符，用于将变量的值加 1 或减 1。这两个运算符有前置和后置两种形式，区别在于返回值的时机：

```
int x = 5;
int y = ++x; // 前置：先加 1，再取值。x = 6, y = 6
int z = x++; // 后置：先取值，再加 1。x = 7, z = 6
```

前置形式返回修改后的值，后置形式返回修改前的值。在不需要使用返回值的情况下（如单独的 `i++` 语句），两者效果相同，但前置形式理论上效率略高，因为它不需要保存旧值。在 RoboMaster 代码中，循环变量的自增通常使用 `++i` 的形式。

1.5.4.2. 关系运算符

关系运算符用于比较两个值的大小关系，返回布尔值 `true` 或 `false`。C++ 提供六种关系运算符：

```
int a = 10, b = 20;

bool eq = (a == b);    // 等于, false
bool ne = (a != b);    // 不等于, true
bool lt = (a < b);    // 小于, true
bool le = (a <= b);   // 小于等于, true
bool gt = (a > b);    // 大于, false
bool ge = (a >= b);   // 大于等于, false
```

一个极其常见的错误是将相等判断 `==` 误写成赋值 `=`。这种错误往往不会产生编译错误，却会导致完全不同的行为：

```
int status = 0;

if (status = 1) {      // 错误！这是赋值, status 变为 1, 条件恒为 true
    // 总是执行
}

if (status == 1) {     // 正确, 判断 status 是否等于 1
    // 当 status 为 1 时执行
}
```

为了避免这类错误，有些程序员习惯将常量写在比较运算符的左侧（如 `1 == status`），这样误写成赋值时编译器会报错。不过，现代编译器通常会对条件语句中的赋值操作发出警告，开启足够的警告级别（如 `-Wall`）可以捕获大部分此类问题。

前面提到，浮点数的直接相等比较是不可靠的。在比较浮点数时，应当检查它们的差值是否在可接受的范围内：

```
double targetAngle = 45.0;
double currentAngle = computeAngle(); // 可能有浮点误差
double tolerance = 0.001;

if (std::abs(currentAngle - targetAngle) < tolerance) {
    // 认为角度已到达目标
}
```

1.5.4.3. 逻辑运算符

逻辑运算符用于组合多个布尔表达式，构建复杂的条件判断。C++ 提供三种逻辑运算符：逻辑与 (`&&`)、逻辑或 (`||`) 和逻辑非 (`!`)。

逻辑与要求两个条件都为真，结果才为真：

```
bool canShoot = isTargetLocked && hasBullets && !isOverheated;
```

逻辑或只要有一个条件为真，结果就为真：

```
bool needWarning = isBatteryLow || isTemperatureHigh || hasError;
```

逻辑非将布尔值取反：

```
bool isInvalid = !isValid;
```

逻辑运算符具有短路求值 (short-circuit evaluation) 的特性。对于 `&&`, 如果左侧表达式为假, 右侧表达式不会被求值, 因为结果已经确定为假; 对于 `||`, 如果左侧为真, 右侧同样不会被求值。这一特性不仅能提高效率, 还可以用于避免潜在的错误:

```
// 安全的指针检查: 先判断指针非空, 再访问其成员
if (ptr != nullptr && ptr->isValid()) {
    // 如果 ptr 为空, ptr->isValid() 不会被调用, 避免空指针解引用
}

// 提供默认值
int value = (userInput > 0) ? userInput : defaultValue;
```

在复杂的条件表达式中, 适当使用括号可以提高可读性, 即使在运算符优先级正确的情况下:

```
// 不够清晰
if (a > 0 && b > 0 || c > 0 && d > 0) { }

// 更清晰
if ((a > 0 && b > 0) || (c > 0 && d > 0)) { }
```

1.5.4.4. 赋值运算符

最基本的赋值运算符是 `=`, 它将右侧的值赋给左侧的变量。C++ 还提供了一系列复合赋值运算符, 将算术运算和赋值合并为一步:

```
int x = 10;

x += 5;    // 等价于 x = x + 5, x 变为 15
x -= 3;    // 等价于 x = x - 3, x 变为 12
x *= 2;    // 等价于 x = x * 2, x 变为 24
x /= 4;    // 等价于 x = x / 4, x 变为 6
x %= 4;    // 等价于 x = x % 4, x 变为 2
```

复合赋值运算符不仅使代码更简洁, 在某些情况下还能避免重复计算左侧表达式。例如, `array[computeIndex()] += 1` 只调用一次 `computeIndex()`, 而 `array[computeIndex()] = array[computeIndex()] + 1` 会调用两次。

赋值表达式本身也有值, 其值为赋值后左侧变量的值。这使得链式赋值成为可能:

```
int a, b, c;
a = b = c = 0; // 从右向左执行, 三个变量都被赋值为 0
```

然而, 滥用赋值表达式的值会降低代码可读性, 应当谨慎使用。

1.5.4.5. 位运算符

位运算符直接操作整数的二进制位, 在底层编程和性能敏感的场景中很有用。C++ 提供六种位运算符:

```
int a = 0b1100; // 二进制 1100, 即十进制 12
int b = 0b1010; // 二进制 1010, 即十进制 10

int andResult = a & b; // 按位与: 0b1000, 即 8
int orResult = a | b; // 按位或: 0b1110, 即 14
int xorResult = a ^ b; // 按位异或: 0b0110, 即 6
int notResult = ~a; // 按位取反: 所有位翻转
```

```
int leftShift = a << 2; // 左移 2 位: 0b110000, 即 48
int rightShift = a >> 2; // 右移 2 位: 0b0011, 即 3
```

在 RoboMaster 开发中，位运算常用于以下场景。

状态标志的管理是位运算的典型应用。使用单个整数的不同位表示不同的状态，可以高效地存储和检查多个布尔标志：

```
const uint8_t FLAG_MOTOR_READY = 0b00000001; // 第 0 位
const uint8_t FLAG_SENSOR_OK = 0b00000010; // 第 1 位
const uint8_t FLAG_COMM_ACTIVE = 0b00000100; // 第 2 位
const uint8_t FLAG_AUTO_AIM = 0b00001000; // 第 3 位

uint8_t robotStatus = 0;

// 设置标志
robotStatus |= FLAG_MOTOR_READY; // 将第 0 位置 1
robotStatus |= FLAG_SENSOR_OK; // 将第 1 位置 1

// 清除标志
robotStatus &= ~FLAG_AUTO_AIM; // 将第 3 位置 0

// 检查标志
if (robotStatus & FLAG_MOTOR_READY) {
    // 电机已就绪
}

// 切换标志
robotStatus ^= FLAG_COMM_ACTIVE; // 翻转第 2 位
```

位移运算在处理通信协议时也很常见。例如，将多个字节组合成一个整数，或者从整数中提取特定字节：

```
// 将两个字节组合成 16 位整数（大端序）
uint8_t highByte = 0x12;
uint8_t lowByte = 0x34;
uint16_t combined = (highByte << 8) | lowByte; // 0x1234

// 从 32 位整数中提取各字节
uint32_t value = 0x12345678;
uint8_t byte0 = value & 0xFF; // 0x78
uint8_t byte1 = (value >> 8) & 0xFF; // 0x56
uint8_t byte2 = (value >> 16) & 0xFF; // 0x34
uint8_t byte3 = (value >> 24) & 0xFF; // 0x12
```

左移一位相当于乘以 2，右移一位相当于除以 2（对于无符号数或非负有符号数）。在某些性能关键的代码中，位移可能比乘除法更快，但现代编译器通常会自动进行这类优化，因此不必刻意使用位移替代乘除。

1.5.4.6. 条件运算符

条件运算符 (?:) 是 C++ 中唯一的三元运算符，它根据条件选择两个值中的一个：

```
int max = (a > b) ? a : b; // 如果 a > b, 取 a; 否则取 b
```

条件运算符可以替代简单的 if-else 语句，使代码更紧凑：

```
// 使用 if-else
std::string status;
```

```

if (isOnline) {
    status = "在线";
} else {
    status = "离线";
}

// 使用条件运算符
std::string status = isOnline ? "在线" : "离线";

```

条件运算符也可以嵌套，但过度嵌套会严重损害可读性：

```

// 勉强可以接受
int sign = (x > 0) ? 1 : (x < 0) ? -1 : 0;

// 嵌套过深，难以理解
int result = a ? b ? c : d : e ? f : g; // 不要这样写

```

当逻辑变得复杂时，应当使用普通的 if-else 语句。条件运算符最适合用于简单的二选一场景。

1.5.4.7. 运算符优先级

当一个表达式包含多个运算符时，运算符优先级决定了计算的顺序。C++ 定义了详细的优先级规则，以下是常用运算符按优先级从高到低的大致排列：

1. 后缀运算符：() [] -> . ++(后置) --(后置)
2. 一元运算符：++(前置) --(前置) ! ~+(正号) -(负号) *(解引用) &(取地址)
3. 乘除模：* / %
4. 加减：+ -
5. 位移：<< >>
6. 关系：< <= > >=
7. 相等：== !=
8. 按位与：&
9. 按位异或：^
10. 按位或：|
11. 逻辑与：&&
12. 逻辑或：||
13. 条件：?:
14. 赋值：+= -= 等
15. 逗号：,

记住所有优先级规则是困难的，也没有必要。实践中的建议是：对于不确定优先级的表达式，使用括号明确计算顺序。括号不仅保证了正确性，也提高了代码的可读性：

```

// 不确定优先级，容易出错
int result = a + b * c >> d & e;

// 使用括号，意图明确
int result = ((a + (b * c)) >> d) & e;

```

特别需要注意的是，位运算符的优先级低于关系运算符，这常常导致意外：

```
// 错误：实际执行的是 flags & (FLAG_A == FLAG_A)，结果恒为 flags & 1
if (flags & FLAG_A == FLAG_A) { }

// 正确
if ((flags & FLAG_A) == FLAG_A) { }
```

1.5.4.8. 表达式与语句

理解表达式和语句的区别有助于写出更清晰的代码。表达式是可以求值的代码片段，它产生一个结果；语句是执行某种操作的完整指令，以分号结束。

```
a + b          // 表达式，值为 a 和 b 的和
x = a + b      // 也是表达式，值为赋值后 x 的值
x = a + b;     // 语句，执行赋值操作

int y = 10;     // 声明语句
if (x > 0) {}   // 控制语句
```

任何表达式加上分号都成为表达式语句。有些表达式语句是有意义的（如赋值、函数调用），有些则毫无作用：

```
x = 5;          // 有意义，修改了 x 的值
calculate();    // 有意义，调用了函数
a + b;          // 无意义，计算结果被丢弃（编译器可能警告）
```

在 C++ 中，某些看似语句的结构实际上是表达式。例如，赋值操作返回被赋的值，这使得 `a = b = c = 0` 这样的链式赋值成为可能。再如，逗号运算符连接多个表达式，从左到右依次求值，整个表达式的值为最右侧表达式的值：

```
int x = (a = 1, b = 2, a + b); // x = 3
```

逗号运算符在 for 循环中偶尔有用（如同时更新多个变量），但在其他地方使用会降低代码可读性，应当避免。

理解了运算符和表达式的工作方式，就具备了编写计算逻辑的基础。下一节将介绍控制语句，它们决定了程序的执行流程——哪些代码会被执行，哪些会被跳过，哪些会重复执行。

1.5.5. 控制语句

程序并非总是从头到尾顺序执行。根据不同的条件选择不同的执行路径，或者重复执行某段代码直到满足特定条件，这些都是程序设计中的基本需求。控制语句正是实现这些功能的工具，它们决定了程序的执行流程。C++ 提供了三类控制语句：条件语句用于分支选择，循环语句用于重复执行，跳转语句用于改变执行顺序。

1.5.5.1. if 语句

if 语句是最基本的条件语句，它根据条件表达式的真假决定是否执行某段代码：

```
if (temperature > 60.0) {
    std::cout << "警告：电机温度过高！" << std::endl;
    disableMotor();
}
```

当条件为真时，花括号内的代码块被执行；当条件为假时，代码块被跳过。花括号定义了 if 语句的作用范围，即使只有一条语句，也建议使用花括号，这样可以避免后续添加代码时遗漏花括号导致的逻辑错误：

```

// 危险的写法
if (error)
    logError();
    shutdown(); // 这行总是执行，不属于 if 语句！

// 安全的写法
if (error) {
    logError();
    shutdown(); // 现在正确地属于 if 语句
}

```

当需要在条件为假时执行另一段代码，可以使用 `else` 子句：

```

if (batteryLevel > 20) {
    status = "正常";
} else {
    status = "电量不足";
    enablePowerSaving();
}

```

多个条件可以用 `else if` 串联起来，形成多路分支：

```

if (distance < 1000) {
    range = "近距离";
    adjustStrategy(CLOSE_RANGE);
} else if (distance < 3000) {
    range = "中距离";
    adjustStrategy(MID_RANGE);
} else if (distance < 5000) {
    range = "远距离";
    adjustStrategy(LONG_RANGE);
} else {
    range = "超出范围";
    adjustStrategy(OUT_OF_RANGE);
}

```

`else if` 链接顺序检查每个条件，一旦某个条件为真，执行对应的代码块后就跳出整个结构，后续条件不再检查。因此，条件的排列顺序很重要——应当将最可能满足或最需要优先处理的条件放在前面。

`if` 语句可以嵌套，但过深的嵌套会严重损害代码可读性。当嵌套超过两三层时，应当考虑重构代码，例如提取函数或使用提前返回（early return）：

```

// 嵌套过深，难以阅读
if (conditionA) {
    if (conditionB) {
        if (conditionC) {
            doSomething();
        }
    }
}

// 使用提前返回，更清晰
if (!conditionA) return;
if (!conditionB) return;
if (!conditionC) return;
doSomething();

```

C++17 引入了带初始化的 if 语句，允许在条件判断前声明变量，该变量的作用域限于 if 语句内部：

```
if (auto result = computeSomething(); result > threshold) {
    process(result);
} else {
    handleFailure(result);
}
// result 在这里不可见
```

这种写法在需要检查函数返回值时特别有用，它将变量声明和条件检查合并，同时限制了变量的作用域。

1.5.5.2. switch 语句

当需要根据一个变量的不同取值执行不同的代码时，switch 语句比一连串的 if-else if 更加清晰：

```
enum class RobotState { Idle, Patrol, Attack, Retreat };

RobotState state = getCurrentState();

switch (state) {
    case RobotState::Idle:
        standby();
        break;
    case RobotState::Patrol:
        moveAlongPath();
        scanForTargets();
        break;
    case RobotState::Attack:
        aimAtTarget();
        fire();
        break;
    case RobotState::Retreat:
        moveToSafeZone();
        break;
    default:
        handleUnknownState();
        break;
}
```

switch 语句的工作方式是：计算括号内表达式的值，然后跳转到匹配的 case 标签处开始执行。每个 case 后的 break 语句用于跳出 switch 结构；如果省略 break，程序会继续执行下一个 case 的代码，这称为“贯穿”（fall-through）。

贯穿行为有时是有意为之的，例如多个 case 共享同一段代码：

```
char grade = getGrade();

switch (grade) {
    case 'A':
    case 'B':
    case 'C':
        std::cout << "通过" << std::endl;
        break;
    case 'D':
```

```

        case 'F':
            std::cout << "未通过" << std::endl;
            break;
        default:
            std::cout << "无效成绩" << std::endl;
            break;
    }
}

```

然而，无意的贯穿是常见的 bug 来源。C++17 引入了 `[[fallthrough]]` 属性，用于明确表示贯穿是有意的，同时让编译器在其他地方对遗漏的 `break` 发出警告：

```

switch (command) {
    case CMD_INIT:
        initialize();
        [[fallthrough]];
    // 明确表示继续执行下一个 case
    case CMD_START:
        start();
        break;
    // ...
}

```

`switch` 语句的表达式必须是整数类型或枚举类型，不能是浮点数或字符串。`case` 标签必须是编译期常量，不能是变量。`default` 标签处理所有未匹配的情况，虽然不是必需的，但添加 `default` 是良好的习惯，它可以捕获意外的输入值。

在 `case` 内部声明变量需要特别注意。由于 `switch` 的跳转特性，直接在 `case` 中声明变量可能导致跳过初始化，编译器会报错。解决方法是用花括号创建局部作用域：

```

switch (type) {
    case TYPE_A: {
        int localVar = 10; // 在局部作用域内声明
        process(localVar);
        break;
    }
    case TYPE_B: {
        std::string msg = "hello";
        display(msg);
        break;
    }
}

```

1.5.5.3. while 循环

`while` 循环在条件为真时重复执行代码块。它首先检查条件，如果为真则执行循环体，然后再次检查条件，如此反复直到条件为假：

```

int attempts = 0;
const int maxAttempts = 5;

while (attempts < maxAttempts && !connectionEstablished()) {
    std::cout << "尝试连接..." (" << attempts + 1 << "/" << maxAttempts << ")"
<< std::endl;
    tryConnect();
    attempts++;
    sleep(1000);
}

```

```

if (connectionEstablished()) {
    std::cout << "连接成功" << std::endl;
} else {
    std::cout << "连接失败" << std::endl;
}

```

`while` 循环适合用于事先不知道循环次数的场景，例如等待某个条件满足、处理不定长度的输入、或者实现重试机制。

编写 `while` 循环时必须确保循环条件最终会变为假，否则会产生无限循环。无限循环有时是有意为之的（如主事件循环），但意外的无限循环会导致程序挂起：

```

// 意外的无限循环：忘记更新 i
int i = 0;
while (i < 10) {
    process(i);
    // 忘记 i++, 循环永不结束
}

// 有意的无限循环：主控制循环
while (true) {
    readSensors();
    updateState();
    sendCommands();

    if (shutdownRequested()) {
        break; // 使用 break 退出
    }
}

```

1.5.5.4. do-while 循环

`do-while` 循环与 `while` 循环类似，但它先执行循环体，再检查条件。这保证了循环体至少执行一次：

```

int input;
do {
    std::cout << "请输入一个正整数：";
    std::cin >> input;

    if (input <= 0) {
        std::cout << "输入无效，请重试。" << std::endl;
    }
} while (input <= 0);

std::cout << "你输入的是：" << input << std::endl;

```

`do-while` 的典型应用场景是需要先执行操作再判断是否继续的情况，如用户输入验证、菜单交互等。由于循环体至少执行一次，使用前应当确认这符合程序逻辑。

注意 `do-while` 语句以分号结尾，这与其他循环不同，容易遗漏。

1.5.5.5. for 循环

`for` 循环是最常用的循环结构，它将初始化、条件检查和迭代更新集中在一行，特别适合已知循环次数的场景：

```

// 遍历数组
int scores[] = {85, 92, 78, 96, 88};
int sum = 0;

for (int i = 0; i < 5; i++) {
    sum += scores[i];
}

double average = static_cast<double>(sum) / 5;

```

`for` 语句的三个部分——初始化、条件、迭代——都是可选的。省略条件表达式等同于条件恒为真，这是创建无限循环的常见方式：

```

for (;;) {
    // 无限循环
    if (shouldExit()) break;
}

```

`for` 循环的初始化部分可以声明多个同类型的变量，迭代部分也可以包含多个表达式，用逗号分隔：

```

// 双指针技术：从两端向中间遍历
for (int left = 0, right = size - 1; left < right; left++, right--) {
    if (array[left] > array[right]) {
        std::swap(array[left], array[right]);
    }
}

```

C++11 引入了范围 `for` 循环 (range-based for loop)，它提供了一种更简洁的方式遍历容器或数组中的元素：

```

std::vector<int> numbers = {1, 2, 3, 4, 5};

// 传统 for 循环
for (size_t i = 0; i < numbers.size(); i++) {
    std::cout << numbers[i] << " ";
}

// 范围 for 循环
for (int num : numbers) {
    std::cout << num << " ";
}

```

范围 `for` 循环自动处理迭代细节，代码更加简洁，也不容易出现索引越界等错误。如果需要修改元素，应当使用引用；如果元素较大且不需要修改，应当使用常量引用以避免复制开销：

```

std::vector<std::string> names = {"Alice", "Bob", "Charlie"};

// 修改元素：使用引用
for (std::string& name : names) {
    name = "Mr./Ms. " + name;
}

// 只读访问：使用常量引用，避免复制
for (const std::string& name : names) {
    std::cout << name << std::endl;
}

```

```
// 使用 auto 简化类型声明
for (const auto& name : names) {
    std::cout << name << std::endl;
}
```

在 RoboMaster 开发中，范围 for 循环常用于遍历检测到的目标列表、处理传感器数据数组等场景。

1.5.5.6. break 与 continue

`break` 语句用于立即退出最内层的循环或 `switch` 语句。当在循环中遇到某个条件需要提前结束时，`break` 非常有用：

```
// 在数组中查找目标值
int target = 42;
int foundIndex = -1;

for (int i = 0; i < arraySize; i++) {
    if (array[i] == target) {
        foundIndex = i;
        break; // 找到后立即退出，无需继续搜索
    }
}
```

`continue` 语句用于跳过当前迭代的剩余部分，直接进入下一次迭代。它适合用于在满足某些条件时跳过处理：

```
// 处理有效数据，跳过无效数据
for (const auto& reading : sensorReadings) {
    if (!reading.isValid()) {
        continue; // 跳过无效读数
    }

    processReading(reading);
    updateStatistics(reading);
}
```

`break` 和 `continue` 只影响最内层的循环。在嵌套循环中，如果需要从内层循环跳出到外层，可以使用标志变量或将循环封装成函数并使用 `return`：

```
// 使用标志变量
bool found = false;
for (int i = 0; i < rows && !found; i++) {
    for (int j = 0; j < cols; j++) {
        if (matrix[i][j] == target) {
            found = true;
            break; // 只跳出内层循环
        }
    }
}

// 更优雅的方式：封装成函数
std::pair<int, int> findInMatrix(int target) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (matrix[i][j] == target) {
                return {i, j}; // 直接返回，退出所有循环
            }
        }
    }
}
```

```

        }
    }
    return {-1, -1}; // 未找到
}

```

1.5.5.7. 循环的选择

三种循环结构在功能上是等价的，任何一种都可以改写成另一种。选择哪种循环主要取决于代码的清晰度：

`for` 循环适合循环次数已知或有明确迭代模式的场景。它将循环的三个要素集中在一处，便于理解循环的行为。遍历数组、计数循环、有规律的迭代都适合用 `for`。

`while` 循环适合循环次数未知、依赖于运行时条件的场景。等待事件、处理流数据、重试机制等都适合用 `while`。

`do-while` 循环适合需要先执行再判断的场景，如用户输入验证。由于循环体必定执行一次，使用场景相对较少。

```

// 计数循环: for 最清晰
for (int i = 0; i < 10; i++) { }

// 等待条件: while 最清晰
while (!isReady()) {
    wait();
}

// 输入验证: do-while 最清晰
do {
    input = getInput();
} while (!isValid(input));

```

1.5.5.8. 控制语句的嵌套与组合

实际程序中，控制语句往往需要组合使用。条件语句可以嵌套在循环中，循环也可以嵌套在条件语句中。合理的嵌套能够表达复杂的逻辑，但过度嵌套会使代码难以理解和维护。

```

// RoboMaster 自动瞄准逻辑示例
void autoAimLoop() {
    while (isAutoAimEnabled()) {
        auto targets = detectTargets();

        if (targets.empty()) {
            continue; // 无目标，进入下一帧
        }

        // 选择最优目标
        Target* bestTarget = nullptr;
        double bestScore = -1;

        for (const auto& target : targets) {
            if (!target.isValid()) {
                continue;
            }

            double score = evaluateTarget(target);
            if (score > bestScore) {

```

```

        bestScore = score;
        bestTarget = &target;
    }
}

if (bestTarget != nullptr) {
    aimAt(*bestTarget);

    if (isAimStable() && canFire()) {
        fire();
    }
}

waitForNextFrame();
}
}
}

```

保持代码清晰的几个原则：控制嵌套深度，一般不超过三层；将复杂的条件表达式提取为命名良好的布尔变量或函数；使用提前返回或 `continue` 减少嵌套；将独立的逻辑块提取为函数。

控制语句是程序逻辑的骨架。掌握了变量、运算符和控制语句，就具备了编写基本程序的能力。然而，随着程序规模的增长，将所有代码写在 `main` 函数中会变得难以管理。下一节将介绍函数，它是组织和复用代码的基本单元。

1.5.6. 函数基础

随着程序规模的增长，将所有代码堆砌在 `main` 函数中会迅速变得不可维护。相似的代码片段可能在多处重复出现，修改一处逻辑需要同时修改多个位置，程序的整体结构也变得难以把握。函数是解决这些问题的基本工具——它将一段完成特定任务的代码封装起来，赋予一个名称，使其可以在需要时被调用。良好的函数设计能够提高代码的可读性、可维护性和复用性。

1.5.6.1. 函数的定义与调用

一个完整的函数定义包含返回类型、函数名、参数列表和函数体四个部分：

```
// 返回类型 函数名 参数列表
double average(int a, int b) {
    // 函数体
    return static_cast<double>(a + b) / 2;
}
```

返回类型指定函数返回值的类型。如果函数不返回任何值，使用 `void` 作为返回类型。函数名应当清晰地描述函数的功能，通常使用动词或动词短语。参数列表定义了函数接受的输入，多个参数用逗号分隔；如果函数不需要参数，参数列表为空。函数体是实际执行的代码，用花括号包围。

定义好函数后，通过函数名加括号的方式调用它：

```
int x = 10, y = 20;
double avg = average(x, y); // 调用函数，传入参数 x 和 y
std::cout << "平均值: " << avg << std::endl;
```

函数调用时，实际传入的值称为实参 (argument)，函数定义中的变量称为形参 (parameter)。调用发生时，实参的值被复制给形参，函数在自己的作用域内使用这些副本进行计算。

`return` 语句用于从函数返回一个值并结束函数执行。对于返回 `void` 的函数，可以使用不带值的 `return;` 提前退出，也可以让函数自然执行到末尾结束：

```
void printPositive(int value) {
    if (value <= 0) {
        return; // 提前退出，不打印非正数
    }
    std::cout << value << std::endl;
}
```

1.5.6.2. 函数声明与定义分离

在 C++ 中，函数必须在调用之前被声明。声明告诉编译器函数的存在及其接口（返回类型、名称、参数类型），而定义提供函数的具体实现。声明可以出现多次，但定义只能有一次。

```
// 函数声明（也称为函数原型）
double calculateDistance(double x1, double y1, double x2, double y2);

int main() {
    // 可以调用，因为前面已经声明
    double dist = calculateDistance(0, 0, 3, 4);
    std::cout << "距离: " << dist << std::endl;
    return 0;
}

// 函数定义
double calculateDistance(double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    return std::sqrt(dx * dx + dy * dy);
}
```

将声明放在文件开头或头文件中，定义放在源文件中，是 C++ 项目组织代码的常见方式。这种分离使得多个源文件可以共享同一个函数，也使得编译器能够独立编译各个源文件。

声明中的参数名是可选的，只需要类型信息即可。但为了可读性，通常会保留有意义的参数名：

```
// 合法但可读性差
double calculateDistance(double, double, double, double);

// 更好的写法
double calculateDistance(double x1, double y1, double x2, double y2);
```

1.5.6.3. 参数传递

C++ 中参数传递有三种主要方式：值传递、引用传递和指针传递。理解它们的区别对于写出正确且高效的代码至关重要。

值传递是默认方式。调用函数时，实参的值被复制给形参，函数内部对形参的修改不会影响实参：

```
void increment(int n) {
    n++; // 只修改了副本
    std::cout << "函数内: " << n << std::endl;
}
```

```

int main() {
    int value = 10;
    increment(value);
    std::cout << "函数外: " << value << std::endl; // 仍然是 10
    return 0;
}

```

值传递的优点是安全——函数不会意外修改调用者的数据。缺点是对于大型对象，复制的开销可能很大。

引用传递通过在参数类型后加`&`实现。此时形参是实参的别名，函数内部对形参的修改会直接反映到实参上：

```

void increment(int& n) {
    n++; // 直接修改原变量
}

int main() {
    int value = 10;
    increment(value);
    std::cout << value << std::endl; // 输出 11
    return 0;
}

```

引用传递有两个主要用途。一是允许函数修改调用者的变量，如上例所示。二是避免复制大型对象的开销——即使不需要修改，也可以使用常量引用传递：

```

// 传值：复制整个 vector，开销大
void processData(std::vector<double> data);

// 常量引用：不复制，不允许修改，安全高效
void processData(const std::vector<double>& data);

```

使用常量引用传递大型对象是 C++ 中的常见惯用法。对于基本类型（如`int`、`double`），由于复制成本很低，通常直接值传递即可。

指针传递将在后续章节详细介绍。简单来说，它通过传递变量的地址来实现类似引用的效果，但语法更显式，且允许传递空指针表示“无值”。

1.5.6.4. 返回值

函数可以通过`return`语句返回一个值给调用者。返回值的类型必须与函数声明的返回类型兼容：

```

int findMax(int a, int b, int c) {
    int max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
}

```

返回值可以是基本类型、对象、引用或指针。返回局部变量的值是安全的，因为返回时会复制该值；但返回局部变量的引用或指针是危险的，因为函数结束后局部变量被销毁，引用或指针将指向无效内存：

```

// 安全：返回值的副本
int getValue() {
    int local = 42;

```

```

        return local; // 返回 local 的副本
    }

// 危险：返回局部变量的引用
int& getReference() {
    int local = 42;
    return local; // 错误！local 在函数返回后销毁
}

```

C++11 允许函数返回初始化列表，结合 `auto` 可以方便地返回多个值：

```

std::pair<double, double> getPosition() {
    return {3.14, 2.71};
}

```

```
auto [x, y] = getPosition(); // C++17 结构化绑定
```

对于需要返回多个值的场景，除了返回 `std::pair` 或 `std::tuple`，也可以使用输出参数（通过引用或指针传递）：

```

// 方式一：返回结构体或 pair
struct Result {
    bool success;
    double value;
};

Result compute(double input);

// 方式二：输出参数
bool compute(double input, double& output);

```

两种方式各有优劣。返回结构体语义更清晰，输出参数在某些场景下（如需要区分返回值和错误码）更方便。现代 C++ 倾向于使用返回值而非输出参数。

1.5.6.5. 默认参数

函数参数可以指定默认值。调用时如果不提供该参数，就使用默认值：

```

void connect(const std::string& host, int port = 8080, int timeout = 5000) {
    std::cout << "连接到 " << host << ":" << port
        << ", 超时 " << timeout << "ms" << std::endl;
}

int main() {
    connect("192.168.1.1");           // 使用默认端口和超时
    connect("192.168.1.1", 9000);     // 自定义端口，默认超时
    connect("192.168.1.1", 9000, 3000); // 全部自定义
    return 0;
}

```

默认参数必须从右向左连续指定，不能跳过中间的参数：

```

void func(int a, int b = 2, int c = 3); // 合法
void func(int a = 1, int b, int c = 3); // 非法，b 没有默认值但在有默认值的参数之后

```

如果函数声明和定义分离，默认参数应当只在声明中指定，不要在定义中重复：

```

// 头文件中的声明
void log(const std::string& message, int level = 1);

// 源文件中的定义

```

```
void log(const std::string& message, int level) { // 不重复默认值
    // ...
}
```

默认参数在编译时确定，因此必须是编译期常量或能在编译时求值的表达式。

1.5.6.6. 函数重载

C++ 允许定义多个同名但参数列表不同的函数，这称为函数重载。编译器根据调用时提供的参数类型和数量选择合适的版本：

```
// 计算两个整数的平均值
double average(int a, int b) {
    return static_cast<double>(a + b) / 2;
}

// 计算三个整数的平均值
double average(int a, int b, int c) {
    return static_cast<double>(a + b + c) / 3;
}

// 计算浮点数数组的平均值
double average(const std::vector<double>& values) {
    double sum = 0;
    for (double v : values) sum += v;
    return sum / values.size();
}

int main() {
    std::cout << average(10, 20) << std::endl;           // 调用第一个版本
    std::cout << average(10, 20, 30) << std::endl;       // 调用第二个版本
    std::cout << average({1.5, 2.5, 3.5}) << std::endl; // 调用第三个版本
    return 0;
}
```

重载函数必须在参数的数量或类型上有所不同。仅返回类型不同不构成有效的重载，因为编译器无法仅根据返回类型确定调用哪个版本：

```
int process(int x);
double process(int x); // 错误！仅返回类型不同
```

重载解析是编译器选择最佳匹配函数的过程。当存在多个可能的匹配时，编译器会选择“最佳匹配”——需要最少类型转换的版本。如果无法确定唯一的最佳匹配，编译器会报告歧义错误：

```
void print(int x);
void print(double x);

print(3.14f); // float 到 int 和 float 到 double 都需要转换，可能产生歧义
```

函数重载使得同一操作可以作用于不同类型的数据，提高了接口的一致性。在 RoboMaster 开发中，重载常见于各种初始化函数、数据转换函数等。

1.5.6.7. 内联函数

函数调用有一定的开销：保存当前状态、跳转到函数代码、执行、返回。对于非常短小且频繁调用的函数，这些开销可能超过函数体本身的执行时间。内联函数通过在调用处直接展开函数代码来消除这种开销：

```
inline int max(int a, int b) {
    return (a > b) ? a : b;
}
```

`inline` 关键字是对编译器的建议而非强制命令。现代编译器会自行判断是否内联——即使没有 `inline` 关键字，编译器也可能内联短小的函数；即使有 `inline`，编译器也可能拒绝内联过大或递归的函数。

内联函数的定义通常放在头文件中，因为编译器需要在调用处看到完整的函数体才能展开。这与普通函数不同——普通函数的定义通常放在源文件中。

```
// math_utils.h
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

inline int square(int x) {
    return x * x;
}

#endif
```

在实践中，不必过度关注内联优化。编译器的优化能力很强，除非性能分析表明某个函数调用是瓶颈，否则让编译器自行决定即可。

1.5.6.8. 递归函数

函数可以调用自身，这称为递归。递归是一种强大的问题解决技术，适合处理具有自相似结构的问题：

```
// 计算阶乘: n! = n * (n-1)!
int factorial(int n) {
    if (n <= 1) {
        return 1; // 基本情况: 0! = 1! = 1
    }
    return n * factorial(n - 1); // 递归情况
}
```

每个递归函数都必须有基本情况（base case）和递归情况（recursive case）。基本情况定义了不再递归的条件，递归情况将问题分解为更小的子问题。缺少基本情况或递归未能向基本情况收敛会导致无限递归，最终耗尽栈空间导致程序崩溃。

递归的经典例子包括树的遍历、快速排序、归并排序等。以下是一个在 RoboMaster 中可能用到的例子——路径搜索的简化版本：

```
bool findPath(int x, int y, int targetX, int targetY,
              std::vector<std::pair<int,int>>& path) {
    // 基本情况: 到达目标
    if (x == targetX && y == targetY) {
        path.push_back({x, y});
        return true;
    }

    // 基本情况: 越界或遇到障碍
    if (!isValid(x, y) || isObstacle(x, y) || isVisited(x, y)) {
        return false;
    }
```

```

    markVisited(x, y);
    path.push_back({x, y});

    // 递归情况：尝试四个方向
    if (findPath(x + 1, y, targetX, targetY, path) ||
        findPath(x - 1, y, targetX, targetY, path) ||
        findPath(x, y + 1, targetX, targetY, path) ||
        findPath(x, y - 1, targetX, targetY, path)) {
        return true;
    }

    // 回溯
    path.pop_back();
    return false;
}

```

递归代码通常简洁优雅，但也有缺点：每次递归调用都会占用栈空间，深度过大可能会导致栈溢出；此外，递归的函数调用开销可能影响性能。对于性能敏感的场景，可以考虑将递归改写为迭代。

1.5.6.9. `constexpr` 函数

C++11 引入了 `constexpr` 函数，它可以在编译期求值。如果传入的参数是编译期常量，函数的结果也是编译期常量，可用于数组大小、模板参数等需要常量的场合：

```

constexpr int square(int x) {
    return x * x;
}

int main() {
    constexpr int size = square(10); // 编译期计算, size = 100
    int array[size]; // 合法, size 是编译期常量

    int runtime = getUserInput();
    int result = square(runtime); // 运行时计算, 仍然有效
    return 0;
}

```

`constexpr` 函数有一些限制：在 C++11 中，函数体只能包含一条 `return` 语句；C++14 放宽了这一限制，允许使用局部变量、循环和条件语句。

```

// C++14 及以后
constexpr int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

constexpr int fact5 = factorial(5); // 编译期计算, fact5 = 120

```

`constexpr` 函数在 RoboMaster 开发中可用于计算固定的配置参数、查找表等，将计算从运行时移到编译期可以提高程序启动速度和运行效率。

1.5.6.10. 函数设计原则

良好的函数设计能够显著提高代码质量。以下是一些实践中总结的原则：

函数应当短小精悍，专注于完成单一任务。如果一个函数做了太多事情，应当拆分成多个函数。一个经验法则是：如果函数超过一屏（约 40-50 行），就应当考虑拆分。

函数名应当清晰描述其功能。好的命名使代码自文档化，减少注释的需要。动词开头的名称（如 `calculateDistance`、`isValid`、`sendCommand`）通常比名词更能表达函数的行为。

函数的参数不宜过多。超过三四个参数的函数难以记忆和使用，也往往表明函数承担了过多职责。可以考虑将相关参数组织成结构体，或者拆分函数。

优先使用返回值而非输出参数。返回值的语义更清晰，也更符合函数式编程的风格。现代 C++ 的移动语义使得返回大型对象的开销很小。

避免副作用。纯函数（相同输入总是产生相同输出，且不修改外部状态）更容易理解、测试和调试。当然，某些函数的目的就是产生副作用（如 I/O 函数），但应当明确标识这类函数的行为。

```
// 不好的设计：函数过长，职责不清
void processRobot() {
    // 200 行代码，混合了传感器读取、状态更新、通信、日志...
}

// 好的设计：职责分离，函数短小
void updateRobotState() {
    auto sensorData = readSensors();
    auto newState = computeState(sensorData);
    applyState(newState);
    logStateChange(newState);
}
```

函数是组织代码的基本单元，也是抽象的基本工具。通过将复杂问题分解为一系列函数调用，程序的逻辑结构变得清晰，各部分也可以独立开发和测试。掌握函数的使用是迈向更大规模程序开发的关键一步。下一节将介绍数组和字符串，它们是处理批量数据的基础工具。

1.5.7. 数组与字符串

程序经常需要处理一组相关的数据：传感器的连续读数、检测到的多个目标、机器人的历史轨迹。逐个声明变量来存储这些数据既繁琐又不切实际——如果有一千个数据点，难道要声明一千个变量吗？数组正是为解决这一问题而设计的数据结构，它将多个相同类型的元素存储在连续的内存空间中，通过索引访问各个元素。字符串则是字符数组的特殊形式，用于处理文本数据。本节将介绍 C 风格数组、现代 C++ 的 `std::array` 以及字符串的基本操作。

1.5.7.1. C 风格数组

C 风格数组是最基本的数组形式，它直接在栈上分配一块连续的内存。声明数组时需要指定元素类型和数组大小：

```
int scores[5];           // 声明一个包含 5 个整数的数组
double sensorReadings[100]; // 声明一个包含 100 个双精度浮点数的数组
```

数组大小必须是编译期常量，不能使用运行时才能确定的值（变长数组是 C99 的特性，C++ 标准不支持）。数组元素通过下标访问，下标从 0 开始：

```

int values[5] = {10, 20, 30, 40, 50};

int first = values[0]; // 10, 第一个元素
int third = values[2]; // 30, 第三个元素
int last = values[4]; // 50, 最后一个元素

values[1] = 25; // 修改第二个元素

```

数组可以在声明时初始化。如果初始值的数量少于数组大小，剩余元素被初始化为零；如果提供了初始值但省略数组大小，编译器会根据初始值数量自动确定大小：

```

int a[5] = {1, 2, 3}; // a = {1, 2, 3, 0, 0}
int b[5] = {}; // b = {0, 0, 0, 0, 0}, 全部初始化为 0
int c[] = {1, 2, 3, 4, 5}; // 编译器推断大小为 5

```

数组的一个重要特性是它不进行边界检查。访问超出范围的索引不会产生编译错误，但会导致未定义行为——程序可能崩溃、产生错误结果，或者看似正常运行但实际上破坏了其他内存区域：

```

int arr[5] = {1, 2, 3, 4, 5};
int x = arr[10]; // 未定义行为！访问越界
arr[-1] = 0; // 未定义行为！负索引

```

这种越界错误是 C/C++ 程序中最常见的 bug 来源之一。在 RoboMaster 开发中，数组越界可能导致机器人行为异常甚至失控，因此必须格外小心。

遍历数组通常使用 `for` 循环。由于数组本身不存储大小信息，需要单独跟踪数组长度：

```

const int SIZE = 5;
int values[SIZE] = {10, 20, 30, 40, 50};

// 传统 for 循环
for (int i = 0; i < SIZE; i++) {
    std::cout << values[i] << " ";
}

// 范围 for 循环 (C++11)
for (int v : values) {
    std::cout << v << " ";
}

```

多维数组用于表示表格、矩阵等结构。声明时指定每个维度的大小，访问时提供每个维度的索引：

```

// 3x4 的二维数组 (3 行 4 列)
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

int element = matrix[1][2]; // 第 2 行第 3 列，值为 7

// 遍历二维数组
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        std::cout << matrix[i][j] << " ";
    }
}

```

```
    std::cout << std::endl;
}
```

在内存中，多维数组按行优先（row-major）顺序存储，即同一行的元素在内存中相邻。遍历时按行优先顺序访问可以获得更好的缓存性能。

1.5.7.2. 数组与函数

将数组传递给函数时，数组会退化为指向首元素的指针，数组的大小信息丢失。因此，通常需要额外传递数组的大小：

```
// 数组参数实际上是指针
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

// 等价写法
void printArray(int* arr, int size);

int main() {
    int values[] = {1, 2, 3, 4, 5};
    printArray(values, 5);
    return 0;
}
```

由于传递的是指针，函数内部对数组元素的修改会影响原数组。如果不希望函数修改数组内容，可以使用 `const` 修饰：

```
double average(const int arr[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return static_cast<double>(sum) / size;
}
```

对于多维数组，除第一维外的其他维度大小必须在函数参数中明确指定：

```
void processMatrix(int matrix[][4], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 4; j++) {
            // 处理 matrix[i][j]
        }
    }
}
```

这一限制源于编译器需要知道每行的大小才能正确计算元素地址。

1.5.7.3. std::array

C++11 引入的 `std::array` 是对 C 风格数组的封装，提供了更安全、更方便的接口，同时保持与 C 风格数组相当的性能：

```
#include <array>

std::array<int, 5> values = {10, 20, 30, 40, 50};
```

```

int first = values[0];           // 下标访问，不检查边界
int second = values.at(1);      // at() 访问，检查边界，越界抛出异常

std::cout << "大小: " << values.size() << std::endl; // 5
std::cout << "是否为空: " << values.empty() << std::endl; // false

```

`std::array` 相比 C 风格数组的主要优势包括：

它知道自己的大小。`size()` 成员函数返回元素数量，无需单独维护大小变量。

它提供边界检查选项。`at()` 方法在越界时抛出 `std::out_of_range` 异常，便于调试。

它可以像普通对象一样复制和赋值。C 风格数组不能直接赋值，`std::array` 可以：

```

std::array<int, 3> a = {1, 2, 3};
std::array<int, 3> b = a; // 复制整个数组
b = {4, 5, 6};          // 赋值

```

它可以作为函数返回值。C 风格数组不能从函数返回，`std::array` 可以：

```

std::array<double, 3> getPosition() {
    return {1.0, 2.0, 3.0};
}

```

它与标准库算法和容器兼容。`std::array` 提供迭代器，可以与 `std::sort`、`std::find` 等算法配合使用：

```

std::array<int, 5> arr = {5, 2, 8, 1, 9};
std::sort(arr.begin(), arr.end()); // 排序

auto it = std::find(arr.begin(), arr.end(), 8);
if (it != arr.end()) {
    std::cout << "找到了 8" << std::endl;
}

```

`std::array` 的大小必须是编译期常量，这与 C 风格数组相同。如果需要运行时确定大小的数组，应当使用 `std::vector`（将在后续章节介绍）。

在 RoboMaster 开发中，`std::array` 适合存储固定大小的数据，如三维坐标、四元数、固定数量的电机参数等：

```

std::array<double, 3> position = {0.0, 0.0, 0.0};
std::array<double, 4> quaternion = {1.0, 0.0, 0.0, 0.0};
std::array<int, 4> motorIDs = {1, 2, 3, 4};

```

1.5.7.4. C 风格字符串

在 C 语言中，字符串是以空字符 ('\\0') 结尾的字符数组。C++ 继承了这种表示方式，称为 C 风格字符串：

```

char greeting[] = "Hello"; // 编译器自动添加 '\\0'，实际大小为 6
char manual[] = {'H', 'e', 'l', 'l', 'o', '\\0'}; // 等价写法

```

字符串字面量（如 "Hello"）的类型是 `const char[]`，存储在只读内存区域。尝试修改字符串字面量会导致未定义行为：

```

char* str = "Hello"; // 不推荐，应该用 const char*
str[0] = 'h';        // 未定义行为！修改只读内存

const char* str2 = "Hello"; // 正确的写法

```

C 风格字符串的操作函数定义在 `<cstring>` 头文件中：

```
#include <cstring>

char str1[20] = "Hello";
char str2[] = "World";

size_t len = strlen(str1);           // 字符串长度（不含 '\0'），返回 5
strcpy(str1, str2);                // 复制 str2 到 str1
strcat(str1, "!");
int cmp = strcmp(str1, str2);       // 比较字符串，相等返回 0
const char* pos = strstr(str1, "or"); // 查找子串
```

这些函数存在严重的安全隐患。`strcpy` 和 `strcat` 不检查目标缓冲区大小，容易导致缓冲区溢出——这是最常见的安全漏洞之一。现代 C++ 代码应当尽量避免使用这些函数，改用 `std::string`。

1.5.7.5. `std::string`

`std::string` 是 C++ 标准库提供的字符串类，它自动管理内存、提供丰富的操作接口，并且安全易用：

```
#include <string>

std::string greeting = "Hello";
std::string name = "RoboMaster";
std::string message = greeting + ", " + name + "!"; // 字符串连接

std::cout << message << std::endl; // 输出: Hello, RoboMaster!
std::cout << "长度: " << message.length() << std::endl; // 长度: 18
```

`std::string` 支持多种初始化方式：

```
std::string s1;                      // 空字符串
std::string s2 = "Hello";              // 从字符串字面量初始化
std::string s3("Hello");              // 等价写法
std::string s4(5, 'x');               // "xxxxx", 5 个 'x'
std::string s5 = s2;                  // 复制构造
std::string s6 = s2.substr(0, 3);     // "Hel", 子串
```

访问单个字符可以使用下标运算符或 `at()` 方法：

```
std::string str = "Hello";

char c1 = str[0];      // 'H', 不检查边界
char c2 = str.at(1);   // 'e', 检查边界

str[0] = 'h';          // 修改第一个字符
```

`std::string` 提供了丰富的操作方法：

```
std::string str = "Hello, World!";

// 查找
size_t pos = str.find("World");        // 返回 7
size_t pos2 = str.find("xyz");         // 返回 std::string::npos (未找到)

if (pos != std::string::npos) {
    std::cout << "找到了, 位置: " << pos << std::endl;
```

```

}

// 子串
std::string sub = str.substr(7, 5);      // "World", 从位置 7 开始取 5 个字符

// 替换
str.replace(7, 5, "C++");                // "Hello, C++!"

// 插入和删除
str.insert(7, "Dear ");                  // "Hello, Dear C++!"
str.erase(7, 5);                        // "Hello, C++!"

// 追加
str.append(" is great");                // "Hello, C++! is great"
str += "!";                            // 等价于 append

// 清空和判空
str.clear();                           // 清空字符串
bool isEmpty = str.empty();             // true

```

字符串比较可以直接使用关系运算符，按字典序比较：

```

std::string a = "apple";
std::string b = "banana";

if (a < b) {
    std::cout << a << " 在 " << b << " 之前" << std::endl;
}

if (a == "apple") {
    std::cout << "是苹果" << std::endl;
}

```

与 C 风格字符串的转换有时是必要的，特别是与 C 语言库或系统 API 交互时：

```

std::string cppStr = "Hello";

// std::string 转 C 风格字符串
const char* cStr = cppStr.c_str();

// C 风格字符串转 std::string
const char* source = "World";
std::string newStr = source; // 自动转换

```

1.5.7.6. 字符串与数值转换

在处理配置文件、用户输入或通信协议时，经常需要在字符串和数值之间转换。C++11 提供了一组便捷的转换函数：

```

#include <iostream>

// 数值转字符串
int num = 42;
double pi = 3.14159;
std::string s1 = std::to_string(num); // "42"
std::string s2 = std::to_string(pi); // "3.141590"

// 字符串转数值

```

```

std::string str1 = "123";
std::string str2 = "3.14";
std::string str3 = "42abc";

int i = std::stoi(str1);           // 123
double d = std::stod(str2);       // 3.14
int partial = std::stoi(str3);    // 42, 解析到非数字字符停止

// 其他转换函数
long l = std::stol("1234567890");
float f = std::stof("2.718");

```

这些函数在遇到无法解析的输入时会抛出异常 (`std::invalid_argument` 或 `std::out_of_range`)，使用时应当进行错误处理：

```

std::string input = "not a number";

try {
    int value = std::stoi(input);
} catch (const std::invalid_argument& e) {
    std::cerr << "无效输入: " << e.what() << std::endl;
} catch (const std::out_of_range& e) {
    std::cerr << "数值超出范围: " << e.what() << std::endl;
}

```

1.5.7.7. 字符串流

`<sstream>` 头文件提供的字符串流允许像使用 `cin/cout` 一样操作字符串，这在格式化输出和解析输入时非常有用：

```

#include <sstream>

// 格式化输出到字符串
std::ostringstream oss;
oss << "Position: (" << 1.5 << ", " << 2.5 << ", " << 3.5 << ")";
std::string result = oss.str(); // "Position: (1.5, 2.5, 3.5)"

// 从字符串解析数据
std::string data = "100 200 300";
std::istringstream iss(data);
int x, y, z;
iss >> x >> y >> z; // x=100, y=200, z=300

```

字符串流在 RoboMaster 开发中常用于构建日志消息、解析配置文件或处理串口通信数据：

```

// 构建日志消息
std::ostringstream log;
log << "[" << timestamp << "] Motor " << motorId
    << ": speed=" << speed << " rpm, temp=" << temperature << "°C";
logger.write(log.str());

// 解析串口数据
std::string response = "OK 1234 5678";
std::istringstream parser(response);
std::string status;
int value1, value2;
parser >> status >> value1 >> value2;

```

1.5.7.8. 实践建议

在现代 C++ 开发中，应当优先使用 `std::array` 和 `std::string`，而非 C 风格数组和字符串。它们更安全、更易用，且性能开销可以忽略不计。

C 风格数组和字符串仍有其用武之地：与 C 语言库交互、嵌入式开发中的内存受限场景、或者需要精确控制内存布局时。但即使在这些场景下，也应当尽量将 C 风格代码封装在有限的范围内，对外提供现代 C++ 接口。

对于大小在运行时确定或需要动态增长的数组，应当使用 `std::vector`。它提供了动态数组的功能，是 C++ 中最常用的容器之一，将在后续章节详细介绍。

数组和字符串是处理数据的基础工具。然而，到目前为止我们操作的都是数据的“值”。下一节将介绍指针，它允许我们直接操作数据的“地址”，是理解 C++ 内存模型的关键。

1.5.8. 指针基础

到目前为止，我们操作的都是变量的值——读取它、修改它、传递它的副本。但有时候，我们需要直接操作变量在内存中的位置：让多个变量指向同一块数据、在函数中修改调用者的变量、动态分配内存、构建链表和树等复杂数据结构。指针正是实现这些功能的工具。理解指针是掌握 C++ 的关键一步，它揭示了程序与内存交互的底层机制。

1.5.8.1. 内存地址与指针的概念

计算机内存可以想象成一排连续编号的格子，每个格子存储一个字节的数据，格子的编号就是内存地址。当我们声明一个变量时，编译器会在内存中分配一块空间来存储它的值，这块空间有一个起始地址。

```
int x = 42;
```

假设 `x` 被分配在地址 `0x7ffd5e8c` 处，占用 4 个字节（`int` 类型的典型大小）。变量名 `x` 是这块内存的标签，`42` 是其中存储的值，`0x7ffd5e8c` 是它的地址。

取地址运算符 `&` 用于获取变量的内存地址：

```
int x = 42;
std::cout << "x 的值: " << x << std::endl;
std::cout << "x 的地址: " << &x << std::endl; // 输出类似 0x7ffd5e8c
```

指针是一种特殊的变量，它存储的不是普通数据，而是另一个变量的内存地址。声明指针时，在类型名后加上 `*`：

```
int x = 42;
int* ptr = &x; // ptr 是指向 int 的指针，存储 x 的地址

std::cout << "ptr 的值: " << ptr << std::endl; // x 的地址
std::cout << "ptr 指向的值: " << *ptr << std::endl; // 42
```

这里 `int*` 表示“指向 `int` 的指针”类型。`ptr` 存储的是 `x` 的地址，通过解引用运算符 `*` 可以访问 `ptr` 所指向的内存中的值。

指针声明的语法有几种等价写法，选择哪种主要是风格问题：

```
int* ptr1; // 星号靠近类型，强调 ptr1 的类型是 int*
int *ptr2; // 星号靠近变量名，C 语言传统风格
int * ptr3; // 两边都有空格，较少使用
```

需要注意的是，星号只与紧跟其后的变量名结合。在同一行声明多个变量时容易出错：

```
int* p1, p2; // 注意: p1 是指针, p2 是普通 int!
int *p3, *p4; // p3 和 p4 都是指针
```

为避免混淆, 建议每行只声明一个指针变量。

1.5.8.2. 解引用与指针运算

解引用运算符 `*` 用于访问指针所指向的内存。它既可以读取值, 也可以修改值:

```
int x = 10;
int* ptr = &x;

std::cout << *ptr << std::endl; // 读取: 输出 10

*ptr = 20; // 写入: 修改 ptr 指向的内存
std::cout << x << std::endl; // x 现在是 20
```

通过指针修改数据是指针最重要的用途之一。它使得函数能够修改调用者的变量, 也是实现复杂数据结构的基础。

指针支持算术运算, 但其行为与普通整数不同。指针加 1 不是地址值加 1, 而是移动到下一个元素的位置:

```
int arr[] = {10, 20, 30, 40, 50};
int* ptr = arr; // 指向数组首元素

std::cout << *ptr << std::endl; // 10
std::cout << *(ptr + 1) << std::endl; // 20
std::cout << *(ptr + 2) << std::endl; // 30

ptr++; // ptr 现在指向 arr[1]
std::cout << *ptr << std::endl; // 20
```

指针加 1 实际上使地址值增加了 `sizeof(int)` 字节 (通常是 4)。这种设计使得指针能够方便地遍历数组, 而无需关心元素的具体大小。

两个指向同一数组的指针可以相减, 结果是它们之间的元素个数:

```
int arr[] = {10, 20, 30, 40, 50};
int* p1 = &arr[1];
int* p2 = &arr[4];

std::ptrdiff_t diff = p2 - p1; // 3, 相差 3 个元素
```

1.5.8.3. 指针与数组

数组和指针的关系非常密切。在大多数表达式中, 数组名会自动转换为指向首元素的指针:

```
int arr[] = {10, 20, 30, 40, 50};

int* ptr = arr; // arr 退化为 &arr[0]
std::cout << *ptr << std::endl; // 10
std::cout << ptr[2] << std::endl; // 30, 等价于 *(ptr + 2)
std::cout << arr[2] << std::endl; // 30, 等价于 *(arr + 2)
```

下标运算符 `[]` 实际上是指针算术的语法糖: `arr[i]` 完全等价于 `*(arr + i)`。这也解释了为什么数组下标从 0 开始——`arr[0]` 就是 `*(arr + 0)`, 即首元素本身。

尽管数组名可以像指针一样使用, 但它们并不完全相同。数组名是常量, 不能被重新赋值; 数组名使用 `sizeof` 返回整个数组的大小, 而指针返回指针本身的大小:

```

int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr;

// arr = ptr; // 错误！数组名不能被赋值

std::cout << sizeof(arr) << std::endl; // 20 (5 个 int)
std::cout << sizeof(ptr) << std::endl; // 8 (64 位系统上指针的大小)

```

遍历数组可以使用下标，也可以使用指针：

```

int arr[] = {10, 20, 30, 40, 50};
int size = sizeof(arr) / sizeof(arr[0]);

// 下标方式
for (int i = 0; i < size; i++) {
    std::cout << arr[i] << " ";
}

// 指针方式
for (int* p = arr; p < arr + size; p++) {
    std::cout << *p << " ";
}

// 指针 + 下标混合
int* ptr = arr;
for (int i = 0; i < size; i++) {
    std::cout << ptr[i] << " ";
}

```

这三种方式在效果上完全等价，选择哪种主要看个人偏好和具体场景。

1.5.8.4. 指针与函数参数

在前面的章节中，我们已经了解到 C++ 默认使用值传递——函数接收的是参数的副本，对副本的修改不影响原变量。通过指针传递，函数可以修改调用者的数据：

```

// 值传递：无法修改原变量
void incrementByValue(int n) {
    n++; // 只修改了副本
}

// 指针传递：可以修改原变量
void incrementByPointer(int* ptr) {
    (*ptr)++; // 修改指针指向的内存
}

int main() {
    int x = 10;

    incrementByValue(x);
    std::cout << x << std::endl; // 仍然是 10

    incrementByPointer(&x);
    std::cout << x << std::endl; // 变成 11

    return 0;
}

```

指针参数常用于以下场景：

需要修改调用者的变量时，如交换两个变量的值：

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int x = 10, y = 20;
swap(&x, &y); // x = 20, y = 10
```

需要返回多个值时，可以通过指针参数“返回”额外的结果：

```
// 返回商和余数
void divide(int dividend, int divisor, int* quotient, int* remainder) {
    *quotient = dividend / divisor;
    *remainder = dividend % divisor;
}

int q, r;
divide(17, 5, &q, &r); // q = 3, r = 2
```

传递大型数据结构时，避免复制开销：

```
// 传值：复制整个结构体，开销大
void processData(LargeStruct data);

// 传指针：只传递地址，开销小
void processData(const LargeStruct* data);
```

当函数不需要修改指针指向的数据时，应当使用 `const` 修饰，这既表明了意图，也让编译器帮助检查错误。

1.5.8.5. 空指针与野指针

指针的一个重要特性是它可以不指向任何有效对象。空指针（null pointer）是一个特殊的指针值，表示“不指向任何东西”。在 C++11 之前，通常使用 `NULL` 或 `0` 表示空指针；C++11 引入了 `nullptr` 关键字，它是类型安全的空指针常量：

```
int* ptr1 = nullptr; // C++11 推荐写法
int* ptr2 = NULL; // C 风格，仍然有效
int* ptr3 = 0; // 也有效，但不推荐

if (ptr1 == nullptr) {
    std::cout << "ptr1 是空指针" << std::endl;
}
```

解引用空指针会导致程序崩溃（通常是段错误）。因此，在使用指针之前，通常需要检查它是否为空：

```
void processTarget(Target* target) {
    if (target == nullptr) {
        std::cerr << "错误：目标指针为空" << std::endl;
        return;
    }

    // 安全地使用 target
```

```
    target->track();
}
```

野指针 (dangling pointer) 是指向已释放或无效内存的指针，比空指针更危险。野指针的解引用可能不会立即崩溃，而是产生难以预测的行为：

```
int* createDanglingPointer() {
    int local = 42;
    return &local; // 危险！返回局部变量的地址
}

int main() {
    int* ptr = createDanglingPointer();
    // ptr 现在是野指针，local 已经被销毁
    std::cout << *ptr << std::endl; // 未定义行为！
    return 0;
}
```

避免野指针的原则包括：不返回局部变量的地址；释放内存后立即将指针置为 `nullptr`；使用智能指针（后续章节介绍）自动管理内存。

1.5.8.6. `const` 与指针

`const` 关键字与指针结合使用时，可以限制指针本身或指针所指向的数据。根据 `const` 的位置不同，有三种情况：

指向常量的指针 (pointer to `const`)：不能通过指针修改所指向的数据，但指针本身可以改变指向：

```
int x = 10, y = 20;
const int* ptr = &x; // ptr 指向 const int

// *ptr = 30; // 错误！不能通过 ptr 修改数据
ptr = &y; // 允许，ptr 可以指向其他地址
```

常量指针 (`const` pointer)：指针本身不能改变指向，但可以通过指针修改数据：

```
int x = 10, y = 20;
int* const ptr = &x; // ptr 是 const，指向 int

*ptr = 30; // 允许，可以修改数据
// ptr = &y; // 错误！ptr 不能指向其他地址
```

指向常量的常量指针：两者都不能改变：

```
int x = 10;
const int* const ptr = &x;

// *ptr = 30; // 错误！
// ptr = &y; // 错误！
```

记忆技巧是从右向左读：`const int* ptr` 读作“ptr 是指针，指向 `const int`”；`int* const ptr` 读作“ptr 是 `const`，是指针，指向 `int`”。

在函数参数中，`const` 指针用于表明函数不会修改传入的数据：

```
// 承诺不修改数组内容
double calculateAverage(const int* arr, int size) {
    double sum = 0;
    for (int i = 0; i < size; i++) {
```

```

        sum += arr[i];
        // arr[i] = 0; // 错误! 不能修改
    }
    return sum / size;
}

```

1.5.8.7. 指针与动态内存

到目前为止，我们使用的变量都在栈上分配，它们的生命周期由作用域自动管理。但有时候，我们需要在运行时决定分配多少内存，或者让数据的生命周期超出创建它的函数。这时就需要动态内存分配。

C++ 使用 `new` 运算符在堆上分配内存，使用 `delete` 运算符释放内存：

```

// 分配单个对象
int* ptr = new int;           // 分配一个 int
*ptr = 42;
delete ptr;                  // 释放内存

// 分配并初始化
int* ptr2 = new int(100);     // 分配并初始化为 100
delete ptr2;

// 分配数组
int* arr = new int[10];       // 分配 10 个 int 的数组
for (int i = 0; i < 10; i++) {
    arr[i] = i * 10;
}
delete[] arr;                // 注意：数组用 delete[]

```

动态内存管理是 C++ 中最容易出错的领域之一。常见的问题包括：

内存泄漏：分配了内存但忘记释放，导致内存被持续占用：

```

void memoryLeak() {
    int* ptr = new int[1000];
    // 忘记 delete[], 函数返回后内存泄漏
}

```

重复释放：对同一块内存调用多次 `delete`，导致未定义行为：

```

int* ptr = new int;
delete ptr;
delete ptr; // 错误！重复释放

```

使用已释放的内存：释放后继续使用指针：

```

int* ptr = new int(42);
delete ptr;
std::cout << *ptr << std::endl; // 未定义行为！

```

数组与单对象混淆：用 `delete` 释放 `new[]` 分配的内存，或反过来：

```

int* arr = new int[10];
delete arr; // 错误！应该用 delete[]

```

为了避免这些问题，现代 C++ 推荐使用智能指针（`std::unique_ptr`、`std::shared_ptr`）来管理动态内存，它们会在适当的时机自动释放内存。智能指针将在后续章节详细介绍。

在 RoboMaster 开发中，动态内存分配常见于创建运行时大小确定的缓冲区、管理检测到的目标列表等场景。但由于实时性要求，应当尽量减少动态分配的频率，或在初始化阶段预分配所需内存。

1.5.8.8. 指针的常见用途

指针在 C++ 中有多种重要用途，以下是一些典型场景：

实现数据结构。链表、树、图等结构需要节点之间相互引用，指针是实现这种引用的自然方式：

```
struct ListNode {
    int data;
    ListNode* next;
};
```

多态与动态绑定。通过基类指针调用派生类的方法，是面向对象编程的核心技术（后续章节介绍）：

```
class Robot {
public:
    virtual void move() = 0;
};

void controlRobot(Robot* robot) {
    robot->move(); // 根据实际类型调用对应的实现
}
```

可选参数。指针可以为 `nullptr` 表示“无值”，这在引用无法做到的场景很有用：

```
void processData(const Config* optionalConfig = nullptr) {
    if (optionalConfig != nullptr) {
        // 使用配置
    } else {
        // 使用默认配置
    }
}
```

与 C 语言库交互。许多系统 API 和第三方库使用 C 风格接口，需要通过指针传递数据。

1.5.8.9. 实践建议

指针是强大但危险的工具。以下是一些实践建议：

始终初始化指针。未初始化的指针包含随机地址，解引用会导致不可预测的行为。如果暂时没有有效地址，将指针初始化为 `nullptr`。

使用前检查空指针。特别是对于函数参数和函数返回值，不要假设指针一定有效。

释放后置空。`delete` 后立即将指针置为 `nullptr`，可以避免野指针和重复释放。

```
delete ptr;
ptr = nullptr;
```

优先使用引用。当不需要“可空”语义时，引用比指针更安全。

优先使用智能指针。对于动态内存管理，`std::unique_ptr` 和 `std::shared_ptr` 能够自动处理释放，大大减少内存错误。

理解指针是理解 C++ 内存模型的关键。它解释了为什么函数参数默认是值传递、为什么数组在传递时会丢失大小信息、为什么需要区分栈和堆。下一节将介绍引用，它提供了一种更安全、更简洁的方式来实现指针的部分功能。

1.5.9. 引用

在上一节中，我们学习了指针——一种直接操作内存地址的机制。指针功能强大，但使用起来需要格外小心：需要解引用才能访问数据，需要检查空指针，还要防范野指针。C++ 提供了另一种间接访问数据的方式——引用（reference）。引用可以看作是变量的别名，它提供了指针的部分功能，却更加安全和直观。理解引用与指针的区别，并在恰当的场合选择合适的工具，是写出清晰、安全的 C++ 代码的重要一步。

1.5.9.1. 引用的基本概念

引用是已存在变量的另一个名字。声明引用时，在类型名后加上 `&` 符号，并且必须在声明时初始化：

```
int x = 42;
int& ref = x; // ref 是 x 的引用，即 x 的别名

std::cout << x << std::endl; // 42
std::cout << ref << std::endl; // 42

ref = 100; // 通过引用修改
std::cout << x << std::endl; // 100, x 也变了
```

在这个例子中，`ref` 和 `x` 指向同一块内存。对 `ref` 的任何操作，实际上就是对 `x` 的操作。引用不是一个独立的对象，它只是现有对象的另一个名称。

与指针不同，引用一旦绑定到某个变量，就不能再绑定到其他变量。引用在声明时必须初始化，之后对引用的任何赋值都是在修改被引用对象的值，而非改变引用本身的绑定：

```
int a = 10;
int b = 20;
int& ref = a; // ref 绑定到 a

ref = b; // 这不是让 ref 绑定到 b，而是把 b 的值赋给 a
std::cout << a << std::endl; // 20, a 的值变成了 b 的值
std::cout << &a << std::endl; // a 的地址
std::cout << &ref << std::endl; // 相同的地址，ref 仍然绑定到 a
```

这一特性使得引用比指针更容易理解：一旦建立引用关系，就不会改变。使用引用时也无需像指针那样使用 `*` 解引用或 `&` 取地址，语法上与普通变量完全相同。

1.5.9.2. 引用与指针的对比

引用和指针都提供了间接访问数据的能力，但它们在设计理念和使用方式上有本质区别。理解这些区别有助于在不同场景下做出正确的选择。

首先，引用必须在声明时初始化，且始终绑定到某个有效对象；指针可以不初始化，也可以指向 `nullptr` 表示“不指向任何对象”。这意味着引用更安全——使用引用时不需要检查它是否有效，而使用指针时通常需要进行空指针检查：

```
void processWithPointer(int* ptr) {
    if (ptr == nullptr) { // 必须检查
```

```

        return;
    }
    *ptr = 100;
}

void processWithReference(int& ref) {
    // 无需检查, ref 一定绑定到有效对象
    ref = 100;
}

```

其次, 引用一旦绑定就不能改变, 而指针可以随时指向其他对象。这在需要重新指向不同对象的场景下, 指针更加灵活; 而在希望保持绑定关系不变的场景下, 引用更加安全。

再者, 语法上的差异也很明显。指针需要`*`来解引用、`&`来取地址, 而引用的使用与普通变量无异:

```

int x = 42;

// 使用指针
int* ptr = &x;
*ptr = 100;
std::cout << *ptr << std::endl;

// 使用引用
int& ref = x;
ref = 100;
std::cout << ref << std::endl;

```

引用的语法更简洁, 代码也更容易阅读。特别是在连续的操作中, 不断地写`*ptr`会让代码显得杂乱。

此外, 引用没有“引用的引用”或“引用的数组”这样的概念, 而指针可以有多级指针(如`int**`)和指针数组。在需要复杂数据结构的场合, 指针提供了更大的灵活性。

在实践中, 应当优先考虑使用引用, 除非以下情况需要使用指针: 需要表示“可能不存在”(使用`nullptr`)、需要在运行时改变所指对象、需要进行指针运算、或者需要与 C 语言接口交互。现代 C++ 的设计哲学是: 能用引用就不用指针, 能用智能指针就不用裸指针。

1.5.9.3. 引用作为函数参数

引用最常见的用途是作为函数参数。在前面的章节中, 我们已经见过这种用法, 现在来更深入地理解它的工作原理和优势。

当使用值传递时, 函数接收的是实参的副本, 对形参的修改不会影响实参。而使用引用传递时, 形参是实参的别名, 对形参的修改直接作用于实参:

```

void swapByValue(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    // 交换的只是副本, 对原变量没有影响
}

void swapByReference(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

```

```

    // 直接操作原变量，交换成功
}

int main() {
    int x = 10, y = 20;

    swapByValue(x, y);
    std::cout << x << ", " << y << std::endl; // 10, 20, 没有交换

    swapByReference(x, y);
    std::cout << x << ", " << y << std::endl; // 20, 10, 交换成功

    return 0;
}

```

引用参数的第一个优势是允许函数修改调用者的变量。这在需要返回多个值的场景下特别有用：

```

// 计算同时返回商和余数
void divideWithRemainder(int dividend, int divisor,
                          int& quotient, int& remainder) {
    quotient = dividend / divisor;
    remainder = dividend % divisor;
}

int main() {
    int q, r;
    divideWithRemainder(17, 5, q, r);
    std::cout << "商: " << q << ", 余数: " << r << std::endl; // 商: 3, 余数: 2
    return 0;
}

```

引用参数的第二个优势是避免复制开销。对于大型对象，值传递需要调用拷贝构造函数创建副本，这可能耗费大量时间和内存。通过引用传递，只传递对象的别名，完全消除了复制的开销：

```

// 值传递：每次调用都会复制整个 vector
double calculateAverageByValue(std::vector<double> data) {
    double sum = 0;
    for (double v : data) sum += v;
    return sum / data.size();
}

// 引用传递：不复制，效率高
double calculateAverageByReference(std::vector<double>& data) {
    double sum = 0;
    for (double v : data) sum += v;
    return sum / data.size();
}

```

对于包含成百上千个元素的 `std::vector`，两者的性能差异可能非常显著。在 RoboMaster 开发中，传感器数据、图像帧、目标列表等往往是大型对象，使用引用传递是标准做法。

引用参数与指针参数相比，调用时的语法更加自然。使用指针参数时，调用者需要显式取地址；使用引用参数时，直接传入变量名即可：

```

void updateWithPointer(int* ptr) { *ptr = 100; }
void updateWithReference(int& ref) { ref = 100; }

```

```
int x = 0;
updateWithPointer(&x); // 需要 &
updateWithReference(x); // 直接传变量名
```

然而，这种简洁性也带来一个潜在的问题：从调用代码来看，无法直接判断函数是否会修改参数。`updateWithReference(x)`看起来像是值传递，但实际上 `x` 可能被修改。为了解决这个问题，当函数不需要修改参数时，应当使用常量引用。

1.5.9.4. 常量引用

常量引用（const reference）是指向常量的引用，通过它不能修改被引用的对象。声明常量引用时，在类型前加上 `const`：

```
int x = 42;
const int& cref = x;

std::cout << cref << std::endl; // 42, 可以读取
// cref = 100; // 错误！不能通过常量引用修改
```

常量引用最重要的用途是作为函数参数，表明函数只读取数据而不修改它。这既是一种性能优化（避免复制），也是一种文档说明（告诉调用者参数不会被修改）：

```
// 常量引用参数：承诺不修改 data
double calculateAverage(const std::vector<double>& data) {
    double sum = 0;
    for (double v : data) sum += v;
    return sum / data.size();
    // data.push_back(0); // 错误！不能修改
}
```

使用常量引用后，调用者可以放心地传入自己的数据，知道函数不会对其进行任何修改。编译器也会检查函数体内是否有违反承诺的操作，在编译阶段捕获潜在的 bug。

常量引用还有一个特殊性质：它可以绑定到临时对象（右值）。普通引用只能绑定到变量（左值），但常量引用例外：

```
// int& ref = 42; // 错误！普通引用不能绑定到字面量
const int& cref = 42; // 正确，常量引用可以绑定到临时值

// 这使得以下调用成为可能
void printValue(const int& value) {
    std::cout << value << std::endl;
}

printValue(100); // 可以传入字面量
printValue(x + y); // 可以传入表达式的结果
```

如果函数参数是非常量引用，就不能传入临时值：

```
void modify(int& value) {
    value = 100;
}

// modify(42); // 错误！不能将临时值绑定到非常量引用
```

这个限制是合理的：修改一个即将消失的临时值没有意义，编译器通过这个规则帮助我们避免无意义甚至错误的操作。

在 RoboMaster 开发中，常量引用是函数参数的常用形式。一条简单的经验法则是：对于基本类型（如 `int`、`double`），由于它们很小，直接值传递即可；对于类类型和容器（如 `std::string`、`std::vector`、自定义类），如果函数不需要修改参数，就使用常量引用传递：

```
// 基本类型：值传递
int square(int x) {
    return x * x;
}

// 类类型：常量引用传递
void logMessage(const std::string& message) {
    std::cout << "[LOG] " << message << std::endl;
}

// 容器：常量引用传递
double processReadings(const std::vector<double>& readings) {
    // 处理传感器读数
}

// 需要修改时：非常量引用传递
void normalizeVector(std::vector<double>& vec) {
    double sum = 0;
    for (double v : vec) sum += v;
    for (double& v : vec) v /= sum;
}
```

1.5.9.5. 引用作为返回值

函数不仅可以接受引用作为参数，还可以返回引用。返回引用允许调用者直接访问和修改函数返回的对象，而无需复制。这在操作容器元素时特别有用：

```
class Warehouse {
private:
    std::vector<int> inventory;

public:
    Warehouse() : inventory(10, 0) {}

    // 返回引用，允许外部修改库存
    int& itemAt(size_t index) {
        return inventory[index];
    }

    // 返回常量引用，只允许读取
    const int& itemAt(size_t index) const {
        return inventory[index];
    }
};

int main() {
    Warehouse warehouse;
    warehouse.itemAt(3) = 100; // 直接修改库存
    std::cout << warehouse.itemAt(3) << std::endl; // 100
    return 0;
}
```

标准库中的 `std::vector::operator[]` 和 `std::map::operator[]` 都返回引用，这就是为什么我们可以写 `vec[0] = 10` 这样的代码。

然而，返回引用时必须格外小心：绝不能返回局部变量的引用。局部变量在函数返回后被销毁，返回它的引用会导致未定义行为：

```
// 危险！返回局部变量的引用
int& badFunction() {
    int local = 42;
    return local; // local 即将被销毁，返回的引用是悬空的
}

int main() {
    int& ref = badFunction(); // ref 是悬空引用
    std::cout << ref << std::endl; // 未定义行为！
    return 0;
}
```

安全的做法是只返回以下对象的引用：成员变量、静态变量、通过引用参数传入的对象、或者动态分配的对象。简单的原则是：确保被引用的对象在引用使用期间始终存在。

1.5.9.6. 引用的底层实现

从使用者的角度，引用表现得像是变量的别名，但在编译器的实现中，引用通常是通过指针来实现的。编译器在底层将引用转换为常量指针（指针本身不能改变指向），并自动处理解引用操作：

```
int x = 42;
int& ref = x;
ref = 100;

// 编译器可能将上述代码转换为类似：
int x = 42;
int* const ref_impl = &x; // 常量指针，不能改变指向
*ref_impl = 100;          // 自动解引用
```

这解释了为什么引用在大多数情况下与指针具有相同的性能特征：传递引用参数实际上传递的是地址，返回引用实际上返回的是地址。引用的优势不在于运行效率，而在于语法的简洁性和使用的安全性。

理解这一点也有助于解释引用的一些限制：引用不能为空（因为它必须绑定到有效对象，就像常量指针必须初始化一样）、引用不能重新绑定（因为底层指针是常量）、引用没有自己的地址（对引用取地址得到的是被引用对象的地址）。

1.5.9.7. 实践中的选择

掌握了引用和指针的区别后，在实际编程中应当根据具体需求选择合适的工具。以下是一些指导原则：

当需要表示“可能不存在”的语义时，使用指针。引用必须绑定到有效对象，无法表示空值；而指针可以为 `nullptr`，明确表示“当前没有指向任何对象”：

```
// 指针：可能找不到目标
Target* findTarget(const std::vector<Target>& targets) {
    for (auto& t : targets) {
        if (t.isValid()) return &t;
```

```

    }
    return nullptr; // 没有找到
}

```

当需要在函数中修改调用者的变量时，使用引用。引用语法更简洁，且调用者无需担心空指针问题：

```

void updatePosition(double& x, double& y, double dx, double dy) {
    x += dx;
    y += dy;
}

```

当传递大型对象且不需要修改时，使用常量引用。这既避免了复制开销，又明确了函数的意图：

```

void analyze(const SensorData& data) {
    // 分析数据，不修改
}

```

当需要重新指向不同对象时，使用指针。引用一旦绑定就不能改变，而指针可以：

```

void processTargets(std::vector<Target*>& targets) {
    Target* current = nullptr;
    for (auto& t : targets) {
        current = &t; // 指针可以改变指向
        process(*current);
    }
}

```

当与 C 语言库或需要指针语义的 API 交互时，使用指针。许多系统调用和第三方库采用 C 风格接口，必须使用指针。

在 RoboMaster 开发中，引用广泛用于函数参数传递，而指针常见于与底层硬件接口、动态数据结构、以及可选参数的场景。良好的代码会根据语义需求选择最合适的工具，而不是一味地使用某一种方式。

引用是 C++ 相对于 C 语言的重要改进之一，它使得代码更加安全和易读。结合前面学习的指针知识，现在我们已经掌握了 C++ 中两种间接访问数据的机制。接下来，我们将把注意力从单个变量转向数据的组织方式，学习如何使用结构体将相关的数据组合在一起，这将为后续的面向对象编程打下基础。

1.5.10. 结构体

到目前为止，我们处理的数据都是单一类型的：一个整数表示速度，一个浮点数表示角度，一个数组存储一系列读数。然而，现实世界中的实体往往具有多个不同类型的属性。以 RoboMaster 比赛中的目标为例，一个目标可能包含位置坐标（三个浮点数）、颜色（一个枚举值）、置信度（一个浮点数）、时间戳（一个整数）等信息。如果用独立的变量来表示这些属性，代码会变得零散且难以管理；如果用数组，又无法存储不同类型的数据。结构体（struct）正是为解决这一问题而设计的——它允许我们将多个相关的数据组合成一个自定义的复合类型。

1.5.10.1. 结构体的定义

结构体使用 `struct` 关键字定义，在花括号内列出各个成员变量（也称为字段）：

```

struct Target {
    double x;
}

```

```

    double y;
    double z;
    int color;
    double confidence;
    long long timestamp;
};


```

这段代码定义了一个名为 `Target` 的新类型。`Target` 类型的变量将包含六个成员：三个表示空间坐标的 `double`、一个表示颜色的 `int`、一个表示置信度的 `double`、以及一个表示时间戳的 `long long`。注意结构体定义末尾的分号是必需的，遗漏它是常见的语法错误。

定义好结构体类型后，就可以像使用内置类型一样声明该类型的变量：

```

Target enemy;           // 声明一个 Target 类型的变量
Target detectedTargets[10]; // Target 类型的数组

```

结构体将相关数据封装在一起，使代码的意图更加清晰。与使用六个独立变量相比，一个 `Target` 变量明确表达了“这些数据属于同一个目标”的语义关系。

1.5.10.2. 成员访问与初始化

访问结构体的成员使用点运算符（`.`）。通过点运算符，可以读取或修改结构体变量的各个字段：

```

Target enemy;
enemy.x = 1000.0;
enemy.y = 500.0;
enemy.z = 200.0;
enemy.color = 1; // 假设 1 表示红色
enemy.confidence = 0.95;
enemy.timestamp = 1703491200000;

std::cout << "目标位置: (" << enemy.x << ", " << enemy.y << ", " << enemy.z <<
")" << std::endl;
std::cout << "置信度: " << enemy.confidence << std::endl;

```

逐个赋值的方式有些繁琐，C++ 提供了多种更简洁的初始化语法。最直接的是使用花括号初始化列表，按照成员声明的顺序提供初始值：

```
Target enemy = {1000.0, 500.0, 200.0, 1, 0.95, 1703491200000};
```

C++11 引入了更灵活的初始化方式，可以省略等号，也可以使用指定初始化器（C++20）明确指定各字段的值：

```
// C++11 风格
Target enemy1{1000.0, 500.0, 200.0, 1, 0.95, 1703491200000};
```

```
// C++20 指定初始化器，更加清晰
Target enemy2{.x = 1000.0, .y = 500.0, .z = 200.0,
               .color = 1, .confidence = 0.95, .timestamp = 1703491200000};
```

如果初始化列表提供的值少于成员数量，剩余成员将被零初始化。如果使用空的花括号 {}，所有成员都被零初始化：

```
Target partial = {100.0, 200.0}; // x=100, y=200, 其余为 0
Target zeroed = {};             // 所有成员为 0
```

结构体也可以在定义时为成员指定默认值，这是 C++11 引入的特性：

```

struct Target {
    double x = 0.0;
    double y = 0.0;
    double z = 0.0;
    int color = 0;
    double confidence = 0.0;
    long long timestamp = 0;
};

Target t; // 所有成员都使用默认值

```

默认值使得创建结构体变量时无需显式初始化每个成员，特别适合那些有合理默认状态的类型。

1.5.10.3. 结构体与函数

结构体可以作为函数的参数和返回值，这使得函数能够处理复合数据而无需传递大量独立参数。

将结构体作为参数传递时，默认是值传递——函数接收的是结构体的副本：

```

void printTarget(Target t) {
    std::cout << "位置: (" << t.x << ", " << t.y << ", " << t.z << ")"
<< std::endl;
    std::cout << "置信度: " << t.confidence << std::endl;
}

Target enemy = {1000.0, 500.0, 200.0, 1, 0.95, 1703491200000};
printTarget(enemy); // 传递副本

```

然而，对于较大的结构体，值传递会产生复制开销。更高效的做法是使用引用传递。如果函数不需要修改结构体，应当使用常量引用：

```

// 常量引用：避免复制，且不会修改原数据
void printTarget(const Target& t) {
    std::cout << "位置: (" << t.x << ", " << t.y << ", " << t.z << ")"
<< std::endl;
}

// 引用：允许修改原数据
void updatePosition(Target& t, double dx, double dy, double dz) {
    t.x += dx;
    t.y += dy;
    t.z += dz;
}

```

函数也可以返回结构体。这是从函数返回多个相关值的自然方式：

```

Target createTarget(double x, double y, double z, int color) {
    Target t;
    t.x = x;
    t.y = y;
    t.z = z;
    t.color = color;
    t.confidence = 1.0;
    t.timestamp = getCurrentTime();
    return t;
}

```

```
Target newTarget = createTarget(500.0, 300.0, 100.0, 1);
```

现代 C++ 编译器会对返回值进行优化（返回值优化，RVO），通常不会产生额外的复制开销，因此不必担心返回结构体的效率问题。

比较一下使用结构体前后的函数签名，可以明显感受到结构体带来的清晰度提升：

```
// 不使用结构体：参数众多，容易混淆顺序  
void processTarget(double x, double y, double z, int color,  
                    double confidence, long long timestamp);  
  
// 使用结构体：意图清晰，不易出错  
void processTarget(const Target& target);
```

1.5.10.4. 结构体与指针

当通过指针访问结构体成员时，需要先解引用指针，再使用点运算符。由于运算符优先级的关系，必须使用括号：

```
Target enemy = {1000.0, 500.0, 200.0, 1, 0.95, 0};  
Target* ptr = &enemy;  
  
// 方式一：先解引用，再访问成员  
(*ptr).x = 1500.0;  
  
// 方式二：使用箭头运算符（更常用）  
ptr->x = 600.0;
```

箭头运算符 `->` 是 `(*ptr).` 的简写形式，在实际代码中更为常见。它使得通过指针访问成员的语法更加简洁：

```
void updateTarget(Target* t) {  
    if (t == nullptr) return;  
  
    t->x += 10.0;  
    t->y += 10.0;  
    t->confidence *= 0.99; // 置信度衰减  
}
```

在 RoboMaster 开发中，指向结构体的指针常用于动态分配的对象、链表节点、以及需要可选参数的场景。

1.5.10.5. 结构体的嵌套与组合

结构体的成员可以是另一个结构体，这使得我们能够构建层次化的数据结构。例如，可以先定义一个表示三维坐标的结构体，再在目标结构体中使用它：

```
struct Point3D {  
    double x;  
    double y;  
    double z;  
};  
  
struct Target {  
    Point3D position;  
    Point3D velocity;  
    int color;  
    double confidence;
```

```
    long long timestamp;  
};
```

访问嵌套成员时，连续使用点运算符：

```
Target enemy;  
enemy.position.x = 1000.0;  
enemy.position.y = 500.0;  
enemy.position.z = 200.0;  
enemy.velocity.x = 10.0;  
enemy.velocity.y = -5.0;  
enemy.velocity.z = 0.0;  
  
std::cout << "位置: (" << enemy.position.x << ", "  
        << enemy.position.y << ", " << enemy.position.z << ")" << std::endl;
```

初始化嵌套结构体时，使用嵌套的花括号：

```
Target enemy = {  
    {1000.0, 500.0, 200.0}, // position  
    {10.0, -5.0, 0.0}, // velocity  
    1, // color  
    0.95, // confidence  
    1703491200000 // timestamp  
};
```

这种组合方式使得数据结构更加模块化。`Point3D`可以在多处复用，而不必在每个需要三维坐标的地方重复定义三个浮点数。

结构体还可以包含数组作为成员，这在表示固定大小的数据集合时很有用：

```
struct RobotState {  
    double jointAngles[6]; // 六个关节的角度  
    double motorCurrents[4]; // 四个电机的电流  
    bool sensorStatus[8]; // 八个传感器的状态  
    long long timestamp;  
};
```

1.5.10.6. 结构体的比较与赋值

结构体支持整体赋值——将一个结构体变量的所有成员复制到另一个同类型的变量：

```
Target t1 = {1000.0, 500.0, 200.0, 1, 0.95, 0};  
Target t2;  
  
t2 = t1; // 复制所有成员  
  
t2.x = 2000.0; // 修改 t2 不影响 t1  
std::cout << t1.x << std::endl; // 仍然是 1000.0
```

这种赋值是成员逐一复制 (memberwise copy)，对于包含指针的结构体需要特别注意——复制的是指针值而非指针指向的数据，这可能导致两个结构体共享同一块动态内存（浅拷贝问题，将在后续章节详细讨论）。

然而，结构体默认不支持使用 `==` 或 `!=` 进行比较。如果需要比较两个结构体是否相等，必须逐个比较成员，或者自定义比较运算符（后续章节介绍运算符重载）：

```
// 不能直接比较  
// if (t1 == t2) { } // 错误!
```

```
// 需要逐个成员比较
bool targetsEqual(const Target& a, const Target& b) {
    return a.x == b.x && a.y == b.y && a.z == b.z &&
           a.color == b.color && a.confidence == b.confidence &&
           a.timestamp == b.timestamp;
}
```

C++20 引入了默认比较运算符的生成机制，可以使用 `= default` 让编译器自动生成比较函数，但这是较新的特性。

1.5.10.7. 结构体的内存布局

理解结构体在内存中的布局有助于优化性能和与底层系统交互。结构体的成员在内存中按声明顺序排列，但编译器可能会在成员之间插入填充字节（padding）以满足对齐要求。

大多数处理器要求特定类型的数据存储在特定对齐边界上。例如，4 字节的 `int` 通常需要存储在 4 的倍数的地址上，8 字节的 `double` 需要存储在 8 的倍数的地址上。为满足这些要求，编译器会在必要时插入填充：

```
struct Example1 {
    char a;        // 1 字节
    int b;         // 4 字节, 但需要 4 字节对齐
    char c;        // 1 字节
};

// 内存布局可能是:
// a (1) + padding (3) + b (4) + c (1) + padding (3) = 12 字节
std::cout << sizeof(Example1) << std::endl; // 可能输出 12, 而非 6
```

通过调整成员顺序，可以减少填充，节省内存：

```
struct Example2 {
    int b;         // 4 字节
    char a;        // 1 字节
    char c;        // 1 字节
};

// 内存布局: b (4) + a (1) + c (1) + padding (2) = 8 字节
std::cout << sizeof(Example2) << std::endl; // 可能输出 8
```

在 RoboMaster 开发中，当结构体用于与下位机通信时，内存布局尤为重要。通信双方必须对数据格式有一致的理解，否则会导致数据解析错误。可以使用编译器特定的指令（如 `#pragma pack`）来控制对齐方式，或者使用定宽类型（如 `int32_t`）确保成员大小一致。

1.5.10.8. 结构体的局限性

结构体为组织数据提供了有力的工具，但它也有一些局限性。

首先，结构体的所有成员默认都是公开的——任何代码都可以直接访问和修改结构体的任何成员。这种开放性在小型程序中不成问题，但在大型项目中可能导致数据被意外修改，难以追踪 bug 的来源：

```
Target t;
t.confidence = 2.0; // 置信度应该在 0-1 之间, 但没有任何检查
t.x = -99999.0; // 可能是无效坐标, 但无法阻止
```

其次，结构体只是数据的集合，它不包含操作这些数据的行为。与数据相关的操作必须定义为独立的函数，数据和操作是分离的：

```
// 数据
struct Target {
    double x, y, z;
    double confidence;
};

// 操作数据的函数（与结构体分离）
double distanceToTarget(const Target& t) {
    return std::sqrt(t.x * t.x + t.y * t.y + t.z * t.z);
}

void updateConfidence(Target& t, double factor) {
    t.confidence *= factor;
}
```

这种分离在逻辑上不够自然——距离计算和置信度更新明显是属于“目标”这个概念的操作，却不得不定义在结构体外部。随着项目规模增长，相关的函数可能散落在各处，难以管理。

再者，结构体缺乏对初始化过程的控制。虽然可以为成员指定默认值，但无法在创建对象时执行复杂的初始化逻辑，如参数验证、资源分配、日志记录等。

这些局限性正是 C++ 引入“类”(class) 的动机。类在结构体的基础上增加了访问控制（可以将某些成员设为私有）、成员函数（将数据和操作封装在一起）、构造函数和析构函数（控制对象的创建和销毁过程）等特性。实际上，在 C++ 中，**struct** 和 **class** 的唯一区别是默认的访问权限——**struct** 默认公开，**class** 默认私有。结构体可以看作是类的特例。

理解结构体是迈向面向对象编程的重要一步。结构体引入了自定义类型的概念，让我们能够按照问题领域的方式组织数据。下一节将正式介绍类与对象，学习如何将数据和操作封装在一起，构建更加健壮和可维护的代码。

1.5.11. 类与对象

上一节我们学习了结构体，它允许将多个相关数据组合成一个自定义类型。然而，结构体存在一些局限：成员完全暴露，任何代码都可以随意修改；数据和操作分离，相关的函数散落在各处。这些问题在小型程序中尚可接受，但随着项目规模增长，代码会变得难以维护。C++ 的类(class) 正是为解决这些问题而设计的。类将数据和操作封装在一起，并提供访问控制机制，是面向对象编程的核心概念。掌握类的使用，标志着从“面向过程”思维向“面向对象”思维的转变。

1.5.11.1. 从结构体到类

让我们从一个具体的例子开始。假设要表示 RoboMaster 比赛中的电机，使用结构体可能是这样：

```
struct Motor {
    int id;
    double speed;           // 当前转速 RPM
    double targetSpeed;   // 目标转速
    double temperature;  // 温度
};

// 操作电机的函数
```

```

void setTargetSpeed(Motor& m, double speed) {
    m.targetSpeed = speed;
}

double getSpeedError(const Motor& m) {
    return m.targetSpeed - m.speed;
}

bool isOverheated(const Motor& m) {
    return m.temperature > 80.0;
}

```

这种写法的问题在于：Motor 只是数据的容器，而操作它的函数是独立存在的。从逻辑上讲，`setTargetSpeed`、`getSpeedError`、`isOverheated` 都是“电机”这个概念的一部分，但代码结构并未体现这种关联。此外，任何代码都可以直接修改 `speed` 或 `temperature`，即使这些值应该只由硬件反馈更新。

使用类可以将数据和操作封装在一起：

```

class Motor {
public:
    void setTargetSpeed(double speed) {
        targetSpeed = speed;
    }

    double getSpeedError() const {
        return targetSpeed - speed;
    }

    bool isOverheated() const {
        return temperature > 80.0;
    }

private:
    int id;
    double speed;
    double targetSpeed;
    double temperature;
};

```

现在，`setTargetSpeed`、`getSpeedError`、`isOverheated` 成为了 `Motor` 类的成员函数，它们与数据紧密绑定。`private` 关键字则保护了内部数据，外部代码无法直接访问 `speed` 或 `temperature`。这就是封装——将数据和操作包装在一起，并控制外部的访问权限。

1.5.11.2. 类的定义

类使用 `class` 关键字定义，基本结构如下：

```

class ClassName {
public:
    // 公开成员：外部可以访问

private:
    // 私有成员：只有类内部可以访问

protected:

```

```
// 受保护成员：类内部和派生类可以访问
};
```

与结构体类似，类定义末尾需要分号。`public`、`private`、`protected` 是访问说明符（access specifier），用于控制成员的可见性。一个类定义中可以有多个访问说明符，它们的作用范围从当前位置延续到下一个访问说明符或类定义结束。

类的成员包括成员变量（也称为数据成员或属性）和成员函数（也称为方法）。成员变量存储对象的状态，成员函数定义对象的行为：

```
class Target {
public:
    // 成员函数
    void setPosition(double newX, double newY, double newZ) {
        x = newX;
        y = newY;
        z = newZ;
    }

    double distanceFromOrigin() const {
        return std::sqrt(x * x + y * y + z * z);
    }

    void print() const {
        std::cout << "Target at (" << x << ", " << y << ", " << z << ")"
        << std::endl;
    }
};

private:
    // 成员变量
    double x = 0.0;
    double y = 0.0;
    double z = 0.0;
    double confidence = 0.0;
};
```

定义好类之后，可以创建该类的对象（也称为实例）。对象是类的具体化——类是蓝图，对象是按照蓝图建造的房子：

```
Target t1;           // 创建一个 Target 对象
Target t2;           // 创建另一个 Target 对象
Target targets[10];   // Target 对象数组

t1.setPosition(100.0, 200.0, 50.0);
t2.setPosition(300.0, 400.0, 100.0);

std::cout << "t1 距原点: " << t1.distanceFromOrigin() << std::endl;
std::cout << "t2 距原点: " << t2.distanceFromOrigin() << std::endl;
```

每个对象都有自己独立的成员变量副本。`t1` 的位置与 `t2` 的位置互不影响，修改一个对象不会改变另一个对象的状态。

1.5.11.3. 成员函数

成员函数是定义在类内部的函数，它可以直接访问类的所有成员（包括私有成员），无需通过参数传递。调用成员函数时使用点运算符，语法与访问成员变量相同：

```

Target t;
t.setPosition(100.0, 200.0, 50.0); // 调用成员函数
double dist = t.distanceFromOrigin();
t.print();

```

成员函数可以在类定义内部直接实现（如上面的例子），也可以在类定义中只声明，然后在类外部实现。后者在大型项目中更为常见，它将接口（头文件中的类定义）与实现（源文件中的函数体）分离：

```

// Target.h - 头文件
class Target {
public:
    void setPosition(double newX, double newY, double newZ);
    double distanceFromOrigin() const;
    void print() const;

private:
    double x = 0.0;
    double y = 0.0;
    double z = 0.0;
};

// Target.cpp - 源文件
#include "Target.h"
#include <cmath>
#include <iostream>

void Target::setPosition(double newX, double newY, double newZ) {
    x = newX;
    y = newY;
    z = newZ;
}

double Target::distanceFromOrigin() const {
    return std::sqrt(x * x + y * y + z * z);
}

void Target::print() const {
    std::cout << "Target at (" << x << ", " << y << ", " << z << ")"
    << std::endl;
}

```

在类外部定义成员函数时，需要使用作用域解析运算符 `::` 指明函数属于哪个类。`Target::setPosition` 表示“`Target` 类的 `setPosition` 函数”。

注意某些成员函数声明末尾的 `const` 关键字。这表示该函数不会修改对象的状态——它是一个“只读”操作。常量成员函数可以被常量对象调用，而非常量成员函数则不行：

```

void analyzeTarget(const Target& t) {
    t.print();           // 正确, print() 是 const 成员函数
    double d = t.distanceFromOrigin(); // 正确, 也是 const
    // t.setPosition(0, 0, 0); // 错误! setPosition() 不是 const, 不能通过 const
    // 引用调用
}

```

将不修改对象状态的成员函数声明为 `const` 是良好的编程习惯。它既是对函数行为的文档说明，也让编译器帮助检查意外的修改操作。

1.5.11.4. 访问控制

访问控制是类与结构体的关键区别之一。C++ 提供三个访问级别：

`public` (公开) 成员可以在任何地方访问。类的公开成员构成了它的接口——外部代码通过公开成员与对象交互。通常，成员函数是公开的，它们定义了对象能够执行的操作。

`private` (私有) 成员只能在类的内部访问，外部代码（包括其他类）无法直接访问。成员变量通常是私有的，这样可以保护数据不被随意修改，确保对象始终处于有效状态。

`protected` (受保护) 成员介于两者之间：类内部和派生类（子类）可以访问，但其他代码不能。这在继承体系中有用，将在后续章节详细讨论。

```
class BankAccount {
public:
    // 公开接口
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            transactionCount++;
        }
    }

    bool withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            transactionCount++;
            return true;
        }
        return false;
    }

    double getBalance() const {
        return balance;
    }
};

private:
    // 私有数据
    double balance = 0.0;
    int transactionCount = 0;
    std::string accountNumber;
};
```

在这个例子中，`balance` 是私有的，外部代码不能直接修改它。存款和取款必须通过 `deposit` 和 `withdraw` 函数进行，这些函数可以包含验证逻辑（如检查金额是否为正、余额是否充足）。这种设计确保了账户余额不会出现非法状态（如负数余额）。

```
BankAccount account;
account.deposit(1000.0);      // 正确，通过公开接口操作
account.withdraw(500.0);      // 正确
std::cout << account.getBalance() << std::endl; // 500.0

// account.balance = 1000000.0; // 错误！balance 是私有的
// account.transactionCount = 0; // 错误！
```

将数据设为私有、通过公开函数访问的模式称为“封装”。封装的好处不仅在于保护数据，还在于隔离变化。如果将来需要修改 `balance` 的存储方式（比如从 `double` 改为使用更精确的货币类型），只需要修改类的内部实现，外部代码无需任何改动，因为它们只依赖于公开接口。

有时需要提供对私有成员的读取或写入能力，这通常通过 `getter` 和 `setter` 函数实现：

```
class Motor {
public:
    // Getter: 获取私有成员的值
    double getSpeed() const { return speed; }
    double getTemperature() const { return temperature; }
    int getId() const { return id; }

    // Setter: 设置私有成员的值（可以包含验证逻辑）
    void setId(int newId) {
        if (newId > 0) {
            id = newId;
        }
    }

    void setTargetSpeed(double target) {
        // 限制目标转速在合理范围内
        if (target < -maxSpeed) target = -maxSpeed;
        if (target > maxSpeed) target = maxSpeed;
        targetSpeed = target;
    }

private:
    int id = 0;
    double speed = 0.0;
    double targetSpeed = 0.0;
    double temperature = 0.0;
    static constexpr double maxSpeed = 10000.0;
};
```

`getter` 和 `setter` 看似增加了代码量，但它们提供了控制点。`getter` 可以在返回前进行计算或格式化，`setter` 可以验证输入、记录日志、触发其他操作。即使当前不需要这些功能，保留这种结构也为将来的扩展留出了空间。

1.5.11.5. struct 与 class 的区别

在 C++ 中，`struct` 和 `class` 几乎是相同的，唯一的区别是默认访问级别。`struct` 的成员默认是 `public`，而 `class` 的成员默认是 `private`：

```
struct Point {
    double x; // 默认 public
    double y;
};

class Vector {
    double x; // 默认 private
    double y;
};
```

因此，以下两个定义是等价的：

```

struct A {
    private:
        int data;
    public:
        void func();
};

class B {
    int data; // 默认 private
    public:
        void func();
};

```

由于这种区别只是默认值不同，技术上可以用 `struct` 实现任何 `class` 能实现的功能。然而，惯例上两者有不同的使用场景：`struct` 通常用于简单的数据聚合，成员都是公开的，很少或没有成员函数；`class` 用于需要封装的情况，有私有数据和公开接口。遵循这一惯例可以让代码的意图更加清晰。

```

// 使用 struct: 简单数据聚合，全部公开
struct Point3D {
    double x, y, z;
};

// 使用 class: 有封装需求，私有数据 + 公开接口
class Robot {
    public:
        void move(double dx, double dy);
        void rotate(double angle);
        Point3D getPosition() const;

    private:
        Point3D position;
        double heading;
        // ...
};

```

1.5.11.6. this 指针

在成员函数内部，如何访问调用该函数的对象本身？C++ 提供了 `this` 指针来解决这个问题。`this` 是一个隐含的指针，指向调用成员函数的那个对象。

当我们写 `t.print()` 时，实际上编译器将 `&t` 作为隐含参数传递给 `print` 函数。在 `print` 内部，`this` 就指向 `t`。通常情况下，访问成员变量时不需要显式使用 `this`——直接写 `x` 等价于 `this->x`。但在某些情况下，显式使用 `this` 是必要或有益的。

第一种情况是当成员变量与参数同名时。如果不使用 `this`，参数名会遮蔽成员变量名：

```

class Rectangle {
    public:
        void setDimensions(double width, double height) {
            // 参数 width 遮蔽了成员变量 width
            // 直接写 width = width 没有意义
            this->width = width;
            this->height = height;
        }
};

```

```

private:
    double width;
    double height;
};

```

一种避免命名冲突的做法是给成员变量加前缀（如 `m_width`）或后缀（如 `width_`），这样就不需要 `this`：

```

class Rectangle {
public:
    void setDimensions(double width, double height) {
        m_width = width;
        m_height = height;
    }

private:
    double m_width;
    double m_height;
};

```

第二种情况是需要返回对象自身的引用，这在实现链式调用时很常见：

```

class QueryBuilder {
public:
    QueryBuilder& select(const std::string& columns) {
        query += "SELECT " + columns + " ";
        return *this; // 返回对象自身的引用
    }

    QueryBuilder& from(const std::string& table) {
        query += "FROM " + table + " ";
        return *this;
    }

    QueryBuilder& where(const std::string& condition) {
        query += "WHERE " + condition + " ";
        return *this;
    }

    std::string build() const {
        return query;
    }

private:
    std::string query;
};

// 链式调用
QueryBuilder builder;
std::string sql = builder.select("*")
    .from("targets")
    .where("confidence > 0.9")
    .build();

```

每个成员函数返回 `*this`（对象自身的引用），使得可以连续调用多个函数。这种模式在许多库中都有应用，如流操作符 `cout << a << b << c` 就是链式调用的例子。

第三种情况是需要将对象自身传递给其他函数：

```

class Observer;

class Subject {
public:
    void registerObserver(Observer* obs) {
        observers.push_back(obs);
    }

    void notifyObservers();
}

private:
    std::vector<Observer*> observers;
};

class Observer {
public:
    void observe(Subject* subject) {
        subject->registerObserver(this); // 将自身注册到被观察对象
    }

    virtual void onNotify() = 0;
};

```

理解 `this` 指针有助于理解成员函数的工作原理。从底层看，成员函数与普通函数的主要区别就是隐含的 `this` 参数。`obj.func(arg)` 在概念上类似于 `func(&obj, arg)`，只不过 C++ 的语法将这一细节隐藏了起来。

1.5.11.7. 一个完整的例子

让我们用一个完整的例子来综合运用本节学习的概念。以下是一个简化的 PID 控制器类，它在 RoboMaster 开发中非常常见：

```

#include <algorithm>

class PIDController {
public:
    // 设置 PID 参数
    void setGains(double kp, double ki, double kd) {
        this->kp = kp;
        this->ki = ki;
        this->kd = kd;
    }

    // 设置输出限幅
    void setOutputLimits(double minOutput, double maxOutput) {
        this->minOutput = minOutput;
        this->maxOutput = maxOutput;
    }

    // 设置积分限幅（防止积分饱和）
    void setIntegralLimits(double minIntegral, double maxIntegral) {
        this->minIntegral = minIntegral;
        this->maxIntegral = maxIntegral;
    }

    // 计算控制输出
}

```

```

double compute(double setpoint, double measurement, double dt) {
    double error = setpoint - measurement;

    // 比例项
    double pTerm = kp * error;

    // 积分项
    integral += error * dt;
    integral = std::clamp(integral, minIntegral, maxIntegral);
    double iTerm = ki * integral;

    // 微分项
    double derivative = (error - lastError) / dt;
    double dTerm = kd * derivative;

    lastError = error;

    // 计算总输出并限幅
    double output = pTerm + iTerm + dTerm;
    return std::clamp(output, minOutput, maxOutput);
}

// 重置控制器状态
void reset() {
    integral = 0.0;
    lastError = 0.0;
}

// Getter 函数
double getKp() const { return kp; }
double getKi() const { return ki; }
double getKd() const { return kd; }
double getIntegral() const { return integral; }

private:
    // PID 增益
    double kp = 0.0;
    double ki = 0.0;
    double kd = 0.0;

    // 输出限幅
    double minOutput = -1e9;
    double maxOutput = 1e9;

    // 积分限幅
    double minIntegral = -1e9;
    double maxIntegral = 1e9;

    // 状态变量
    double integral = 0.0;
    double lastError = 0.0;
};

```

这个类展示了封装的价值：`integral` 和 `lastError` 是控制器的内部状态，外部代码不应该直接修改它们，否则会破坏控制器的正确行为。通过将它们设为私有，我们确保状态只能通过 `compute` 和 `reset` 函数正确地更新。

使用这个类：

```
int main() {
    PIDController speedController;
    speedController.setGains(1.5, 0.1, 0.05);
    speedController.setOutputLimits(-1000.0, 1000.0);
    speedController.setIntegralLimits(-500.0, 500.0);

    double targetSpeed = 3000.0; // 目标转速
    double currentSpeed = 0.0; // 当前转速
    double dt = 0.001; // 控制周期 1ms

    for (int i = 0; i < 1000; i++) {
        double output = speedController.compute(targetSpeed, currentSpeed, dt);

        // 简化的电机模型：输出直接影响速度变化
        currentSpeed += output * 0.01;

        if (i % 100 == 0) {
            std::cout << "Step " << i << ": speed = " << currentSpeed
            << std::endl;
        }
    }

    return 0;
}
```

1.5.11.8. 类设计的基本原则

设计良好的类应当遵循一些基本原则。

首先是单一职责原则：一个类应该只负责一件事情。如果一个类承担了过多职责，它会变得庞大而难以维护。当发现类变得过于复杂时，应当考虑将其拆分为多个更小、更专注的类。

其次是信息隐藏：尽可能将成员设为私有，只暴露必要的接口。公开的成员越少，类与外部代码的耦合就越松，将来修改内部实现时需要调整的地方也越少。

再者是接口设计：公开接口应当简洁、直观、难以误用。函数名应当清晰表达其功能，参数顺序应当符合直觉，异常情况应当有合理的处理方式。

最后是保持一致性：类的成员函数在命名风格、参数顺序、返回值约定等方面应当保持一致。一致的风格使得类更容易学习和使用。

类是 C++ 面向对象编程的基石。通过封装，我们将数据和操作绑定在一起，创建出具有清晰边界和明确职责的代码模块。然而，目前我们创建的对象都使用默认的初始化方式，还没有办法在创建对象时执行自定义的初始化逻辑。下一节将介绍构造函数和析构函数，它们控制着对象的“生”与“死”——如何创建和如何销毁。

1.5.12. 构造函数与析构函数

在上一节中，我们学习了如何定义类和创建对象。但仔细观察会发现，我们创建对象时要么依赖成员变量的默认值，要么在创建后逐一调用 setter 函数进行初始化。这种方式不仅繁琐，还容易遗漏某些必要的初始化步骤，导致对象处于不完整或无效的状态。C++ 提供了构造函数 (constructor) 来解决这一问题——它是对象创建时自动调用的特殊函数，负责将对象初始化到有效状态。与之对应的是析构函数 (destructor)，它在对象销毁时自动调用，负责清理对象占用的资源。理解构造函数和析构函数，就是理解对象的生命周期——从诞生到消亡的完整过程。

1.5.12.1. 对象的生命周期

每个对象都有其生命周期：创建、使用、销毁。对象的存储位置决定了它的生命周期特征。

栈上的对象（局部变量）在进入作用域时创建，离开作用域时销毁。它们的生命周期由编译器自动管理：

```
void processTarget() {
    Target t; // t 在这里被创建
    t.setPosition(100, 200, 50);
    // 使用 t...
} // t 在这里被销毁，离开作用域

int main() {
    processTarget();
    // 此时 t 已经不存在了
    return 0;
}
```

堆上的对象（通过 new 创建）在调用 new 时创建，调用 delete 时销毁。程序员必须手动管理它们的生命周期：

```
void dynamicExample() {
    Target* ptr = new Target(); // 在堆上创建对象
    ptr->setPosition(100, 200, 50);
    // 使用 ptr...
    delete ptr; // 手动销毁对象
}
```

全局对象和静态对象在程序启动时创建，在程序结束时销毁。

无论对象在哪里创建，构造函数都会在创建时被调用，析构函数都会在销毁时被调用。这种自动调用机制是 C++ 资源管理的基础。

1.5.12.2. 默认构造函数

构造函数是一种特殊的成员函数，它的名称与类名相同，没有返回类型（连 void 都没有）。最简单的构造函数是默认构造函数，它不接受任何参数：

```
class Motor {
public:
    Motor() {
        std::cout << "Motor 对象被创建" << std::endl;
        id = 0;
        speed = 0.0;
        temperature = 25.0;
    }
}
```

```

private:
    int id;
    double speed;
    double temperature;
};

int main() {
    Motor m; // 自动调用默认构造函数
    return 0;
}

```

当我们写 `Motor m;` 时，编译器自动调用 `Motor()` 构造函数。构造函数中的代码被执行，成员变量被初始化为指定的值。构造函数确保了对象一旦创建就处于有效状态，使用者无需额外的初始化步骤。

如果类没有定义任何构造函数，编译器会自动生成一个默认构造函数。这个编译器生成的默认构造函数不做任何事情（对于没有默认值的成员变量，它们将包含未定义的值）。但一旦我们定义了任何构造函数，编译器就不再自动生成默认构造函数：

```

class Point {
public:
    Point(double x, double y) { // 只定义了带参数的构造函数
        this->x = x;
        this->y = y;
    }

private:
    double x, y;
};

int main() {
    Point p1(1.0, 2.0); // 正确，调用 Point(double, double)
    // Point p2;          // 错误！没有默认构造函数
    return 0;
}

```

如果希望在定义了其他构造函数的同时保留默认构造函数，可以显式声明它：

```

class Point {
public:
    Point() = default; // 显式要求编译器生成默认构造函数
    Point(double x, double y) : x(x), y(y) {}

private:
    double x = 0.0;
    double y = 0.0;
};

```

`= default` 是 C++11 引入的语法，它告诉编译器生成该函数的默认版本。

1.5.12.3. 带参数的构造函数

更常见的情况是构造函数需要接受参数，以便根据不同的输入创建不同状态的对象：

```

class Target {
public:
    Target(double x, double y, double z, int color) {

```

```

        this->x = x;
        this->y = y;
        this->z = z;
        this->color = color;
        this->confidence = 1.0;
        this->timestamp = getCurrentTime();
    }

    void print() const {
        std::cout << "Target at (" << x << ", " << y << ", " << z << ")"
<< std::endl;
    }

private:
    double x, y, z;
    int color;
    double confidence;
    long long timestamp;

    long long getCurrentTime() {
        // 返回当前时间戳
        return 0; // 简化示例
    }
};

int main() {
    Target t1(100.0, 200.0, 50.0, 1); // 直接初始化
    Target t2 = Target(300.0, 400.0, 100.0, 2); // 显式调用构造函数
    Target t3{500.0, 600.0, 150.0, 1}; // C++11 花括号初始化

    t1.print();
    t2.print();
    t3.print();

    return 0;
}

```

带参数的构造函数使得创建对象和初始化可以在一步完成，避免了“创建后再设置”的繁琐模式。构造函数还可以包含验证逻辑，确保对象以合法状态创建：

```

class Motor {
public:
    Motor(int id, double maxSpeed) {
        if (id <= 0) {
            throw std::invalid_argument("电机 ID 必须为正数");
        }
        if (maxSpeed <= 0) {
            throw std::invalid_argument("最大转速必须为正数");
        }

        this->id = id;
        this->maxSpeed = maxSpeed;
        this->currentSpeed = 0.0;
    }

private:

```

```
    int id;
    double maxSpeed;
    double currentSpeed;
};
```

1.5.12.4. 构造函数重载

与普通函数一样，构造函数也可以重载——定义多个同名但参数列表不同的构造函数，为对象创建提供多种方式：

```
class PIDController {
public:
    // 默认构造函数：使用默认参数
    PIDController() {
        kp = 1.0;
        ki = 0.0;
        kd = 0.0;
        reset();
    }

    // 只设置 P 参数
    PIDController(double kp) {
        this->kp = kp;
        this->ki = 0.0;
        this->kd = 0.0;
        reset();
    }

    // 设置 PID 三个参数
    PIDController(double kp, double ki, double kd) {
        this->kp = kp;
        this->ki = ki;
        this->kd = kd;
        reset();
    }

    void reset() {
        integral = 0.0;
        lastError = 0.0;
    }

private:
    double kp, ki, kd;
    double integral;
    double lastError;
};

int main() {
    PIDController pid1;                      // 调用默认构造函数
    PIDController pid2(2.0);                  // 调用 PIDController(double)
    PIDController pid3(1.5, 0.1, 0.05);       // 调用 PIDController(double,
                                              double, double)

    return 0;
}
```

编译器根据调用时提供的参数自动选择匹配的构造函数版本。

另一种实现类似效果的方法是使用默认参数：

```
class PIDController {
public:
    PIDController(double kp = 1.0, double ki = 0.0, double kd = 0.0) {
        this->kp = kp;
        this->ki = ki;
        this->kd = kd;
        integral = 0.0;
        lastError = 0.0;
    }

private:
    double kp, ki, kd;
    double integral, lastError;
};

int main() {
    PIDController pid1; // kp=1.0, ki=0.0, kd=0.0
    PIDController pid2(2.0); // kp=2.0, ki=0.0, kd=0.0
    PIDController pid3(1.5, 0.1); // kp=1.5, ki=0.1, kd=0.0
    PIDController pid4(1.5, 0.1, 0.05); // kp=1.5, ki=0.1, kd=0.05

    return 0;
}
```

默认参数使得一个构造函数可以处理多种调用方式，减少了代码重复。

1.5.12.5. 成员初始化列表

在前面的例子中，我们在构造函数体内对成员变量进行赋值。C++ 提供了一种更高效的初始化方式——成员初始化列表（member initializer list）。它出现在构造函数参数列表之后、函数体之前，以冒号开头：

```
class Target {
public:
    Target(double x, double y, double z, int color)
        : x(x), y(y), z(z), color(color), confidence(1.0), timestamp(0)
    {
        // 构造函数体，可以为空
    }

private:
    double x, y, z;
    int color;
    double confidence;
    long long timestamp;
};
```

初始化列表中，每个成员的初始化形式为成员名(初始值)，多个成员之间用逗号分隔。这种语法与函数体内赋值有本质区别：初始化列表是真正的初始化，而函数体内是赋值。

对于基本类型，两者的效果通常相同。但对于某些情况，必须使用初始化列表：

常量成员必须在初始化列表中初始化，因为常量一旦创建就不能被赋值：

```
class Config {
public:
```

```

    Config(int maxConnections)
        : MAX_CONNECTIONS(maxConnections) // 必须在初始化列表中
    {
        // MAX_CONNECTIONS = maxConnections; // 错误！不能给 const 成员赋值
    }

private:
    const int MAX_CONNECTIONS;
};

```

引用成员也必须在初始化列表中初始化，因为引用必须在创建时绑定：

```

class Observer {
public:
    Observer(Subject& subject)
        : subject(subject) // 引用必须在初始化列表中绑定
    {
    }

private:
    Subject& subject;
};

```

没有默认构造函数的成员对象必须在初始化列表中初始化：

```

class Engine {
public:
    Engine(int power) : power(power) {} // 没有默认构造函数
private:
    int power;
};

class Car {
public:
    Car(int enginePower)
        : engine(enginePower) // 必须在初始化列表中初始化 engine
    {
    }

private:
    Engine engine;
};

```

即使不是必须使用初始化列表的情况，使用它也是推荐的做法。对于类类型的成员，初始化列表直接调用构造函数进行初始化，而函数体内赋值会先调用默认构造函数，再调用赋值运算符——多了一步不必要的操作：

```

class Example {
public:
    // 低效：先默认构造 name，再赋值
    Example(const std::string& n) {
        name = n; // 调用 string 的赋值运算符
    }

    // 高效：直接用 n 构造 name
    Example(const std::string& n)
        : name(n) // 调用 string 的拷贝构造函数
    {

```

```
}
```

```
private:  
    std::string name;  
};
```

初始化列表中成员的初始化顺序由它们在类中的声明顺序决定，而非初始化列表中的书写顺序。为避免混淆，建议初始化列表的顺序与成员声明顺序保持一致：

```
class Example {  
public:  
    Example(int val)  
        : a(val), b(a + 1) // 正确：a 先于 b 声明，所以 a 先初始化  
    {  
    }  
  
private:  
    int a; // 先声明  
    int b; // 后声明  
};
```

1.5.12.6. 委托构造函数

C++11 引入了委托构造函数 (delegating constructor)，允许一个构造函数调用同一类的另一个构造函数，减少代码重复：

```
class Motor {  
public:  
    // 主构造函数  
    Motor(int id, double maxSpeed, double gearRatio)  
        : id(id), maxSpeed(maxSpeed), gearRatio(gearRatio),  
          currentSpeed(0.0), temperature(25.0)  
    {  
        validateParameters();  
    }  
  
    // 委托构造函数：使用默认齿轮比  
    Motor(int id, double maxSpeed)  
        : Motor(id, maxSpeed, 1.0) // 委托给主构造函数  
    {  
    }  
  
    // 委托构造函数：使用默认参数  
    Motor(int id)  
        : Motor(id, 10000.0, 1.0) // 委托给主构造函数  
    {  
    }  
  
private:  
    void validateParameters() {  
        if (id <= 0 || maxSpeed <= 0 || gearRatio <= 0) {  
            throw std::invalid_argument("无效参数");  
        }  
    }  
  
    int id;  
    double maxSpeed;
```

```
    double gearRatio;
    double currentSpeed;
    double temperature;
};
```

委托构造函数在初始化列表中调用目标构造函数，调用形式与普通成员初始化类似。使用委托构造函数后，初始化逻辑集中在一个主构造函数中，其他构造函数只需要提供不同的参数组合，代码更加简洁且易于维护。

1.5.12.7. 析构函数

析构函数是与构造函数对应的特殊成员函数，它在对象销毁时自动调用。析构函数的名称是类名前加波浪号 ~，没有返回类型，也不接受任何参数：

```
class Logger {
public:
    Logger(const std::string& filename) {
        file.open(filename, std::ios::app);
        std::cout << "Logger 创建, 打开文件: " << filename << std::endl;
    }

    ~Logger() {
        if (file.is_open()) {
            file.close();
        }
        std::cout << "Logger 销毁, 关闭文件" << std::endl;
    }

    void log(const std::string& message) {
        file << message << std::endl;
    }

private:
    std::ofstream file;
};

void processData() {
    Logger logger("app.log"); // 构造函数被调用
    logger.log("开始处理");
    logger.log("处理完成");
} // logger 离开作用域, 析构函数被调用

int main() {
    std::cout << "程序开始" << std::endl;
    processData();
    std::cout << "程序结束" << std::endl;
    return 0;
}
```

输出将是：

```
程序开始
Logger 创建, 打开文件: app.log
Logger 销毁, 关闭文件
程序结束
```

析构函数的主要职责是释放对象持有的资源。这些资源可能包括：

动态分配的内存（通过 new 分配的）需要在析构函数中 delete：

```
class DynamicArray {
public:
    DynamicArray(size_t size)
        : size(size), data(new int[size])
    {
        std::fill(data, data + size, 0);
    }

    ~DynamicArray() {
        delete[] data; // 释放动态分配的内存
    }

    int& operator[](size_t index) {
        return data[index];
    }

private:
    size_t size;
    int* data;
};
```

打开的文件需要关闭，网络连接需要断开，锁需要释放——所有这些清理工作都应该在析构函数中完成。

与构造函数不同，析构函数不能重载，每个类只能有一个析构函数。如果没有显式定义析构函数，编译器会自动生成一个，它会调用每个成员对象的析构函数，但不会做其他事情。对于持有原始指针等需要手动释放的资源的类，必须显式定义析构函数。

1.5.12.8. RAII 思想

RAII (Resource Acquisition Is Initialization, 资源获取即初始化) 是 C++ 中最重要的编程惯用法之一。它的核心思想是：将资源的生命周期与对象的生命周期绑定——在构造函数中获取资源，在析构函数中释放资源。

这个思想解决了资源管理的根本问题：确保资源总是被正确释放，即使在异常发生时也是如此。

考虑一个不使用 RAII 的例子：

```
void processFile(const std::string& filename) {
    FILE* file = fopen(filename.c_str(), "r");
    if (!file) return;

    // 处理文件...
    if (someError) {
        fclose(file); // 必须记得关闭
        return;
    }

    // 更多处理...
    if (anotherError) {
        fclose(file); // 每个退出点都要关闭
        return;
    }
}
```

```
    fclose(file); // 正常退出也要关闭
}
```

每个可能的退出点都需要记得关闭文件，容易遗漏。如果中间的代码抛出异常，文件更是无法被关闭。

使用 RAII 重写：

```
class FileHandle {
public:
    FileHandle(const std::string& filename, const char* mode)
        : file(fopen(filename.c_str(), mode))
    {
        if (!file) {
            throw std::runtime_error("无法打开文件");
        }
    }

    ~FileHandle() {
        if (file) {
            fclose(file);
        }
    }

    FILE* get() const { return file; }

private:
    FILE* file;
};

void processFile(const std::string& filename) {
    FileHandle handle(filename, "r"); // 构造函数打开文件

    // 处理文件...
    if (someError) {
        return; // 不需要手动关闭，析构函数会处理
    }

    // 更多处理...
    if (anotherError) {
        return; // 同样，析构函数会处理
    }

    // 正常退出，析构函数自动关闭文件
}
```

无论函数如何退出——正常返回、提前返回、还是抛出异常——`FileHandle` 的析构函数都会被调用，文件都会被正确关闭。这就是 RAII 的威力：它将资源管理变成了自动的、异常安全的。

RAII 的应用无处不在：

```
// 标准库中的 RAII 示例
void examples() {
    // std::string 管理字符数组的内存
    std::string text = "Hello"; // 构造时分配内存
    // 离开作用域时自动释放
```

```

// std::vector 管理动态数组
std::vector<int> numbers(100); // 构造时分配
// 离开作用域时自动释放

// std::fstream 管理文件
std::ofstream file("output.txt"); // 构造时打开
file << "Hello";
// 离开作用域时自动关闭

// std::lock_guard 管理互斥锁
std::mutex mtx;
{
    std::lock_guard<std::mutex> lock(mtx); // 构造时加锁
    // 临界区代码
} // 离开作用域时自动解锁

// std::unique_ptr 管理动态分配的对象
std::unique_ptr<Motor> motor(new Motor(1, 10000.0)); // 构造时获得所有权
// 离开作用域时自动 delete
}

```

在 RoboMaster 开发中，RAII 思想应用于串口连接、相机资源、线程管理等各个方面：

```

class SerialPort {
public:
    SerialPort(const std::string& device, int baudrate)
        : fd(open(device.c_str(), O_RDWR | O_NOCTTY))
    {
        if (fd < 0) {
            throw std::runtime_error("无法打开串口");
        }
        configure(baudrate);
        std::cout << "串口已打开：" << device << std::endl;
    }

    ~SerialPort() {
        if (fd >= 0) {
            close(fd);
            std::cout << "串口已关闭" << std::endl;
        }
    }

    ssize_t write(const void* data, size_t len) {
        return ::write(fd, data, len);
    }

    ssize_t read(void* buffer, size_t len) {
        return ::read(fd, buffer, len);
    }

private:
    void configure(int baudrate);
    int fd;
};

void communicate() {
    SerialPort port("/dev/ttyUSB0", 115200);

```

```

        uint8_t command[] = {0xA5, 0x01, 0x02, 0x03};
        port.write(command, sizeof(command));

        uint8_t response[256];
        port.read(response, sizeof(response));

        // 不需要手动关闭串口
    }

```

1.5.12.9. 拷贝控制预览

当我们定义了管理资源的类时，还需要考虑对象被拷贝时会发生什么。默认情况下，C++ 会逐成员复制对象，这对于持有指针的类可能导致问题：

```

class DynamicArray {
public:
    DynamicArray(size_t size)
        : size(size), data(new int[size]) {}

    ~DynamicArray() {
        delete[] data;
    }

private:
    size_t size;
    int* data;
};

void problematic() {
    DynamicArray a(10);
    DynamicArray b = a; // 默认拷贝：b.data 指向与 a.data 相同的内存

} // 问题！a 和 b 的析构函数都会 delete 同一块内存

```

这种默认的浅拷贝导致两个对象共享同一块内存，当它们都被销毁时，会重复释放同一块内存，这是严重的错误。

解决方案是定义拷贝构造函数和拷贝赋值运算符来实现深拷贝，或者禁止拷贝。这些内容将在下一节“拷贝控制”中详细讨论。

1.5.12.10. 构造与析构的顺序

理解构造和析构的调用顺序对于正确使用类层次结构很重要。

对于包含成员对象的类，成员按照声明顺序构造，按相反顺序析构：

```

class Component {
public:
    Component(int id) : id(id) {
        std::cout << "Component " << id << " 构造" << std::endl;
    }
    ~Component() {
        std::cout << "Component " << id << " 析构" << std::endl;
    }
private:
    int id;
}

```

```

};

class System {
public:
    System() : c1(1), c2(2), c3(3) {
        std::cout << "System 构造" << std::endl;
    }
    ~System() {
        std::cout << "System 析构" << std::endl;
    }
private:
    Component c1;
    Component c2;
    Component c3;
};

int main() {
    System sys;
    return 0;
}

```

输出：

```

Component 1 构造
Component 2 构造
Component 3 构造
System 构造
System 析构
Component 3 析构
Component 2 析构
Component 1 析构

```

成员对象在包含它们的对象之前构造，在之后析构。这个顺序确保了在 `System` 的构造函数和析构函数中，所有成员对象都处于有效状态。

对于数组中的对象，按索引顺序构造，按相反顺序析构：

```

void arrayExample() {
    Motor motors[3] = {Motor(1), Motor(2), Motor(3)};
    // Motor(1) 先构造, Motor(3) 最后构造
} // Motor(3) 先析构, Motor(1) 最后析构

```

1.5.12.11. 实践建议

编写构造函数和析构函数时，以下原则值得遵循：

优先使用成员初始化列表。它不仅在某些情况下是必需的，而且通常更高效。养成使用初始化列表的习惯，可以避免很多潜在问题。

确保构造函数将对象初始化到有效状态。对象一旦创建，就应该可以安全使用。如果初始化可能失败，考虑抛出异常。

如果类管理资源，记得定义析构函数释放资源。同时考虑是否需要自定义拷贝行为——这是“三五法则”的一部分，将在下一节详细讨论。

拥抱 RAII。将资源管理封装在类中，让构造函数和析构函数自动处理资源的获取和释放。这不仅使代码更简洁，也使其更加异常安全。

避免在构造函数和析构函数中调用虚函数。这是一个微妙的陷阱，将在多态相关章节中解释。

构造函数和析构函数是类的生命周期管理机制。通过它们，我们可以精确控制对象从创建到销毁的整个过程，确保资源被正确获取和释放。然而，当对象被拷贝或赋值时，还需要额外的控制。下一节将深入探讨拷贝控制——如何正确处理对象的复制和赋值操作。

1.5.13. 拷贝控制

上一节末尾，我们遇到了一个棘手的问题：当一个管理动态内存的对象被拷贝时，默认的逐成员复制会导致两个对象共享同一块内存，最终在析构时重复释放，引发严重错误。这揭示了一个更深层的问题——对象的拷贝行为并非总是简单的内存复制。C++ 允许我们自定义对象在拷贝和赋值时的行为，这就是拷贝控制（copy control）。掌握拷贝控制是编写健壮的资源管理类的关键，也是理解 C++ 对象语义的重要一步。

1.5.13.1. 拷贝的发生时机

在深入拷贝控制的细节之前，先来了解拷贝何时发生。C++ 中对象拷贝的情况比想象中更加普遍。

最直观的是用一个对象初始化另一个对象：

```
Motor m1(1, 10000.0);
Motor m2 = m1;           // 拷贝初始化
Motor m3(m1);           // 直接初始化，也是拷贝
Motor m4{m1};            // 列表初始化，也是拷贝
```

函数参数按值传递时会发生拷贝：

```
void process(Motor m) { // m 是实参的拷贝
    ...
}

Motor motor(1, 10000.0);
process(motor); // 拷贝 motor 到参数 m
```

函数按值返回时也会发生拷贝（尽管编译器常常会优化掉）：

```
Motor createMotor() {
    Motor m(1, 10000.0);
    return m; // 返回时可能拷贝
}

Motor result = createMotor(); // 接收返回值时可能拷贝
```

将对象放入容器、从容器中取出、容器扩容时重新分配空间，这些操作也可能涉及拷贝。

理解拷贝的普遍性，就能理解为什么正确实现拷贝控制如此重要——错误的拷贝行为会在程序中各处引发问题。

1.5.13.2. 拷贝构造函数

拷贝构造函数是一种特殊的构造函数，它使用同类型的另一个对象来初始化新对象。拷贝构造函数的第一个参数必须是本类类型的引用（通常是常量引用），且不应有其他非默认参数：

```
class Motor {
public:
```

```

// 普通构造函数
Motor(int id, double maxSpeed)
    : id(id), maxSpeed(maxSpeed), currentSpeed(0.0)
{
    std::cout << "构造 Motor " << id << std::endl;
}

// 拷贝构造函数
Motor(const Motor& other)
    : id(other.id), maxSpeed(other.maxSpeed),
currentSpeed(other.currentSpeed)
{
    std::cout << "拷贝构造 Motor " << id << std::endl;
}

private:
    int id;
    double maxSpeed;
    double currentSpeed;
};

int main() {
    Motor m1(1, 10000.0); // 调用普通构造函数
    Motor m2 = m1;         // 调用拷贝构造函数
    Motor m3(m1);         // 调用拷贝构造函数
    return 0;
}

```

拷贝构造函数的参数必须是引用，这一点至关重要。如果参数是值传递，那么传递参数时就需要拷贝，而拷贝又需要调用拷贝构造函数，这会导致无限递归。

如果我们没有定义拷贝构造函数，编译器会自动生成一个。编译器生成的拷贝构造函数执行逐成员拷贝——对每个成员调用其拷贝构造函数（对于基本类型，就是简单的值复制）。对于只包含基本类型和标准库类型的简单类，编译器生成的版本通常足够：

```

class Point {
public:
    Point(double x, double y) : x(x), y(y) {}
    // 编译器自动生成的拷贝构造函数足以正确工作

private:
    double x, y;
};

class Target {
public:
    Target(const std::string& name, Point pos)
        : name(name), position(pos) {}
    // 编译器生成的版本会正确拷贝 string 和 Point

private:
    std::string name;
    Point position;
};

```

`std::string` 和 `Point` 都有正确的拷贝行为，因此包含它们的 `Target` 也能正确拷贝。

1.5.13.3. 浅拷贝的问题

问题出现在类包含指向动态分配内存的指针时。编译器生成的拷贝构造函数只复制指针的值（即地址），而不复制指针指向的数据。这称为浅拷贝（shallow copy）：

```
class DynamicArray {
public:
    DynamicArray(size_t size)
        : size(size), data(new int[size])
    {
        std::fill(data, data + size, 0);
        std::cout << "构造, 分配内存: " << data << std::endl;
    }

    ~DynamicArray() {
        std::cout << "析构, 释放内存: " << data << std::endl;
        delete[] data;
    }

    int& operator[](size_t index) { return data[index]; }
    size_t getSize() const { return size; }

private:
    size_t size;
    int* data; // 指向动态分配的内存
};

int main() {
    DynamicArray a(5);
    a[0] = 100;

    DynamicArray b = a; // 浅拷贝: b.data 与 a.data 指向同一块内存

    std::cout << "a[0] = " << a[0] << std::endl; // 100
    std::cout << "b[0] = " << b[0] << std::endl; // 100

    b[0] = 200; // 修改 b 也会影响 a
    std::cout << "a[0] = " << a[0] << std::endl; // 200!

    return 0;
} // 灾难: a 和 b 的析构函数都会 delete 同一块内存
```

运行这段代码，程序可能崩溃，或者产生不可预测的行为。这是因为：

1. $b = a$ 时， $b.data$ 被赋值为 $a.data$ 的值，两个指针指向同一块内存
2. 修改 $b[0]$ 实际上修改的是 a 和 b 共享的那块内存
3. 当 b 被销毁时，它的析构函数释放了那块内存
4. 当 a 被销毁时，它的析构函数试图释放已经被释放的内存——这是未定义行为

浅拷贝的问题本质上是所有权的混乱：两个对象都认为自己拥有同一块内存，都试图管理它的生命周期。

1.5.13.4. 深拷贝

解决方案是实现深拷贝 (deep copy): 不仅复制指针, 还复制指针指向的数据, 使每个对象拥有自己独立的资源副本:

```
class DynamicArray {
public:
    DynamicArray(size_t size)
        : size(size), data(new int[size])
    {
        std::fill(data, data + size, 0);
        std::cout << "构造, 分配内存: " << data << std::endl;
    }

    // 深拷贝的拷贝构造函数
    DynamicArray(const DynamicArray& other)
        : size(other.size), data(new int[other.size]) // 分配新内存
    {
        std::copy(other.data, other.data + size, data); // 复制数据
        std::cout << "拷贝构造, 分配新内存: " << data << std::endl;
    }

    ~DynamicArray() {
        std::cout << "析构, 释放内存: " << data << std::endl;
        delete[] data;
    }

    int& operator[](size_t index) { return data[index]; }
    size_t getSize() const { return size; }

private:
    size_t size;
    int* data;
};

int main() {
    DynamicArray a(5);
    a[0] = 100;

    DynamicArray b = a; // 深拷贝: b 拥有自己的内存

    std::cout << "a[0] = " << a[0] << std::endl; // 100
    std::cout << "b[0] = " << b[0] << std::endl; // 100

    b[0] = 200; // 修改 b 不影响 a
    std::cout << "a[0] = " << a[0] << std::endl; // 100, 不变
    std::cout << "b[0] = " << b[0] << std::endl; // 200

    return 0;
} // 正常: a 和 b 各自释放自己的内存
```

现在每个 `DynamicArray` 对象都拥有自己独立的内存块, 修改一个不会影响另一个, 销毁时也各自释放自己的资源, 不会产生冲突。

1.5.13.5. 拷贝赋值运算符

拷贝构造函数处理的是用已有对象初始化新对象的情况。但还有另一种拷贝场景：将一个已存在的对象赋值给另一个已存在的对象。这由拷贝赋值运算符 (copy assignment operator) 处理：

```
DynamicArray a(5);
DynamicArray b(10);
```

```
b = a; // 赋值，不是初始化！b 已经存在
```

拷贝赋值运算符是 `operator=` 的重载，它接受同类型的常量引用作为参数，返回对象自身的引用（以支持链式赋值）：

```
class DynamicArray {
public:
    // ... 构造函数和拷贝构造函数 ...

    // 拷贝赋值运算符
    DynamicArray& operator=(const DynamicArray& other) {
        std::cout << "拷贝赋值" << std::endl;

        if (this == &other) { // 自赋值检查
            return *this;
        }

        // 释放当前资源
        delete[] data;

        // 分配新资源并复制
        size = other.size;
        data = new int[size];
        std::copy(other.data, other.data + size, data);

        return *this;
    }

    // ... 其他成员 ...
};
```

拷贝赋值运算符与拷贝构造函数的关键区别在于：赋值时，目标对象已经存在，可能已经持有资源。因此，赋值运算符必须先释放旧资源，再获取新资源。

自赋值检查 (`if (this == &other)`) 看似多余，但在某些情况下是必要的。考虑 `a = a` 这种情况：如果没有检查，我们会先 `delete[] data`，然后试图从已释放的 `other.data`（实际上就是刚释放的 `data`）复制数据——这显然是错误的。

实际上，上面的实现还有一个问题：如果 `new int[size]` 抛出异常，对象会处于不一致状态 (`data` 已被删除但尚未重新分配)。更安全的实现是先分配新资源，再释放旧资源：

```
DynamicArray& operator=(const DynamicArray& other) {
    if (this == &other) {
        return *this;
    }

    // 先分配新资源
    int* newData = new int[other.size];
    std::copy(other.data, other.data + other.size, newData);
```

```

    // 成功后再释放旧资源
    delete[] data;
    data = newData;
    size = other.size;

    return *this;
}

```

如果 `new` 失败并抛出异常，旧资源仍然完好，对象状态不变。这种技术称为“先复制后交换”（copy-and-swap），可以进一步改进。

1.5.13.6. copy-and-swap 惯用法

`copy-and-swap` 是实现拷贝赋值运算符的优雅且异常安全的方法。它结合使用拷贝构造函数和 `swap` 函数：

```

class DynamicArray {
public:
    DynamicArray(size_t size)
        : size(size), data(new int[size])
    {
        std::fill(data, data + size, 0);
    }

    DynamicArray(const DynamicArray& other)
        : size(other.size), data(new int[other.size])
    {
        std::copy(other.data, other.data + size, data);
    }

    ~DynamicArray() {
        delete[] data;
    }

    // swap 函数：交换两个对象的内部状态
    friend void swap(DynamicArray& first, DynamicArray& second) noexcept {
        using std::swap;
        swap(first.size, second.size);
        swap(first.data, second.data);
    }

    // copy-and-swap 实现的赋值运算符
    DynamicArray& operator=(DynamicArray other) { // 注意：按值传递
        swap(*this, other);
        return *this;
    }

    int& operator[](size_t index) { return data[index]; }
    size_t getSize() const { return size; }

private:
    size_t size;
    int* data;
};

```

这个实现的精妙之处在于：

参数 `other` 是按值传递的，这意味着在进入函数时，拷贝构造函数已经被调用，`other` 是实参的一个副本。如果拷贝构造失败（如内存分配异常），函数根本不会执行，原对象不受影响。

然后，我们将当前对象与这个副本交换内部状态。交换之后，`*this` 拥有了副本的资源（即新值），而 `other` 拥有了原来的资源（即旧值）。

当函数返回时，`other` 被销毁，它的析构函数释放旧资源。

这种方法自动处理了自赋值 (`a = a` 时，只是与自己的副本交换，没有问题)，代码简洁，且提供强异常安全保证。

1.5.13.7. 三五法则

C++ 有一条著名的经验法则，用于指导何时需要自定义拷贝控制成员。

三法则（Rule of Three）指出：如果一个类需要自定义析构函数、拷贝构造函数或拷贝赋值运算符中的任何一个，那么通常也需要自定义另外两个。

这是因为需要自定义这些函数的情况通常意味着类管理着某种资源。如果析构函数需要释放资源，那么拷贝时就需要决定如何处理资源（深拷贝还是共享）；如果拷贝构造函数需要复制资源，那么赋值运算符同样需要。

C++11 引入了移动语义后，三法则扩展为五法则（Rule of Five）：如果需要自定义析构函数、拷贝构造函数、拷贝赋值运算符、移动构造函数或移动赋值运算符中的任何一个，通常应该考虑全部五个。移动语义将在后续章节介绍。

还有一条相关的零法则（Rule of Zero）：如果可能，设计类时应该不需要自定义任何拷贝控制成员。通过使用智能指针和标准库容器等自动管理资源的类型，可以让编译器生成的默认版本正确工作：

```
// 遵循零法则的设计
class ModernArray {
public:
    ModernArray(size_t size)
        : data(size, 0) // vector 自动管理内存
    {
    }

// 不需要自定义析构函数、拷贝构造函数、拷贝赋值运算符
// 编译器生成的版本会正确调用 vector 的相应函数

    int& operator[](size_t index) { return data[index]; }
    size_t getSize() const { return data.size(); }

private:
    std::vector<int> data; // 让 vector 管理内存
};
```

`std::vector` 已经正确实现了拷贝控制，因此包含 `std::vector` 的类也自动具有正确的拷贝行为。这种设计更简洁、更安全、不容易出错。

1.5.13.8. 禁止拷贝

有些类根本不应该被拷贝。例如，表示唯一资源的类（如线程、互斥锁、文件句柄的独占所有权）、单例模式的类，或者拷贝成本过高且没有意义的类。

C++11 之前，禁止拷贝的方法是将拷贝构造函数和拷贝赋值运算符声明为私有且不实现：

```
// C++11 之前的做法
class NonCopyable {
private:
    NonCopyable(const NonCopyable&); // 只声明，不实现
    NonCopyable& operator=(const NonCopyable&); // 只声明，不实现
public:
    NonCopyable() {}
};
```

C++11 引入了更清晰的语法——使用 `= delete` 显式禁止函数：

```
class UniqueResource {
public:
    UniqueResource() : handle(acquireResource()) {}
    ~UniqueResource() { releaseResource(handle); }

    // 禁止拷贝
    UniqueResource(const UniqueResource&) = delete;
    UniqueResource& operator=(const UniqueResource&) = delete;

    void use() { /* 使用资源 */ }

private:
    ResourceHandle handle;
};

int main() {
    UniqueResource r1;
    // UniqueResource r2 = r1; // 编译错误：拷贝构造函数被删除
    // UniqueResource r3;
    // r3 = r1; // 编译错误：拷贝赋值运算符被删除
    return 0;
}
```

`= delete` 不仅适用于拷贝控制成员，还可以用于禁止任何不希望被调用的函数重载。编译器在尝试调用被删除的函数时会产生清晰的错误信息。

1.5.13.9. 显式默认

与 `= delete` 相对的是 `= default`，它显式要求编译器生成默认版本的特殊成员函数：

```
class Example {
public:
    Example() = default; // 显式使用默认构造函数
    Example(int value) : value(value) {}

    Example(const Example&) = default; // 显式使用默认拷贝构造
    Example& operator=(const Example&) = default; // 显式使用默认拷贝赋值
    ~Example() = default; // 显式使用默认析构

private:
```

```
    int value = 0;
};
```

使用 `= default` 的好处包括：

明确表达意图——告诉读者“这里使用默认行为是有意的，不是遗忘”。

即使定义了其他构造函数，也可以保留默认构造函数。

在某些情况下，编译器生成的版本可能比手写的更高效（如可以是 trivial 的）。

1.5.13.10. 一个完整的例子

让我们用一个完整的例子来综合运用本节的知识。以下是一个简化的字符串类实现：

```
#include <iostream>
#include <cstring>
#include <algorithm>

class MyString {
public:
    // 默认构造函数
    MyString() : len(0), data(new char[1]) {
        data[0] = '\0';
    }

    // 从 C 字符串构造
    MyString(const char* str) {
        len = std::strlen(str);
        data = new char[len + 1];
        std::strcpy(data, str);
    }

    // 拷贝构造函数（深拷贝）
    MyString(const MyString& other) : len(other.len), data(new char[other.len +
1]) {
        std::strcpy(data, other.data);
        std::cout << "拷贝构造: " << data << "\n" << std::endl;
    }

    // 析构函数
    ~MyString() {
        delete[] data;
    }

    // swap 函数
    friend void swap(MyString& first, MyString& second) noexcept {
        using std::swap;
        swap(first.len, second.len);
        swap(first.data, second.data);
    }

    // 拷贝赋值运算符（copy-and-swap）
    MyString& operator=(MyString other) {
        std::cout << "拷贝赋值: " << other.data << "\n" << std::endl;
        swap(*this, other);
        return *this;
    }
}
```

```

// 访问接口
size_t length() const { return len; }
const char* c_str() const { return data; }

char& operator[](size_t index) { return data[index]; }
char operator[](size_t index) const { return data[index]; }

// 字符串连接
MyString operator+(const MyString& other) const {
    MyString result;
    delete[] result.data;

    result.len = len + other.len;
    result.data = new char[result.len + 1];
    std::strcpy(result.data, data);
    std::strcat(result.data, other.data);

    return result;
}

// 输出运算符
friend std::ostream& operator<<(std::ostream& os, const MyString& str) {
    return os << str.data;
}

private:
    size_t len;
    char* data;
};

int main() {
    MyString s1("Hello");
    MyString s2(" World");

    std::cout << "s1: " << s1 << std::endl;
    std::cout << "s2: " << s2 << std::endl;

    MyString s3 = s1; // 拷贝构造
    std::cout << "s3: " << s3 << std::endl;

    MyString s4;
    s4 = s1 + s2; // 赋值
    std::cout << "s4: " << s4 << std::endl;

    s1[0] = 'h';
    std::cout << "修改后 s1: " << s1 << std::endl;
    std::cout << "s3 不变: " << s3 << std::endl; // 深拷贝, s1 的修改不影响 s3

    return 0;
}

```

这个例子展示了：

深拷贝的拷贝构造函数，确保每个对象拥有独立的数据副本。

使用 copy-and-swap 惯用法的拷贝赋值运算符，既简洁又异常安全。

正确的析构函数释放动态分配的内存。

遵循三法则，三个函数配合工作，确保对象在整个生命周期内正确管理资源。

1.5.13.11. 实践建议

编写拷贝控制成员时，以下原则值得遵循：

优先使用零法则。如果可能，使用 `std::string`、`std::vector`、智能指针等标准库类型来管理资源，让编译器生成的默认函数正确工作。这是最简单也最不容易出错的方式。

如果必须管理资源，遵循三/五法则。自定义析构函数、拷贝构造函数和拷贝赋值运算符（C++11 后还包括移动操作），确保它们协调一致。

使用 `copy-and-swap` 实现赋值运算符。这种方式代码简洁、自动处理自赋值、提供强异常安全保证。

对于不应拷贝的类，显式删除拷贝操作。使用 `= delete` 明确表达意图，让编译器帮助捕获错误使用。

测试拷贝行为。编写测试用例验证拷贝后的对象与原对象独立、修改一个不影响另一个、两个对象都能正确析构。

拷贝控制是 C++ 对象模型的核心部分。正确实现拷贝语义，能够让自定义类型像内置类型一样自然地使用——可以放入容器、作为参数传递、作为返回值返回。然而，拷贝有时意味着性能开销，特别是对于大型对象。C++11 引入的移动语义提供了一种优化方案，允许“窃取”临时对象的资源而非复制它们。但在学习移动语义之前，我们先来探索类的另一个重要特性——继承，它使得代码复用和类型层次结构成为可能。

1.5.14. 类的继承

在前面的章节中，我们学习了如何定义类、如何控制对象的生命周期、如何正确处理对象的拷贝。这些都是构建单个类的基础技术。然而，现实世界中的概念往往存在层次关系：电机是一种执行器，步兵机器人是一种机器人，自动瞄准系统是一种瞄准系统。这些“是一种”（is-a）的关系可以用继承（inheritance）来表达。继承允许我们基于已有的类创建新类，新类自动获得已有类的特性，同时可以添加新特性或修改已有行为。这不仅实现了代码复用，更重要的是建立了类型之间的层次结构，为多态奠定了基础。

1.5.14.1. 继承的基本概念

假设我们正在开发 RoboMaster 机器人的控制系统，需要处理不同类型的机器人：步兵、英雄、工程、哨兵等。这些机器人有很多共同的特性——都有底盘、都能移动、都有血量——但也有各自独特的能力。如果为每种机器人都从头编写一个类，会有大量重复代码。

继承提供了一种更好的方式：先定义一个包含共同特性的基类（base class，也称父类或超类），然后让各种具体的机器人类从基类派生（derive），自动继承共同特性，同时添加各自的独特功能。

```
// 基类：通用机器人
class Robot {
public:
    Robot(int id, int maxHealth)
        : id(id), health(maxHealth), maxHealth(maxHealth)
```

```

{
}

void takeDamage(int damage) {
    health -= damage;
    if (health < 0) health = 0;
}

void heal(int amount) {
    health += amount;
    if (health > maxHealth) health = maxHealth;
}

bool isAlive() const {
    return health > 0;
}

int getHealth() const { return health; }
int getId() const { return id; }

protected:
    int id;
    int health;
    int maxHealth;
};

// 派生类: 步兵机器人
class Infantry : public Robot {
public:
    Infantry(int id)
        : Robot(id, 500) // 调用基类构造函数, 步兵血量 500
    {

    }

    void shoot() {
        std::cout << "步兵 " << id << " 发射弹丸" << std::endl;
    }
};

// 派生类: 英雄机器人
class Hero : public Robot {
public:
    Hero(int id)
        : Robot(id, 600) // 英雄血量 600
    {

    }

    void shootLarge() {
        std::cout << "英雄 " << id << " 发射大弹丸" << std::endl;
    }
};

```

在这个例子中，Robot 是基类，Infantry 和 Hero 是派生类。派生类使用 : public Robot 语法声明它们继承自 Robot。派生类自动拥有基类的所有成员（id、health、takeDamage() 等），同时可以定义自己的成员（如 Infantry::shoot()、Hero::shootLarge()）。

使用这些类：

```
int main() {
    Infantry infantry(1);
    Hero hero(2);

    // 派生类对象可以使用基类的成员函数
    infantry.takeDamage(100);
    hero.takeDamage(150);

    std::cout << "步兵血量：" << infantry.getHealth() << std::endl; // 400
    std::cout << "英雄血量：" << hero.getHealth() << std::endl; // 450

    // 派生类对象可以使用自己的成员函数
    infantry.shoot();
    hero.shootLarge();

    return 0;
}
```

继承建立了“是一种”的关系：`Infantry` 是一种 `Robot`, `Hero` 也是一种 `Robot`。这种关系不仅是概念上的，在 C++ 类型系统中也有体现——派生类对象可以被当作基类对象使用，这是多态的基础。

1.5.14.2. 继承的语法

派生类的定义语法如下：

```
class 派生类名 : 访问说明符 基类名 {
    // 派生类成员
};
```

访问说明符可以是 `public`、`protected` 或 `private`，它决定了基类成员在派生类中的访问权限。最常用的是 `public` 继承，它保持基类成员的原有访问级别：

```
class Infantry : public Robot {
    // Robot 的 public 成员在 Infantry 中仍是 public
    // Robot 的 protected 成员在 Infantry 中仍是 protected
    // Robot 的 private 成员在 Infantry 中不可直接访问
};
```

`protected` 继承和 `private` 继承较少使用，它们会降低基类成员的访问级别。在 `protected` 继承中，基类的 `public` 成员变为 `protected`；在 `private` 继承中，基类的所有可访问成员都变为 `private`。这些形式更多用于实现细节的封装，而非表达“是一种”的关系。

C++ 支持多重继承——一个类可以从多个基类派生：

```
class FlyingRobot : public Robot, public Flyable {
    // 同时继承 Robot 和 Flyable 的特性
};
```

多重继承功能强大但也容易引入复杂性（如菱形继承问题），在实践中应谨慎使用。本节主要讨论单继承。

1.5.14.3. 访问控制与继承

理解继承中的访问控制对于正确设计类层次结构至关重要。C++ 的三个访问级别在继承中有不同的表现：

`public` 成员对所有代码可见。基类的公开成员在派生类中仍然是公开的，外部代码可以通过派生类对象访问这些成员。

`private` 成员只对定义它的类可见。基类的私有成员对派生类也是不可见的——派生类不能直接访问基类的私有成员，只能通过基类提供的公开或受保护接口间接访问。

`protected` 成员介于两者之间。它对外部代码不可见（像 `private`），但对派生类可见（不像 `private`）。`protected` 专为继承设计，用于那些需要在派生类中访问但不想暴露给外部的成员。

```
class Base {
public:
    int publicData;
    void publicFunc() {}

protected:
    int protectedData;
    void protectedFunc() {}

private:
    int privateData;
    void privateFunc() {}
};

class Derived : public Base {
public:
    void accessBaseMembers() {
        publicData = 1;           // 正确, public 成员可访问
        publicFunc();            // 正确

        protectedData = 2;       // 正确, protected 成员对派生类可访问
        protectedFunc();         // 正确

        // privateData = 3;      // 错误! private 成员不可访问
        // privateFunc();         // 错误!
    }
};

int main() {
    Derived d;
    d.publicData = 1;          // 正确, public 成员
    d.publicFunc();            // 正确

    // d.protectedData = 2;    // 错误! protected 对外部不可见
    // d.privateData = 3;      // 错误! private 对外部不可见

    return 0;
}
```

在设计类层次时，需要仔细考虑每个成员的访问级别。一般原则是：

数据成员通常设为 `private` 或 `protected`。如果派生类需要直接访问，使用 `protected`；否则使用 `private` 并提供访问接口。

对外接口设为 `public`。这些是类的使用者（包括派生类对象的使用者）可以调用的方法。供派生类扩展或修改的方法设为 `protected` 或 `public`。这取决于是否也想让外部代码调用。

在前面 Robot 的例子中，我们将 id、health、maxHealth 设为 protected，使得派生类可以直接访问这些数据（如 Infantry::shoot() 中使用 id），同时对外部代码隐藏这些实现细节。

1.5.14.4. 派生类的构造函数

派生类对象包含从基类继承的成员和派生类自己定义的成员。构造派生类对象时，这些成员都需要被初始化。C++ 的规则是：先构造基类部分，再构造派生类部分。

派生类构造函数通过成员初始化列表调用基类构造函数：

```
class Robot {
public:
    Robot(int id, int maxHealth)
        : id(id), health(maxHealth), maxHealth(maxHealth)
    {
        std::cout << "Robot 构造: id=" << id << std::endl;
    }

protected:
    int id;
    int health;
    int maxHealth;
};

class Infantry : public Robot {
public:
    Infantry(int id, int ammo)
        : Robot(id, 500), // 先调用基类构造函数
          ammoCount(ammo) // 再初始化派生类成员
    {
        std::cout << "Infantry 构造: ammo=" << ammoCount << std::endl;
    }

private:
    int ammoCount;
};

int main() {
    Infantry infantry(1, 100);
    return 0;
}
```

输出：

```
Robot 构造: id=1
Infantry 构造: ammo=100
```

基类构造函数必须在初始化列表中调用，不能在构造函数体内调用。如果不显式调用基类构造函数，编译器会尝试调用基类的默认构造函数。如果基类没有默认构造函数，则会产生编译错误：

```
class Base {
public:
    Base(int x) : value(x) {} // 只有带参数的构造函数，没有默认构造函数
private:
    int value;
};
```

```

class Derived : public Base {
public:
    Derived() // 错误！没有调用 Base 的构造函数，且 Base 没有默认构造函数
    {
    }

    Derived(int x) : Base(x) {} // 正确，显式调用 Base(int)
};

```

派生类构造函数可以接受额外的参数，用于初始化派生类特有的成员，同时将必要的参数传递给基类构造函数：

```

class Engineer : public Robot {
public:
    Engineer(int id, int maxHealth, int repairRate)
        : Robot(id, maxHealth),           // 传递给基类
          repairRate(repairRate)         // 初始化派生类成员
    {
    }

    void repairAlly(Robot& ally) {
        ally.heal(repairRate);
        std::cout << "工程机器人修理了友军" << std::endl;
    }

private:
    int repairRate;
};

```

1.5.14.5. 析构函数与继承

与构造顺序相反，析构时先执行派生类析构函数，再执行基类析构函数。这个顺序确保了在派生类析构函数执行期间，基类部分仍然有效：

```

class Robot {
public:
    Robot(int id) : id(id) {
        std::cout << "Robot 构造: " << id << std::endl;
    }

    ~Robot() {
        std::cout << "Robot 析构: " << id << std::endl;
    }

protected:
    int id;
};

class Infantry : public Robot {
public:
    Infantry(int id) : Robot(id) {
        std::cout << "Infantry 构造" << std::endl;
    }

    ~Infantry() {
        std::cout << "Infantry 析构" << std::endl;
    }
};

```

```

    }
};

int main() {
    Infantry infantry(1);
    return 0;
}

```

输出：

```

Robot 构造: 1
Infantry 构造
Infantry 析构
Robot 析构: 1

```

这种“先进后出”的顺序与栈的行为一致：基类是“先进”的（先构造），所以“后出”（后析构）。

派生类的析构函数会自动调用基类的析构函数，不需要（也不应该）显式调用。如果派生类管理了资源，只需在派生类析构函数中释放派生类自己的资源，基类的资源由基类析构函数负责：

```

class ResourceHolder : public Robot {
public:
    ResourceHolder(int id) : Robot(id), buffer(new char[1024]) {
        std::cout << "分配缓冲区" << std::endl;
    }

    ~ResourceHolder() {
        delete[] buffer; // 只释放派生类的资源
        std::cout << "释放缓冲区" << std::endl;
        // 基类析构函数会自动被调用
    }

private:
    char* buffer;
};

```

关于析构函数与继承，还有一个重要的话题——虚析构函数，这将在下一节“多态与虚函数”中详细讨论。简单来说，如果一个类可能被继承，其析构函数通常应该声明为 `virtual`。

1.5.14.6. 在派生类中访问基类成员

派生类可以直接访问基类的 `public` 和 `protected` 成员，就像它们是派生类自己的成员一样。但有时需要明确指定访问的是基类的成员，特别是当派生类定义了同名成员时。

使用作用域解析运算符 `::` 可以明确指定访问基类成员：

```

class Base {
public:
    void print() {
        std::cout << "Base::print()" << std::endl;
    }

protected:
    int value = 10;
};

class Derived : public Base {
public:
    void print() { // 隐藏了基类的 print()

```

```

        std::cout << "Derived::print()" << std::endl;
    }

    void printBoth() {
        print();           // 调用 Derived::print()
        Base::print();    // 明确调用 Base::print()
    }

    void showValues() {
        int value = 20;   // 局部变量, 隐藏了成员
        std::cout << "局部 value: " << value << std::endl;          // 20
        std::cout << "成员 value: " << this->value << std::endl;      // 10 (继承
自 Base)
        std::cout << "Base::value: " << Base::value << std::endl;      // 10
    }
};


```

当派生类定义了与基类同名的成员（函数或变量）时，基类成员被“隐藏”（hidden）。这与后面要介绍的“覆盖”（override）不同——隐藏是名称查找层面的概念，覆盖是多态行为层面的概念。

1.5.14.7. 派生类与基类的类型关系

继承建立的“是一种”关系在类型系统中有重要体现：派生类对象可以被当作基类对象使用。具体来说，可以将派生类对象绑定到基类引用，或将派生类对象的地址赋给基类指针：

```

void displayHealth(const Robot& robot) {
    std::cout << "血量: " << robot.getHealth() << std::endl;
}

int main() {
    Infantry infantry(1);
    Hero hero(2);

    displayHealth(infantry); // 正确, Infantry 是 Robot
    displayHealth(hero);    // 正确, Hero 是 Robot

    Robot* robotPtr = &infantry; // 基类指针指向派生类对象
    robotPtr->takeDamage(50);   // 通过基类指针调用基类成员

    Robot& robotRef = hero;     // 基类引用绑定到派生类对象
    robotRef.heal(100);        // 通过基类引用调用基类成员

    return 0;
}

```

这种向上转型（upcasting）是安全的，因为派生类对象包含基类的所有成员。通过基类指针或引用访问对象时，只能访问基类中定义的成员：

```

Robot* ptr = &infantry;
ptr->takeDamage(50); // 正确, takeDamage 是 Robot 的成员
// ptr->shoot();    // 错误! Robot 没有 shoot() 方法

```

反向的转换——将基类对象当作派生类对象——称为向下转型（downcasting），这通常是不安全的，需要使用特殊的转型操作符，将在后续章节讨论。

1.5.14.8. 对象切片

当派生类对象被赋值给基类对象（而非指针或引用）时，会发生对象切片（object slicing）：派生类特有的成员被“切掉”，只保留基类部分：

```
Infantry infantry(1);
infantry.takeDamage(100);

Robot robot = infantry; // 对象切片!
// robot 只有 Robot 的成员，Infantry 特有的成员丢失了
```

对象切片通常不是预期的行为，它会导致信息丢失。要保持派生类的完整信息，应该使用指针或引用：

```
Infantry infantry(1);

Robot& ref = infantry; // 引用，无切片
Robot* ptr = &injury; // 指针，无切片

// ref 和 ptr 仍然指向完整的 Infantry 对象
```

在 RoboMaster 开发中，经常需要将不同类型的机器人统一管理。应该使用指针或引用的容器，而非对象容器：

```
// 错误的做法：会发生对象切片
std::vector<Robot> robots;
robots.push_back(Infantry(1)); // 切片！只保存了 Robot 部分

// 正确的做法：使用指针
std::vector<Robot*> robots;
Infantry* infantry = new Infantry(1);
Hero* hero = new Hero(2);
robots.push_back(infantry);
robots.push_back(hero);

// 更好的做法：使用智能指针
std::vector<std::unique_ptr<Robot>> robots;
robots.push_back(std::make_unique<Infantry>(1));
robots.push_back(std::make_unique<Hero>(2));
```

1.5.14.9. 继承层次的设计

设计良好的继承层次应该反映真实的“是一种”关系。如果 B 继承自 A，那么 B 应该在概念上是 A 的一种特殊情况，B 的对象可以在任何需要 A 的地方使用。这称为里氏替换原则（Liskov Substitution Principle）。

```
// 良好的继承设计
class Shape {
public:
    virtual double area() const = 0;
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() const override { return width * height; }
private:
```

```

        double width, height;
    };

    class Circle : public Shape {
public:
    Circle(double r) : radius(r) {}
    double area() const override { return 3.14159 * radius * radius; }
private:
    double radius;
};

// Rectangle 是一种 Shape, Circle 也是一种 Shape
// 可以统一处理各种形状
void printArea(const Shape& shape) {
    std::cout << "面积: " << shape.area() << std::endl;
}

```

不应该仅仅为了复用代码而使用继承。如果两个类之间的关系是“有一个”(has-a) 而非“是一种”(is-a)，应该使用组合而非继承：

```

// 错误: Engine 不是 Robot 的一种
class Engine : public Robot { }; // 不合理

// 正确: Robot 有一个 Engine
class Robot {
private:
    Engine engine; // 组合关系
};

```

继承层次不宜过深。过深的层次增加复杂性，使得理解和维护变得困难。如果发现层次超过三四层，应该考虑是否可以通过组合或重新设计来简化。

1.5.14.10. 一个完整的例子

让我们用一个更完整的例子来综合运用继承的各个方面：

```

#include <iostream>
#include <string>
#include <vector>
#include <memory>

// 基类: 机器人
class Robot {
public:
    Robot(int id, const std::string& name, int maxHealth)
        : id(id), name(name), health(maxHealth), maxHealth(maxHealth)
    {
        std::cout << "创建机器人: " << name << " (ID: " << id << ")" <<
        std::endl;
    }

    virtual ~Robot() {
        std::cout << "销毁机器人: " << name << std::endl;
    }

    void takeDamage(int damage) {
        health -= damage;
    }

```

```

        if (health < 0) health = 0;
        std::cout << name << " 受到 " << damage << " 点伤害, 剩余血量: " << health
<< std::endl;
    }

    void heal(int amount) {
        health += amount;
        if (health > maxHealth) health = maxHealth;
    }

    bool isAlive() const { return health > 0; }
    int getHealth() const { return health; }
    const std::string& getName() const { return name; }

    virtual void performAction() {
        std::cout << name << " 待机中..." << std::endl;
    }

protected:
    int id;
    std::string name;
    int health;
    int maxHealth;
};

// 派生类: 步兵机器人
class Infantry : public Robot {
public:
    Infantry(int id)
        : Robot(id, "步兵-" + std::to_string(id), 500),
          ammo(300), fireRate(10)
    {
    }

    ~Infantry() override {
        std::cout << "步兵专用资源已释放" << std::endl;
    }

    void shoot() {
        if (ammo > 0) {
            ammo--;
            std::cout << getName() << " 发射弹丸 (剩余: " << ammo << ")"
<< std::endl;
        } else {
            std::cout << getName() << " 弹药耗尽! " << std::endl;
        }
    }

    void reload(int amount) {
        ammo += amount;
        std::cout << getName() << " 装填弹药, 当前: " << ammo << std::endl;
    }

    void performAction() override {
        shoot();
    }
};

```

```

    }

private:
    int ammo;
    int fireRate;
};

// 派生类: 工程机器人
class Engineer : public Robot {
public:
    Engineer(int id)
        : Robot(id, "工程-" + std::to_string(id), 600),
          repairRate(50)
    {
    }

    void repair(Robot& target) {
        if (&target == this) {
            std::cout << getName() << " 无法修理自己" << std::endl;
            return;
        }
        target.heal(repairRate);
        std::cout << getName() << " 修理了 " << target.getName()
            << ", 恢复 " << repairRate << " 点血量" << std::endl;
    }

    void performAction() override {
        std::cout << getName() << " 准备进行工程作业" << std::endl;
    }
};

private:
    int repairRate;
};

int main() {
    // 使用智能指针管理机器人
    std::vector<std::unique_ptr<Robot>> team;

    team.push_back(std::make_unique<Infantry>(1));
    team.push_back(std::make_unique<Infantry>(2));
    team.push_back(std::make_unique<Engineer>(3));

    std::cout << "\n--- 战斗开始 ---\n" << std::endl;

    // 通过基类指针统一操作
    for (auto& robot : team) {
        robot->performAction();
    }

    std::cout << "\n--- 受到攻击 ---\n" << std::endl;

    team[0]->takeDamage(200);
    team[1]->takeDamage(150);

    std::cout << "\n--- 工程机器人修理 ---\n" << std::endl;
}

```

```

// 需要向下转型才能调用派生类特有方法
Engineer* engineer = dynamic_cast<Engineer*>(team[2].get());
if (engineer) {
    engineer->repair(*team[0]);
    engineer->repair(*team[1]);
}

std::cout << "\n--- 程序结束 ---\n" << std::endl;

return 0;
}

```

这个例子展示了继承的多个方面：基类定义共同接口，派生类扩展功能；构造函数链式调用；通过基类指针统一管理不同类型的对象；虚函数实现多态行为（将在下一节详细介绍）。

继承是面向对象编程的三大支柱之一（另外两个是封装和多态）。通过继承，我们可以建立类型层次结构，实现代码复用，并为多态奠定基础。然而，继承本身并不能实现真正的运行时多态——当通过基类指针调用成员函数时，调用的总是基类版本。要让派生类能够替换基类的行为，需要使用虚函数。下一节将深入探讨多态与虚函数，揭示面向对象编程最强大的特性。

1.5.15. 多态与虚函数

上一节我们学习了继承，它允许派生类复用基类的代码，并通过基类指针或引用统一管理不同类型的对象。然而，当通过基类指针调用成员函数时，调用的始终是基类的版本，即使指针实际指向的是派生类对象。这在很多场景下并非我们想要的行为——我们希望调用的是派生类重新定义的版本，让不同类型的对象表现出不同的行为。这种“同一操作作用于不同对象产生不同效果”的能力称为多态（polymorphism），而虚函数（virtual function）正是 C++ 实现运行时多态的机制。多态是面向对象编程最强大的特性之一，它使得程序能够以统一的方式处理各种具体类型，极大地提高了代码的灵活性和可扩展性。

1.5.15.1. 从问题出发

让我们先看一个没有使用虚函数的例子，理解为什么需要多态：

```

class Robot {
public:
    Robot(const std::string& name) : name(name) {}

    void performAction() {
        std::cout << name << " 执行默认动作" << std::endl;
    }

protected:
    std::string name;
};

class Infantry : public Robot {
public:
    Infantry(const std::string& name) : Robot(name) {}

    void performAction() { // 重新定义了 performAction
        std::cout << name << " 发射弹丸" << std::endl;
    }
}

```

```

};

class Engineer : public Robot {
public:
    Engineer(const std::string& name) : Robot(name) {}

    void performAction() { // 重新定义了 performAction
        std::cout << name << " 进行修理" << std::endl;
    }
};

void commandRobot(Robot& robot) {
    robot.performAction(); // 调用哪个版本?
}

int main() {
    Infantry infantry("步兵1号");
    Engineer engineer("工程1号");

    commandRobot(infantry); // 输出: 步兵1号 执行默认动作
    commandRobot(engineer); // 输出: 工程1号 执行默认动作

    return 0;
}

```

尽管 Infantry 和 Engineer 都重新定义了 performAction(), 但通过基类引用调用时, 执行的始终是基类 Robot 的版本。这是因为编译器在编译时根据引用的静态类型 (Robot&) 决定调用哪个函数, 而非根据实际对象的类型。这种绑定方式称为静态绑定或早绑定 (static/early binding)。

我们真正需要的是: 让 commandRobot 函数根据传入对象的实际类型调用相应的 performAction() 版本。这就是动态绑定或晚绑定 (dynamic/late binding), 需要通过虚函数来实现。

1.5.15.2. 虚函数

将基类中的成员函数声明为虚函数, 只需在函数声明前加上 `virtual` 关键字:

```

class Robot {
public:
    Robot(const std::string& name) : name(name) {}

    virtual void performAction() { // 声明为虚函数
        std::cout << name << " 执行默认动作" << std::endl;
    }
};

protected:
    std::string name;
};

class Infantry : public Robot {
public:
    Infantry(const std::string& name) : Robot(name) {}

    void performAction() override { // 覆盖基类虚函数
        std::cout << name << " 发射弹丸" << std::endl;
    }
};

```

```

};

class Engineer : public Robot {
public:
    Engineer(const std::string& name) : Robot(name) {}

    void performAction() override { // 覆盖基类虚函数
        std::cout << name << " 进行修理" << std::endl;
    }
};

void commandRobot(Robot& robot) {
    robot.performAction(); // 现在会根据实际类型调用
}

int main() {
    Infantry infantry("步兵1号");
    Engineer engineer("工程1号");

    commandRobot(infantry); // 输出: 步兵1号 发射弹丸
    commandRobot(engineer); // 输出: 工程1号 进行修理

    return 0;
}

```

现在，`commandRobot` 函数会根据传入对象的实际类型调用相应的 `performAction()` 版本。这就是多态的魔力：同一段代码 (`robot.performAction()`) 作用于不同对象时产生不同的行为。

派生类中重新定义虚函数称为覆盖 (override)。`override` 关键字是 C++11 引入的，它告诉编译器这个函数意图覆盖基类的虚函数。如果基类中没有匹配的虚函数 (比如函数名拼写错误或参数列表不匹配)，编译器会报错，帮助我们捕获潜在的 bug：

```

class Infantry : public Robot {
public:
    // 错误示例: 函数名拼写错误
    void preformAction() override { // 编译错误! 基类没有 preformAction
        std::cout << "发射" << std::endl;
    }

    // 错误示例: 参数列表不匹配
    void performAction(int times) override { // 编译错误! 签名不匹配
        std::cout << "发射 " << times << " 次" << std::endl;
    }
};

```

虽然 `override` 不是必需的，但强烈建议在所有覆盖虚函数时使用它，这是现代 C++ 的最佳实践。

1.5.15.3. 动态绑定的工作原理

虚函数的多态行为是如何实现的？编译器通过虚函数表 (virtual table, 简称 vtable) 机制来实现动态绑定。

当一个类包含虚函数时，编译器会为该类创建一个虚函数表，其中存储了该类所有虚函数的地址。每个包含虚函数的对象内部都有一个隐藏的指针（通常称为 vptr），指向其所属类的虚函数表。

当通过基类指针或引用调用虚函数时，程序会：

1. 通过对对象的 vptr 找到虚函数表
2. 在虚函数表中查找要调用的函数地址
3. 调用该地址处的函数

由于不同类型的对象有不同的虚函数表，即使通过相同的基类指针访问，也会调用到各自类型对应的函数版本。

```
// 概念性地展示 vtable (实际实现由编译器处理)

// Robot 的 vtable:
// [0] -> Robot::performAction()

// Infantry 的 vtable:
// [0] -> Infantry::performAction() // 覆盖了基类版本

// Engineer 的 vtable:
// [0] -> Engineer::performAction() // 覆盖了基类版本
```

这种机制有一些性能影响：每个多态对象需要额外存储一个 vptr（通常 8 字节），每次虚函数调用需要一次间接寻址。在大多数应用中这点开销可以忽略不计，但在极端性能敏感的场景（如每秒调用百万次的紧密循环）可能需要考虑。

需要特别注意的是，动态绑定只在通过指针或引用调用虚函数时发生。如果通过对对象直接调用，或者在构造函数/析构函数中调用虚函数，使用的是静态绑定：

```
Infantry infantry("步兵");

infantry.performAction(); // 直接调用，静态绑定，调用 Infantry::performAction

Robot robot = infantry; // 对象切片！
robot.performAction(); // 静态绑定，调用 Robot::performAction

Robot& ref = infantry;
ref.performAction(); // 动态绑定，调用 Infantry::performAction
```

1.5.15.4. 纯虚函数与抽象类

有时候，基类中的某个虚函数没有合理的默认实现——它的存在只是为了定义接口，具体实现必须由派生类提供。这种情况下可以将虚函数声明为纯虚函数（pure virtual function），通过在声明末尾加上 = 0 来实现：

```
class Robot {
public:
    Robot(const std::string& name) : name(name) {}
    virtual ~Robot() = default;

    // 纯虚函数：没有默认实现，派生类必须覆盖
    virtual void performAction() = 0;

    // 普通虚函数：有默认实现，派生类可以选择覆盖
};
```

```

    virtual void reportStatus() {
        std::cout << name << " 状态正常" << std::endl;
    }

protected:
    std::string name;
};

```

包含纯虚函数的类称为抽象类 (abstract class)。抽象类不能被实例化——只能作为基类使用，由派生类提供纯虚函数的实现：

```

int main() {
    // Robot robot("测试"); // 编译错误！Robot 是抽象类，不能实例化

    Infantry infantry("步兵"); // 正确，Infantry 实现了所有纯虚函数
    Robot* ptr = &infantry; // 可以使用抽象类的指针
    ptr->performAction(); // 调用 Infantry::performAction

    return 0;
}

```

如果派生类没有实现基类的所有纯虚函数，那么派生类也是抽象类：

```

class SpecialRobot : public Robot {
public:
    SpecialRobot(const std::string& name) : Robot(name) {}
    // 没有实现 performAction()，所以 SpecialRobot 也是抽象类
};

// SpecialRobot sr("特殊"); // 编译错误！

```

抽象类的典型用途是定义接口。在 RoboMaster 开发中，可以定义各种抽象接口：

```

// 可射击接口
class Shootable {
public:
    virtual ~Shootable() = default;
    virtual void shoot() = 0;
    virtual int getAmmo() const = 0;
    virtual void reload(int amount) = 0;
};

// 可移动接口
class Movable {
public:
    virtual ~Movable() = default;
    virtual void move(double x, double y) = 0;
    virtual void stop() = 0;
    virtual double getSpeed() const = 0;
};

// 步兵机器人实现多个接口
class Infantry : public Robot, public Shootable, public Movable {
public:
    Infantry(const std::string& name) : Robot(name), ammo(300), speed(0) {}

    // 实现 Shootable 接口
    void shoot() override {
        if (ammo > 0) {

```

```

        ammo--;
        std::cout << name << " 开火! " << std::endl;
    }
}

int getAmmo() const override { return ammo; }
void reload(int amount) override { ammo += amount; }

// 实现 Movable 接口
void move(double x, double y) override {
    speed = 5.0;
    std::cout << name << " 移动到 (" << x << ", " << y << ")" << std::endl;
}
void stop() override { speed = 0; }
double getSpeed() const override { return speed; }

// 实现 Robot 的纯虚函数
void performAction() override { shoot(); }

private:
    int ammo;
    double speed;
};

```

通过纯虚函数定义接口，可以编写不依赖具体实现的通用代码：

```

void fireAll(std::vector<Shootable*>& shooters) {
    for (auto* shooter : shooters) {
        shooter->shoot();
    }
}

void moveSquad(std::vector<Movable*>& units, double x, double y) {
    for (auto* unit : units) {
        unit->move(x, y);
    }
}

```

这些函数可以处理任何实现了相应接口的对象，而无需知道它们的具体类型。

1.5.15.5. 虚析构函数

当通过基类指针删除派生类对象时，如果基类析构函数不是虚函数，只有基类的析构函数会被调用，派生类的析构函数不会执行。这可能导致资源泄漏：

```

class Base {
public:
    Base() { std::cout << "Base 构造" << std::endl; }
    ~Base() { std::cout << "Base 析构" << std::endl; } // 非虚析构函数
};

class Derived : public Base {
public:
    Derived() {
        data = new int[100];
        std::cout << "Derived 构造, 分配内存" << std::endl;
    }
    ~Derived() {
        delete[] data;
    }
}

```

```

        std::cout << "Derived 析构, 释放内存" << std::endl;
    }
private:
    int* data;
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // 只调用 Base::~Base(), Derived::~Derived() 不会被调用!
    // 内存泄漏!
    return 0;
}

```

输出：

```

Base 构造
Derived 构造, 分配内存
Base 析构

```

Derived 的析构函数没有被调用， data 指向的内存泄漏了。

解决方案是将基类析构函数声明为虚函数：

```

class Base {
public:
    Base() { std::cout << "Base 构造" << std::endl; }
    virtual ~Base() { std::cout << "Base 析构" << std::endl; } // 虚析构函数
};

```

现在输出变为：

```

Base 构造
Derived 构造, 分配内存
Derived 析构, 释放内存
Base 析构

```

派生类和基类的析构函数都被正确调用， 资源被正确释放。

这引出了一条重要的规则：如果一个类可能被继承（特别是可能通过基类指针删除派生类对象），它的析构函数应该是虚函数。更简单的经验法则是：如果类中有任何虚函数，析构函数也应该是虚函数。

C++11 的 `override` 关键字也可以用于析构函数， 尽管析构函数的覆盖是隐式的：

```

class Base {
public:
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    ~Derived() override = default; // 显式标记覆盖
};

```

1.5.15.6. final 关键字

C++11 引入的 `final` 关键字可以阻止类被继承或虚函数被覆盖。

将类标记为 `final` 表示该类不能被继承：

```

class FinalRobot final : public Robot {
public:

```

```

FinalRobot() : Robot("终极机器人") {}
void performAction() override {
    std::cout << "终极攻击! " << std::endl;
}
};

// class SuperRobot : public FinalRobot { }; // 编译错误! FinalRobot 是 final 的
将虚函数标记为 final 表示该函数不能在派生类中被覆盖:

```

```

class Infantry : public Robot {
public:
    Infantry(const std::string& name) : Robot(name) {}

    void performAction() override final { // 这个实现是最终的
        std::cout << name << " 发射弹丸" << std::endl;
    }
};

class EliteInfantry : public Infantry {
public:
    EliteInfantry(const std::string& name) : Infantry(name) {}

    // void performAction() override {} // 编译错误! Infantry::performAction 是
final 的
};

```

`final` 的用途包括：防止意外的继承或覆盖、表达设计意图、以及在某些情况下帮助编译器进行优化（编译器知道虚函数不会被进一步覆盖，可能可以去虚拟化）。

1.5.15.7. 多态的实际应用

多态在实际开发中有广泛的应用。让我们看几个 RoboMaster 相关的例子。

第一个例子是统一的目标处理系统。不同类型的目标（装甲板、能量机关、基地）需要不同的处理方式，但可以通过统一的接口进行管理：

```

class Target {
public:
    virtual ~Target() = default;

    virtual void track() = 0; // 追踪目标
    virtual double getPriority() const = 0; // 获取优先级
    virtual bool isValid() const = 0; // 检查是否有效

    virtual void render(cv::Mat& frame) const = 0; // 在图像上绘制
};

class ArmorPlate : public Target {
public:
    ArmorPlate(const cv::Rect& bbox, int color)
        : boundingBox(bbox), armorColor(color), lastSeen(0) {}

    void track() override {
        // 装甲板追踪算法
        std::cout << "追踪装甲板" << std::endl;
    }
};

```

```

        double getPriority() const override {
            // 根据距离、角度等计算优先级
            return 1.0 / (boundingBox.area() + 1);
        }

        bool isValid() const override {
            return boundingBox.area() > 100;
        }

        void render(cv::Mat& frame) const override {
            cv::rectangle(frame, boundingBox, cv::Scalar(0, 255, 0), 2);
        }
    }

private:
    cv::Rect boundingBox;
    int armorColor;
    int64_t lastSeen;
};

class EnergyMechanism : public Target {
public:
    void track() override {
        // 能量机关追踪算法，与装甲板不同
        std::cout << "追踪能量机关" << std::endl;
    }

    double getPriority() const override {
        return 2.0; // 能量机关优先级更高
    }

    bool isValid() const override {
        return activated;
    }

    void render(cv::Mat& frame) const override {
        // 绘制能量机关的特殊标记
    }
}

private:
    bool activated = false;
};

// 统一处理所有目标
void processTargets(std::vector<std::unique_ptr<Target>>& targets) {
    // 移除无效目标
    targets.erase(
        std::remove_if(targets.begin(), targets.end(),
                      [] (const auto& t) { return !t->isValid(); }),
        targets.end()
    );

    // 按优先级排序
    std::sort(targets.begin(), targets.end(),
              [] (const auto& a, const auto& b) {
                  return a->getPriority() > b->getPriority();
    });
}

```

```

    });

    // 追踪最高优先级目标
    if (!targets.empty()) {
        targets[0]->track();
    }
}

```

第二个例子是控制器系统。不同的控制算法（PID、模糊控制、模型预测控制）可以通过统一接口使用：

```

class Controller {
public:
    virtual ~Controller() = default;

    virtual double compute(double setpoint, double measurement, double dt) = 0;
    virtual void reset() = 0;
    virtual std::string getName() const = 0;
};

class PIDController : public Controller {
public:
    PIDController(double kp, double ki, double kd)
        : kp(kp), ki(ki), kd(kd), integral(0), lastError(0) {}

    double compute(double setpoint, double measurement, double dt) override {
        double error = setpoint - measurement;
        integral += error * dt;
        double derivative = (error - lastError) / dt;
        lastError = error;
        return kp * error + ki * integral + kd * derivative;
    }

    void reset() override {
        integral = 0;
        lastError = 0;
    }

    std::string getName() const override {
        return "PID Controller";
    }
}

private:
    double kp, ki, kd;
    double integral, lastError;
};

class FuzzyController : public Controller {
public:
    double compute(double setpoint, double measurement, double dt) override {
        // 模糊控制算法
        return 0; // 简化
    }

    void reset() override { }
}

```

```

        std::string getName() const override {
            return "Fuzzy Controller";
        }
    };

// 使用控制器的代码不需要知道具体类型
class MotorDriver {
public:
    MotorDriver(std::unique_ptr<Controller> ctrl)
        : controller(std::move(ctrl)) {}

    void setController(std::unique_ptr<Controller> ctrl) {
        controller = std::move(ctrl);
        controller->reset();
        std::cout << "切换到 " << controller->getName() << std::endl;
    }

    void update(double targetSpeed, double currentSpeed, double dt) {
        double output = controller->compute(targetSpeed, currentSpeed, dt);
        applyOutput(output);
    }

private:
    void applyOutput(double output) {
        // 应用控制输出到电机
    }

    std::unique_ptr<Controller> controller;
};

```

这种设计的优势在于: `MotorDriver` 可以使用任何类型的控制器, 甚至可以在运行时切换控制器, 而无需修改 `MotorDriver` 的代码。添加新的控制算法只需要创建新的派生类, 现有代码完全不受影响。

1.5.15.8. 避免常见陷阱

使用多态时需要注意一些常见的陷阱。

第一个陷阱是在构造函数和析构函数中调用虚函数。在这两个函数中, 虚函数机制不会正常工作——调用的总是当前类的版本, 而非派生类的版本:

```

class Base {
public:
    Base() {
        initialize(); // 危险! 总是调用 Base::initialize()
    }
    virtual void initialize() {
        std::cout << "Base 初始化" << std::endl;
    }
};

class Derived : public Base {
public:
    Derived() : Base() {}
    void initialize() override {
        std::cout << "Derived 初始化" << std::endl;
    }
};

```

```

        }
    };

int main() {
    Derived d; // 输出 "Base 初始化", 而非 "Derived 初始化"
    return 0;
}

```

这是因为在 Base 的构造函数执行时，对象的 Derived 部分尚未构造，将其当作 Derived 对象处理是不安全的。解决方案是避免在构造函数中调用虚函数，或者使用两阶段初始化。

第二个陷阱是忘记将析构函数声明为虚函数。如前所述，这会导致通过基类指针删除派生类对象时资源泄漏。

第三个陷阱是参数类型不匹配导致隐藏而非覆盖：

```

class Base {
public:
    virtual void process(int x) {
        std::cout << "Base::process(int)" << std::endl;
    }
};

class Derived : public Base {
public:
    void process(double x) { // 注意：参数类型不同，这是隐藏，不是覆盖！
        std::cout << "Derived::process(double)" << std::endl;
    }
};

int main() {
    Derived d;
    Base* ptr = &d;
    ptr->process(42); // 调用 Base::process(int)，不是 Derived::process(double)
    return 0;
}

```

使用 `override` 关键字可以避免这个错误——如果签名不匹配，编译器会报错。

1.5.15.9. 性能考虑

虚函数调用比普通函数调用略慢，因为需要通过虚函数表间接调用。在大多数应用中，这点开销可以忽略不计。但在性能极其敏感的代码路径中（如每帧执行数百万次的图像处理内循环），可能需要考虑：

使用 `final` 关键字帮助编译器去虚拟化。如果编译器能确定虚函数不会被进一步覆盖，可能会将虚调用优化为直接调用。

考虑使用模板实现编译时多态（静态多态），它没有虚函数的运行时开销。

在热点代码中避免不必要的虚函数调用，可以将结果缓存或使用其他设计。

不过，在优化之前应当先进行性能分析，确认虚函数确实是瓶颈。过早优化会增加代码复杂性，得不偿失。

多态是面向对象编程的精髓。它使得代码能够以抽象的方式工作，处理“概念上的对象”而非“具体的对象”。通过虚函数和继承，我们可以编写灵活、可扩展的系统——新增功能只需添加新

的派生类，而无需修改现有代码。这正是开闭原则（对扩展开放，对修改关闭）的体现。下一节将学习运算符重载，它允许我们为自定义类型定义运算符的行为，使得自定义类型能够像内置类型一样自然地使用。

1.5.16. 运算符重载

在前面的章节中，我们学习了如何定义类，如何控制对象的生命周期、拷贝行为，以及如何通过继承和多态建立类型层次结构。现在，让我们思考一个问题：对于自定义类型，如何让它们像内置类型一样自然地使用运算符？

考虑一个表示三维向量的类。我们当然可以定义 `add`、`subtract`、`multiply` 等成员函数来实现向量运算，但这样的代码读起来并不自然：

```
Vector3D result = v1.add(v2.multiply(scalar)); // 不够直观
```

如果能像使用内置类型那样使用运算符，代码会清晰得多：

```
Vector3D result = v1 + v2 * scalar; // 直观、自然
```

C++ 的运算符重载（operator overloading）正是为此设计的。它允许我们为自定义类型定义运算符的行为，使得自定义类型能够融入语言的表达体系，与内置类型一样自然地参与运算。这不仅提高了代码的可读性，也使得泛型编程成为可能——同样的算法可以同时作用于内置类型和自定义类型。

1.5.16.1. 运算符重载的基本语法

运算符重载本质上是定义一个特殊名称的函数，函数名由 `operator` 关键字后跟要重载的运算符组成。例如，重载加法运算符的函数名是 `operator+`，重载等于运算符的函数名是 `operator==`。

运算符可以作为成员函数或非成员函数重载。作为成员函数时，左操作数是调用对象本身；作为非成员函数时，所有操作数都通过参数传递。

让我们从一个简单的例子开始——为三维向量类重载加法运算符：

```
class Vector3D {
public:
    Vector3D(double x = 0, double y = 0, double z = 0)
        : x(x), y(y), z(z) {}

    // 作为成员函数重载 +
    Vector3D operator+(const Vector3D& other) const {
        return Vector3D(x + other.x, y + other.y, z + other.z);
    }

    double getX() const { return x; }
    double getY() const { return y; }
    double getZ() const { return z; }

private:
    double x, y, z;
};

int main() {
    Vector3D v1(1.0, 2.0, 3.0);
    Vector3D v2(4.0, 5.0, 6.0);
```

```

    Vector3D v3 = v1 + v2; // 调用 v1.operator+(v2)

    std::cout << "v3 = (" << v3.getX() << ", " << v3.getY() << ", " <<
    v3.getZ() << ")" << std::endl;
    // 输出: v3 = (5, 7, 9)

    return 0;
}

```

当编译器遇到 `v1 + v2` 时，它会将其转换为 `v1.operator+(v2)` 的函数调用。成员函数版本的 `operator+` 接受一个参数（右操作数），左操作数就是调用该函数的对象 (`*this`)。

注意 `operator+` 被声明为 `const` 成员函数，因为加法运算不应该修改任何一个操作数。它返回一个新的 `Vector3D` 对象，包含运算结果。这与内置类型的加法行为一致——`a + b` 不会修改 `a` 或 `b`。

1.5.16.2. 算术运算符

算术运算符是最常重载的一类运算符。除了加法，我们还可以重载减法、乘法、除法等。以下是 `Vector3D` 类的完整算术运算符实现：

```

class Vector3D {
public:
    Vector3D(double x = 0, double y = 0, double z = 0)
        : x(x), y(y), z(z) {}

    // 向量加法
    Vector3D operator+(const Vector3D& other) const {
        return Vector3D(x + other.x, y + other.y, z + other.z);
    }

    // 向量减法
    Vector3D operator-(const Vector3D& other) const {
        return Vector3D(x - other.x, y - other.y, z - other.z);
    }

    // 标量乘法（向量 * 标量）
    Vector3D operator*(double scalar) const {
        return Vector3D(x * scalar, y * scalar, z * scalar);
    }

    // 标量除法
    Vector3D operator/(double scalar) const {
        return Vector3D(x / scalar, y / scalar, z / scalar);
    }

    // 点积（使用 * 运算符，两个向量相乘）
    double operator*(const Vector3D& other) const {
        return x * other.x + y * other.y + z * other.z;
    }

    // 一元负号（取反）
    Vector3D operator-() const {
        return Vector3D(-x, -y, -z);
    }
}

```

```

private:
    double x, y, z;
};

int main() {
    Vector3D v1(1.0, 2.0, 3.0);
    Vector3D v2(4.0, 5.0, 6.0);

    Vector3D sum = v1 + v2;           // (5, 7, 9)
    Vector3D diff = v1 - v2;         // (-3, -3, -3)
    Vector3D scaled = v1 * 2.0;      // (2, 4, 6)
    Vector3D divided = v2 / 2.0;     // (2, 2.5, 3)
    double dot = v1 * v2;           // 1*4 + 2*5 + 3*6 = 32
    Vector3D negated = -v1;          // (-1, -2, -3)

    return 0;
}

```

注意一元负号运算符 `operator-()` 不接受参数，它作用于单个操作数（调用对象本身）。这与二元减法运算符 `operator-(const Vector3D&)` 是不同的重载。

上面的标量乘法 `v1 * 2.0` 可以工作，但 `2.0 * v1` 却不行——因为成员函数版本要求左操作数是 `Vector3D` 类型。要支持 `2.0 * v1` 这种写法，需要定义非成员函数版本的运算符。

1.5.16.3. 友元函数

非成员函数不能直接访问类的私有成员。如果运算符函数需要访问私有成员，有两种选择：通过公开的 `getter` 函数访问，或者将运算符函数声明为类的友元（friend）。

友元函数不是类的成员，但被授予了访问类私有成员的权限。在类定义内部使用 `friend` 关键字声明友元：

```

class Vector3D {
public:
    Vector3D(double x = 0, double y = 0, double z = 0)
        : x(x), y(y), z(z) {}

    // 成员函数版本：向量 * 标量
    Vector3D operator*(double scalar) const {
        return Vector3D(x * scalar, y * scalar, z * scalar);
    }

    // 友元函数声明：标量 * 向量
    friend Vector3D operator*(double scalar, const Vector3D& v);

private:
    double x, y, z;
};

// 友元函数定义（在类外部）
Vector3D operator*(double scalar, const Vector3D& v) {
    return Vector3D(v.x * scalar, v.y * scalar, v.z * scalar); // 可以访问私有
成员
}

```

```

int main() {
    Vector3D v(1.0, 2.0, 3.0);

    Vector3D r1 = v * 2.0;      // 调用成员函数 v.operator*(2.0)
    Vector3D r2 = 2.0 * v;      // 调用友元函数 operator*(2.0, v)

    return 0;
}

```

友元声明可以放在类的任何访问区域（public、private 或 protected），访问说明符对友元声明没有影响——友元总是可以访问所有成员。

一种常见的做法是在类内部直接定义友元函数，这样可以将声明和定义放在一起：

```

class Vector3D {
public:
    // ...

    // 在类内部定义友元函数
    friend Vector3D operator*(double scalar, const Vector3D& v) {
        return Vector3D(v.x * scalar, v.y * scalar, v.z * scalar);
    }

private:
    double x, y, z;
};

```

虽然这个函数定义在类的花括号内，但它仍然是非成员函数。friend 关键字既声明了友元关系，又允许在此处定义函数。

1.5.16.4. 比较运算符

比较运算符用于判断两个对象之间的关系。最常用的是相等（==）和不等（!=）运算符：

```

class Vector3D {
public:
    Vector3D(double x = 0, double y = 0, double z = 0)
        : x(x), y(y), z(z) {}

    bool operator==(const Vector3D& other) const {
        const double epsilon = 1e-9;
        return std::abs(x - other.x) < epsilon &&
               std::abs(y - other.y) < epsilon &&
               std::abs(z - other.z) < epsilon;
    }

    bool operator!=(const Vector3D& other) const {
        return !(*this == other); // 利用已定义的 ==
    }

private:
    double x, y, z;
};

```

对于浮点数比较，直接使用 == 可能因精度问题产生错误结果，因此使用一个小的容差值（epsilon）进行比较。operator!= 通常可以基于 operator== 实现，避免代码重复。

对于需要排序的类型，还需要重载关系运算符。例如，表示分数的类：

```

class Fraction {
public:
    Fraction(int num = 0, int den = 1) : numerator(num), denominator(den) {
        normalize();
    }

    bool operator==(const Fraction& other) const {
        return numerator == other.numerator && denominator ==
other.denominator;
    }

    bool operator!=(const Fraction& other) const {
        return !(*this == other);
    }

    bool operator<(const Fraction& other) const {
        return numerator * other.denominator < other.numerator * denominator;
    }

    bool operator>(const Fraction& other) const {
        return other < *this;
    }

    bool operator<=(const Fraction& other) const {
        return !(other < *this);
    }

    bool operator>=(const Fraction& other) const {
        return !(*this < other);
    }
};

private:
    void normalize(); // 约分并处理符号
    int numerator;
    int denominator;
};

```

C++20 引入了三路比较运算符 ($<=$ ，也称为“太空船运算符”), 可以一次性定义所有比较运算符。但在 C++20 之前, 通常需要手动实现六个比较运算符, 其中 $=$ 和 $<$ 是基础, 其他四个可以基于它们实现。

1.5.16.5. 复合赋值运算符

复合赋值运算符 (如 $+=$ 、 $-=$ 、 $*=$) 结合了算术运算和赋值。它们修改左操作数并返回其引用:

```

class Vector3D {
public:
    // ...

    Vector3D& operator+=(const Vector3D& other) {
        x += other.x;
        y += other.y;
        z += other.z;
        return *this;
    }
}

```

```

Vector3D& operator-=(const Vector3D& other) {
    x -= other.x;
    y -= other.y;
    z -= other.z;
    return *this;
}

Vector3D& operator*=(double scalar) {
    x *= scalar;
    y *= scalar;
    z *= scalar;
    return *this;
}

Vector3D& operator/=(double scalar) {
    x /= scalar;
    y /= scalar;
    z /= scalar;
    return *this;
}

private:
    double x, y, z;
};

```

返回 `*this` 的引用允许链式操作: `v1 += v2 += v3` (从右向左执行)。

一种良好的实践是基于复合赋值运算符实现普通算术运算符, 这样可以避免代码重复:

```

class Vector3D {
public:
    // 复合赋值运算符 (修改自身)
    Vector3D& operator+=(const Vector3D& other) {
        x += other.x;
        y += other.y;
        z += other.z;
        return *this;
    }

    // 基于 += 实现 +
    Vector3D operator+(const Vector3D& other) const {
        Vector3D result = *this; // 复制当前对象
        result += other; // 使用 +=
        return result;
    }

    // 或者使用更简洁的写法
    Vector3D operator-(const Vector3D& other) const {
        return Vector3D(*this) -= other;
    }

private:
    double x, y, z;
};

```

这种方式确保了 `+` 和 `+=` 的行为一致, 修改一处逻辑时另一处自动更新。

1.5.16.6. 下标运算符

下标运算符 `operator[]` 使对象能够像数组一样通过索引访问元素。它必须作为成员函数定义，通常需要提供 `const` 和非 `const` 两个版本：

```
class Matrix {
public:
    Matrix(size_t rows, size_t cols)
        : rows(rows), cols(cols), data(rows * cols, 0.0) {}

    // 非 const 版本：可以修改元素
    double& operator()(size_t row, size_t col) {
        return data[row * cols + col];
    }

    // const 版本：只读访问
    const double& operator()(size_t row, size_t col) const {
        return data[row * cols + col];
    }

    size_t getRows() const { return rows; }
    size_t getCols() const { return cols; }

private:
    size_t rows, cols;
    std::vector<double> data;
};

int main() {
    Matrix m(3, 3);

    m(0, 0) = 1.0; // 使用非 const 版本
    m(1, 1) = 1.0;
    m(2, 2) = 1.0;

    const Matrix& cm = m;
    double val = cm(0, 0); // 使用 const 版本
    // cm(0, 0) = 2.0;      // 错误！const 版本返回 const 引用

    return 0;
}
```

注意这里使用了 `operator()` 而非 `operator[]`，因为 `operator[]` 只能接受一个参数，而矩阵访问需要行和列两个索引。使用函数调用运算符 `operator()` 可以接受任意数量的参数。

对于一维容器，可以使用标准的 `operator[]`：

```
class DynamicArray {
public:
    DynamicArray(size_t size) : data(size) {}

    double& operator[](size_t index) {
        return data[index];
    }

    const double& operator[](size_t index) const {
        return data[index];
```

```

    }

    size_t size() const { return data.size(); }

private:
    std::vector<double> data;
};

```

是否进行边界检查是一个设计选择。标准库的 `vector::operator[]` 不检查边界（为了性能），而 `vector::at()` 会检查并在越界时抛出异常。自定义类型可以根据需求选择合适的策略。

1.5.16.7. 输入输出运算符

流插入运算符 `<<` 和流提取运算符 `>>` 使得自定义类型可以与 `iostream` 无缝配合。由于左操作数是流对象（`std::ostream` 或 `std::istream`），这些运算符必须作为非成员函数实现：

```

class Vector3D {
public:
    Vector3D(double x = 0, double y = 0, double z = 0)
        : x(x), y(y), z(z) {}

    // 声明友元
    friend std::ostream& operator<<(std::ostream& os, const Vector3D& v);
    friend std::istream& operator>>(std::istream& is, Vector3D& v);

private:
    double x, y, z;
};

// 输出运算符
std::ostream& operator<<(std::ostream& os, const Vector3D& v) {
    os << "(" << v.x << ", " << v.y << ", " << v.z << ")";
    return os;
}

// 输入运算符
std::istream& operator>>(std::istream& is, Vector3D& v) {
    is >> v.x >> v.y >> v.z;
    return is;
}

int main() {
    Vector3D v1(1.0, 2.0, 3.0);
    std::cout << "v1 = " << v1 << std::endl; // 输出: v1 = (1, 2, 3)

    Vector3D v2;
    std::cout << "输入向量 (x y z): ";
    std::cin >> v2;
    std::cout << "你输入的向量是: " << v2 << std::endl;

    return 0;
}

```

输出运算符接受 `const` 引用参数，因为输出不应修改对象；输入运算符接受非 `const` 引用，因为需要修改对象。两者都返回流的引用，以支持链式调用：`std::cout << v1 << " and " << v2`。

输入运算符还应该处理输入失败的情况。更健壮的实现会检查流状态：

```

std::istream& operator>>(std::istream& is, Vector3D& v) {
    double x, y, z;
    if (is >> x >> y >> z) {
        v = Vector3D(x, y, z);
    }
    // 如果输入失败，流的失败状态会保持，v 不变
    return is;
}

```

1.5.16.8. 递增递减运算符

递增 `++` 和递减 `--` 运算符有前置和后置两种形式，它们的行为不同：前置返回修改后的值，后置返回修改前的值。为了区分这两种形式，C++ 使用一个哑参数（dummy parameter）：

```

class Counter {
public:
    Counter(int value = 0) : value(value) {}

    // 前置递增: ++c
    Counter& operator++() {
        ++value;
        return *this;
    }

    // 后置递增: c++
    Counter operator++(int) { // int 参数仅用于区分，不使用
        Counter old = *this; // 保存旧值
        ++(*this); // 使用前置递增
        return old; // 返回旧值
    }

    // 前置递减: --c
    Counter& operator--() {
        --value;
        return *this;
    }

    // 后置递减: c--
    Counter operator--(int) {
        Counter old = *this;
        --(*this);
        return old;
    }

    int getValue() const { return value; }

private:
    int value;
};

int main() {
    Counter c(5);

    std::cout << ++c.getValue() << std::endl; // 6 (先加后用)
    std::cout << c++.getValue() << std::endl; // 6 (先用后加，但这里有问题...)
}

```

```

    // 正确的测试方式
    Counter c1(5);
    Counter c2 = ++c1; // c1 = 6, c2 = 6
    Counter c3 = c1++; // c1 = 7, c3 = 6 (c3 是旧值)

    return 0;
}

```

前置版本返回引用，效率更高；后置版本必须创建副本保存旧值，然后返回这个副本。这就是为什么在不需要旧值的情况下，`++i` 比 `i++` 更推荐使用。

后置版本通常基于前置版本实现，确保行为一致。

1.5.16.9. 类型转换运算符

类型转换运算符允许对对象隐式或显式转换为其他类型。转换运算符的名称是 `operator` 后跟目标类型，没有返回类型声明（因为返回类型就是运算符名称中的类型）：

```

class Fraction {
public:
    Fraction(int num = 0, int den = 1) : numerator(num), denominator(den) {}

    // 转换为 double
    operator double() const {
        return static_cast<double>(numerator) / denominator;
    }

    // 转换为 bool (判断是否为零)
    explicit operator bool() const {
        return numerator != 0;
    }

private:
    int numerator;
    int denominator;
};

int main() {
    Fraction f(3, 4);

    double d = f; // 隐式转换, d = 0.75
    double e = static_cast<double>(f); // 显式转换

    if (f) { // explicit 转换在条件语句中可以隐式使用
        std::cout << "分数非零" << std::endl;
    }

    // bool b = f; // 错误! explicit 禁止了这种隐式转换
    bool b = static_cast<bool>(f); // 正确, 显式转换

    return 0;
}

```

`explicit` 关键字防止意外的隐式转换，这是 C++11 之后的推荐做法。不加 `explicit` 的转换运算符可能导致意想不到的类型转换，引发难以调试的问题。

除了定义从自定义类型到其他类型的转换，还可以通过接受单个参数的构造函数定义从其他类型到自定义类型的转换：

```
class Vector3D {
public:
    // 从 double 转换：创建 (d, d, d) 向量
    Vector3D(double d) : x(d), y(d), z(d) {}

    // 阻止隐式转换
    explicit Vector3D(int i) : x(i), y(i), z(i) {}

private:
    double x, y, z;
};

int main() {
    Vector3D v1 = 1.5;    // 隐式转换，调用 Vector3D(double)
    Vector3D v2(3.0);    // 直接构造

    // Vector3D v3 = 42; // 错误！Vector3D(int) 是 explicit 的
    Vector3D v4(42);    // 正确，显式构造

    return 0;
}
```

1.5.16.10. 函数调用运算符

函数调用运算符 `operator()` 使对象可以像函数一样被调用。具有这种能力的对象称为函数对象或仿函数（functor）：

```
class Adder {
public:
    Adder(int increment) : increment(increment) {}

    int operator()(int value) const {
        return value + increment;
    }

private:
    int increment;
};

class DistanceCalculator {
public:
    DistanceCalculator(double x, double y, double z)
        : originX(x), originY(y), originZ(z) {}

    double operator()(double x, double y, double z) const {
        double dx = x - originX;
        double dy = y - originY;
        double dz = z - originZ;
        return std::sqrt(dx*dx + dy*dy + dz*dz);
    }

private:
    double originX, originY, originZ;
```

```

};
```

```

int main() {
    Adder addFive(5);
    std::cout << addFive(10) << std::endl; // 15
    std::cout << addFive(20) << std::endl; // 25

    DistanceCalculator distFromOrigin(0, 0, 0);
    std::cout << distFromOrigin(3, 4, 0) << std::endl; // 5

    return 0;
}

```

函数对象比普通函数更灵活，因为它们可以携带状态（如上例中的 `increment` 和原点坐标）。它们在标准库算法中广泛使用：

```

std::vector<int> numbers = {1, 2, 3, 4, 5};

// 使用函数对象作为谓词
struct IsEven {
    bool operator()(int n) const {
        return n % 2 == 0;
    }
};

auto it = std::find_if(numbers.begin(), numbers.end(), IsEven());

```

```

// 使用函数对象进行变换
Adder addTen(10);
std::transform(numbers.begin(), numbers.end(), numbers.begin(), addTen);
// numbers 现在是 {11, 12, 13, 14, 15}

```

C++11 引入的 lambda 表达式在很多场景下可以替代函数对象，但函数对象在需要复杂状态或多次复用时仍然有用。

1.5.16.11. 运算符重载的原则与限制

并非所有运算符都可以重载。以下运算符不能重载：

- :: 作用域解析运算符
- . 成员访问运算符
- .* 成员指针访问运算符
- ?: 条件运算符
- sizeof 大小运算符
- typeid 类型信息运算符

此外，还有一些重要的设计原则：

不要改变运算符的基本语义。`+` 应该表示某种“相加”的概念，`==` 应该表示某种“相等”的概念。如果 `a + b` 的含义与加法毫无关系，会让代码极其难以理解。

保持与内置类型的一致性。如果 `a + b` 有效，通常 `a += b` 也应该有效且行为一致。如果 `a == b`，那么 `a != b` 应该返回相反的结果。

对于二元运算符，考虑是否需要支持混合类型操作。例如，`Vector3D * double` 和 `double * Vector3D` 都应该有效。

优先使用成员函数还是非成员函数？一般原则是：

- 赋值（=）、下标（[]）、调用（()）、成员访问（->）必须是成员函数
- 复合赋值运算符（+= 等）通常是成员函数
- 算术运算符（+ 等）如果需要左操作数可以隐式转换，应该是非成员函数
- 输入输出运算符（<<、>>）必须是非成员函数

1.5.16.12. 一个完整的例子

让我们综合运用本节的知识，实现一个完整的复数类：

```
#include <iostream>
#include <cmath>

class Complex {
public:
    // 构造函数
    Complex(double real = 0, double imag = 0)
        : real(real), imag(imag) {}

    // Getter
    double getReal() const { return real; }
    double getImag() const { return imag; }

    // 模（绝对值）
    double abs() const {
        return std::sqrt(real * real + imag * imag);
    }

    // 复合赋值运算符
    Complex& operator+=(const Complex& other) {
        real += other.real;
        imag += other.imag;
        return *this;
    }

    Complex& operator-=(const Complex& other) {
        real -= other.real;
        imag -= other.imag;
        return *this;
    }

    Complex& operator*=(const Complex& other) {
        double newReal = real * other.real - imag * other.imag;
        double newImag = real * other.imag + imag * other.real;
        real = newReal;
        imag = newImag;
        return *this;
    }

    Complex& operator/=(const Complex& other) {
        double denom = other.real * other.real + other.imag * other.imag;
        double newReal = (real * other.real + imag * other.imag) / denom;
        double newImag = (imag * other.real - real * other.imag) / denom;
        real = newReal;
        imag = newImag;
    }
}
```

```

        return *this;
    }

    // 一元运算符
    Complex operator-() const {
        return Complex(-real, -imag);
    }

    Complex operator+() const {
        return *this;
    }

    // 比较运算符
    bool operator==(const Complex& other) const {
        const double epsilon = 1e-9;
        return std::abs(real - other.real) < epsilon &&
               std::abs(imag - other.imag) < epsilon;
    }

    bool operator!=(const Complex& other) const {
        return !(*this == other);
    }

    // 友元声明
    friend Complex operator+(const Complex& a, const Complex& b);
    friend Complex operator-(const Complex& a, const Complex& b);
    friend Complex operator*(const Complex& a, const Complex& b);
    friend Complex operator/(const Complex& a, const Complex& b);
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
    friend std::istream& operator>>(std::istream& is, Complex& c);

private:
    double real;
    double imag;
};

// 算术运算符（非成员函数，支持隐式转换）
Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.real + b.real, a.imag + b.imag);
}

Complex operator-(const Complex& a, const Complex& b) {
    return Complex(a.real - b.real, a.imag - b.imag);
}

Complex operator*(const Complex& a, const Complex& b) {
    return Complex(a.real * b.real - a.imag * b.imag,
                  a.real * b.imag + a.imag * b.real);
}

Complex operator/(const Complex& a, const Complex& b) {
    double denom = b.real * b.real + b.imag * b.imag;
    return Complex((a.real * b.real + a.imag * b.imag) / denom,
                  (a.imag * b.real - a.real * b.imag) / denom);
}

```

```

// 输入输出运算符
std::ostream& operator<<(std::ostream& os, const Complex& c) {
    os << c.real;
    if (c.imag >= 0) os << "+";
    os << c.imag << "i";
    return os;
}

std::istream& operator>>(std::istream& is, Complex& c) {
    is >> c.real >> c.imag;
    return is;
}

int main() {
    Complex c1(3, 4);
    Complex c2(1, -2);

    std::cout << "c1 = " << c1 << std::endl; // 3+4i
    std::cout << "c2 = " << c2 << std::endl; // 1-2i
    std::cout << "|c1| = " << c1.abs() << std::endl; // 5

    std::cout << "c1 + c2 = " << (c1 + c2) << std::endl; // 4+2i
    std::cout << "c1 - c2 = " << (c1 - c2) << std::endl; // 2+6i
    std::cout << "c1 * c2 = " << (c1 * c2) << std::endl; // 11-2i
    std::cout << "c1 / c2 = " << (c1 / c2) << std::endl; // -1+2i

    // 与实数混合运算（利用隐式转换）
    Complex c3 = c1 + 5.0; // 5.0 隐式转换为 Complex(5, 0)
    std::cout << "c1 + 5 = " << c3 << std::endl; // 8+4i

    Complex c4 = 2.0 * c1; // 同样利用隐式转换
    std::cout << "2 * c1 = " << c4 << std::endl; // 6+8i

    return 0;
}

```

这个复数类展示了运算符重载的多个方面：算术运算符、复合赋值运算符、比较运算符、一元运算符、输入输出运算符，以及通过友元函数实现的非成员运算符。它像内置类型一样自然地使用，同时保持了良好的封装。

运算符重载是 C++ 强大而灵活的特性。合理使用它可以使代码更加直观和优雅，但过度或不当使用会让代码变得难以理解。关键在于保持运算符的自然语义，让使用者能够凭直觉正确使用你的类型。下一节我们将学习 STL 容器，这些标准库类型正是运算符重载的典范——它们通过重载下标、迭代器运算符等，提供了直观而强大的接口。

1.5.17. STL 容器

在前面的章节中，我们学习了数组——一种存储固定数量同类型元素的基本数据结构。然而，实际开发中常常面临更复杂的需求：元素数量在运行时才能确定、需要频繁在中间位置插入删除、需要快速查找某个元素是否存在、需要按键值对存储数据。如果每次都从头实现这些数据结构，不仅费时费力，还容易出错。

C++ 标准模板库（Standard Template Library, STL）提供了一套精心设计、经过充分测试的容器类，涵盖了最常用的数据结构。这些容器不仅功能强大、性能优异，而且接口统一、易于学习。掌握 STL 容器是每个 C++ 程序员的必备技能，它们将成为你日常开发中最常使用的工具。

1.5.17.1. 容器的分类

STL 容器可以分为三大类：

顺序容器（Sequence Containers）按照元素插入的顺序存储数据，包括 `vector`、`deque`、`list`、`array`、`forward_list` 等。它们的主要区别在于内存布局和各种操作的性能特征。

关联容器（Associative Containers）按照键的顺序存储数据，内部使用平衡二叉树实现，包括 `set`、`map`、`multiset`、`multimap`。它们提供基于键的快速查找，元素按键自动排序。

无序关联容器（Unordered Associative Containers）使用哈希表实现，包括 `unordered_set`、`unordered_map`、`unordered_multiset`、`unordered_multimap`。它们提供平均常数时间的查找，但元素没有特定顺序。

此外还有容器适配器（Container Adapters），如 `stack`、`queue`、`priority_queue`，它们基于其他容器实现，提供特定的访问模式。

让我们从最常用的容器开始，逐一探索它们的特点和用法。

1.5.17.2. `vector`: 动态数组

`std::vector` 是最常用的 STL 容器，可以看作是能够自动增长的数组。它在内存中连续存储元素，支持随机访问，在末尾添加和删除元素非常高效。

使用 `vector` 需要包含 `<vector>` 头文件：

```
#include <vector>
#include <iostream>

int main() {
    // 创建空 vector
    std::vector<int> numbers;

    // 创建包含 5 个元素的 vector, 初始值为 0
    std::vector<int> zeros(5);

    // 创建包含 5 个元素的 vector, 初始值为 10
    std::vector<int> tens(5, 10);

    // 使用初始化列表创建
    std::vector<int> primes = {2, 3, 5, 7, 11};

    // 复制另一个 vector
    std::vector<int> primesCopy = primes;

    return 0;
}
```

`vector` 模板参数指定元素类型。`std::vector<int>` 存储整数，`std::vector<double>` 存储双精度浮点数，`std::vector<std::string>` 存储字符串，`std::vector<Target>` 存储自定义的 Target 对象。

访问元素可以使用下标运算符或 `at()` 方法。下标运算符不检查边界，`at()` 会在越界时抛出异常：

```
std::vector<int> v = {10, 20, 30, 40, 50};

int a = v[0];          // 10, 不检查边界
int b = v.at(1);      // 20, 检查边界

v[2] = 35;            // 修改元素
// v[10] = 100;        // 未定义行为！越界但不报错
// v.at(10) = 100;     // 抛出 std::out_of_range 异常

int first = v.front(); // 第一个元素
int last = v.back();   // 最后一个元素
```

在 `vector` 末尾添加和删除元素是最常用的操作：

```
std::vector<int> v;

v.push_back(10);    // 添加到末尾: {10}
v.push_back(20);    // {10, 20}
v.push_back(30);    // {10, 20, 30}

v.pop_back();        // 移除末尾元素: {10, 20}

// C++11 引入的 emplace_back, 直接在容器内构造对象, 避免拷贝
v.emplace_back(40); // {10, 20, 30, 40}
```

`push_back` 和 `pop_back` 的时间复杂度是摊销常数 (amortized constant)，非常高效。这是因为 `vector` 会预先分配额外的内存空间，只有当空间用完时才需要重新分配。

`vector` 的大小和容量是两个不同的概念。大小 (`size`) 是当前存储的元素数量，容量 (`capacity`) 是不需要重新分配内存就能存储的最大元素数量：

```
std::vector<int> v;

std::cout << "Size: " << v.size() << std::endl;           // 0
std::cout << "Capacity: " << v.capacity() << std::endl; // 可能是 0

v.reserve(100); // 预分配空间, 避免频繁重新分配
std::cout << "After reserve - Size: " << v.size() << std::endl; // 仍然是 0
std::cout << "After reserve - Capacity: " << v.capacity() << std::endl; // 至少 100

for (int i = 0; i < 50; i++) {
    v.push_back(i);
}
std::cout << "After pushing - Size: " << v.size() << std::endl; // 50
std::cout << "After pushing - Capacity: " << v.capacity() << std::endl; // 仍然至少 100

v.shrink_to_fit(); // 释放多余的容量 (非强制)
```

在 RoboMaster 开发中，如果预先知道大致需要多少元素，使用 `reserve()` 预分配空间可以避免多次内存重新分配，提高性能。

遍历 `vector` 有多种方式：

```

std::vector<int> v = {1, 2, 3, 4, 5};

// 方式一：基于索引
for (size_t i = 0; i < v.size(); i++) {
    std::cout << v[i] << " ";
}

// 方式二：范围 for 循环（推荐）
for (int x : v) {
    std::cout << x << " ";
}

// 方式三：使用引用避免拷贝（对于大型元素）
for (const int& x : v) {
    std::cout << x << " ";
}

// 方式四：使用迭代器
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
}

```

在中间位置插入和删除元素需要移动后续所有元素，时间复杂度是 $O(n)$ ：

```

std::vector<int> v = {1, 2, 3, 4, 5};

// 在位置 2（第三个元素前）插入 100
v.insert(v.begin() + 2, 100); // {1, 2, 100, 3, 4, 5}

// 删除位置 3 的元素
v.erase(v.begin() + 3); // {1, 2, 100, 4, 5}

// 删除范围 [begin+1, begin+3)
v.erase(v.begin() + 1, v.begin() + 3); // {1, 4, 5}

// 清空所有元素
v.clear(); // {}
std::cout << "Empty: " << v.empty() << std::endl; // true

```

1.5.17.3. deque：双端队列

`std::deque` (double-ended queue) 支持在两端高效地插入和删除元素。与 `vector` 不同，`deque` 的内存不是完全连续的，它由多个固定大小的内存块组成。

```

#include <deque>

std::deque<int> dq = {2, 3, 4};

// 在两端操作都是高效的
dq.push_front(1); // {1, 2, 3, 4}
dq.push_back(5); // {1, 2, 3, 4, 5}

dq.pop_front(); // {2, 3, 4, 5}
dq.pop_back(); // {2, 3, 4}

// 随机访问

```

```
int x = dq[1];      // 3
int y = dq.at(2);  // 4
```

`deque` 与 `vector` 的主要区别在于：`deque` 在头部插入删除是常数时间，`vector` 则是线性时间；`deque` 的内存不完全连续，不能直接将其数据传给需要连续内存的 C 函数；`deque` 没有 `capacity()` 和 `reserve()` 方法。

在需要频繁在两端操作的场景下，`deque` 比 `vector` 更合适。

1.5.17.4. `list`: 双向链表

`std::list` 是双向链表，每个元素都包含指向前一个和后一个元素的指针。它支持在任意位置常数时间的插入和删除，但不支持随机访问。

```
#include <list>

std::list<int> lst = {1, 2, 3, 4, 5};

// 在两端操作
lst.push_front(0);    // {0, 1, 2, 3, 4, 5}
lst.push_back(6);     // {0, 1, 2, 3, 4, 5, 6}

// 不支持随机访问
// int x = lst[2];    // 错误! list 没有 operator[]

// 必须通过迭代器访问
auto it = lst.begin();
std::advance(it, 2);  // 移动到第三个元素
std::cout << *it << std::endl; // 2

// 在任意位置插入删除都是常数时间（找到位置后）
lst.insert(it, 100); // 在 it 前插入 100
lst.erase(it);       // 删除 it 指向的元素
```

`list` 提供了一些特有的操作：

```
std::list<int> lst1 = {1, 3, 5};
std::list<int> lst2 = {2, 4, 6};

// 合并两个已排序的链表
lst1.merge(lst2); // lst1 = {1, 2, 3, 4, 5, 6}, lst2 = {}

// 排序（list 不能使用 std::sort，必须用成员函数）
std::list<int> unsorted = {5, 2, 8, 1, 9};
unsorted.sort(); // {1, 2, 5, 8, 9}

// 去除连续重复元素
std::list<int> dups = {1, 1, 2, 2, 2, 3, 3};
dups.unique(); // {1, 2, 3}

// 移除特定值
std::list<int> vals = {1, 2, 3, 2, 4, 2};
vals.remove(2); // {1, 3, 4}

// 翻转
std::list<int> rev = {1, 2, 3};
rev.reverse(); // {3, 2, 1}
```

`list` 适用于需要频繁在中间插入删除的场景，但由于缓存不友好（元素在内存中不连续），遍历性能不如 `vector`。

1.5.17.5. `set`: 有序集合

`std::set` 存储唯一元素的有序集合。它基于红黑树实现，查找、插入、删除的时间复杂度都是 $O(\log n)$ 。元素按照比较函数（默认是 `<`）自动排序。

```
#include <set>

std::set<int> s;

// 插入元素
s.insert(30);
s.insert(10);
s.insert(20);
s.insert(10); // 重复元素，不会插入

// 遍历（自动排序）
for (int x : s) {
    std::cout << x << " "; // 10 20 30
}
std::cout << std::endl;

// 查找
auto it = s.find(20);
if (it != s.end()) {
    std::cout << "找到: " << *it << std::endl;
}

// 检查是否存在
if (s.count(20) > 0) { // count 返回 0 或 1
    std::cout << "20 存在" << std::endl;
}

// C++20 提供了更直观的 contains
// if (s.contains(20)) { ... }

// 删除
s.erase(20); // 删除值为 20 的元素
s.erase(s.begin()); // 删除第一个元素

std::cout << "Size: " << s.size() << std::endl;
```

`set` 中的元素一旦插入就不能修改（因为修改可能破坏排序），只能删除后重新插入。

在 RoboMaster 开发中，`set` 可用于存储已处理的目标 ID、维护不重复的事件列表等：

```
std::set<int> processedTargetIds;

void processTarget(const Target& target) {
    // 检查是否已处理过
    if (processedTargetIds.count(target.id) > 0) {
        return; // 已处理，跳过
    }

    // 处理目标...
```

```
    processedTargetIds.insert(target.id);
}
```

1.5.17.6. map：有序键值对

`std::map` 存储键值对 (key-value pairs)，按键排序。每个键唯一，对应一个值。它同样基于红黑树实现，各操作时间复杂度为 $O(\log n)$ 。

```
#include <map>

std::map<std::string, int> ages;

// 插入键值对
ages["Alice"] = 25;
ages["Bob"] = 30;
ages.insert({"Charlie", 35});
ages.insert(std::make_pair("David", 28));

// 访问
std::cout << "Alice's age: " << ages["Alice"] << std::endl; // 25

// 注意：使用 [] 访问不存在的键会自动插入
std::cout << "Eve's age: " << ages["Eve"] << std::endl; // 0 (自动插入)

// 安全访问：使用 at()，键不存在时抛出异常
try {
    int age = ages.at("Frank");
} catch (const std::out_of_range& e) {
    std::cout << "Frank not found" << std::endl;
}

// 检查键是否存在
if (ages.count("Alice") > 0) {
    std::cout << "Alice exists" << std::endl;
}

// 使用 find 查找
auto it = ages.find("Bob");
if (it != ages.end()) {
    std::cout << it->first << " is " << it->second << " years old" <<
    std::endl;
}

// 遍历（按键排序）
for (const auto& pair : ages) {
    std::cout << pair.first << ":" << pair.second << std::endl;
}

// C++17 结构化绑定
for (const auto& [name, age] : ages) {
    std::cout << name << ":" << age << std::endl;
}

// 删除
ages.erase("Eve");
```

`map` 在 RoboMaster 开发中非常有用，例如存储电机 ID 与电机对象的映射、目标 ID 与跟踪状态的映射等：

```
std::map<int, Motor> motors;
mots[1] = Motor(1, 10000.0);
mots[2] = Motor(2, 10000.0);
mots[3] = Motor(3, 8000.0);
mots[4] = Motor(4, 8000.0);

void updateMotor(int id, double speed) {
    auto it = mots.find(id);
    if (it != mots.end()) {
        it->second.setSpeed(speed);
    }
}
```

1.5.17.7. `unordered_set` 和 `unordered_map`

`std::unordered_set` 和 `std::unordered_map` 是基于哈希表的容器，提供平均 $O(1)$ 的查找、插入、删除操作，但元素没有特定顺序。

```
#include <unordered_set>
#include <unordered_map>

// unordered_set
std::unordered_set<int> us = {3, 1, 4, 1, 5, 9};
// 元素唯一，但顺序不确定
for (int x : us) {
    std::cout << x << " "; // 顺序不确定
}
std::cout << std::endl;

us.insert(2);
us.erase(4);
bool found = (us.find(5) != us.end());

// unordered_map
std::unordered_map<std::string, int> scores;
scores["Alice"] = 95;
scores["Bob"] = 87;
scores["Charlie"] = 92;

// 接口与 map 类似，但元素无序
for (const auto& [name, score] : scores) {
    std::cout << name << ":" << score << std::endl; // 顺序不确定
}
```

无序容器通常比有序容器更快，但有一些注意事项：

哈希冲突会影响性能。在最坏情况下，操作可能退化为 $O(n)$ 。

元素类型必须支持哈希。对于自定义类型，需要提供哈希函数：

```
struct Point {
    int x, y;

    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
```

```

    }

};

// 自定义哈希函数
struct PointHash {
    size_t operator()(const Point& p) const {
        return std::hash<int>()(p.x) ^ (std::hash<int>()(p.y) << 1);
    }
};

std::unordered_set<Point, PointHash> points;
points.insert({1, 2});
points.insert({3, 4});

```

在需要快速查找且不关心元素顺序的场景下，优先选择无序容器。

1.5.17.8. stack：栈

`std::stack` 是容器适配器，提供后进先出（LIFO）的访问模式。它默认基于 `deque` 实现，也可以指定其他底层容器。

```

#include <stack>

std::stack<int> st;

// 压栈
st.push(10);
st.push(20);
st.push(30);

// 查看栈顶
std::cout << "Top: " << st.top() << std::endl; // 30

// 出栈
st.pop(); // 移除 30
std::cout << "After pop: " << st.top() << std::endl; // 20

// 检查是否为空
while (!st.empty()) {
    std::cout << st.top() << " ";
    st.pop();
}
// 输出: 20 10

```

`stack` 只能访问栈顶元素，不能遍历。它适用于需要“最近添加的最先处理”语义的场景，如括号匹配、表达式求值、深度优先搜索等。

```

// 括号匹配示例
bool isBalanced(const std::string& expr) {
    std::stack<char> st;

    for (char c : expr) {
        if (c == '(' || c == '[' || c == '{') {
            st.push(c);
        } else if (c == ')' || c == ']' || c == '}') {
            if (st.empty()) return false;

```

```

        char top = st.top();
        if ((c == ')') && top != '(') ||
           (c == ']') && top != '[') ||
           (c == '}') && top != '{')) {
            return false;
        }
        st.pop();
    }

    return st.empty();
}

```

1.5.17.9. queue：队列

`std::queue` 提供先进先出（FIFO）的访问模式，同样是容器适配器。

```

#include <queue>

std::queue<int> q;

// 入队
q.push(10);
q.push(20);
q.push(30);

// 查看队首和队尾
std::cout << "Front: " << q.front() << std::endl; // 10
std::cout << "Back: " << q.back() << std::endl; // 30

// 出队
q.pop(); // 移除 10
std::cout << "After pop: " << q.front() << std::endl; // 20

// 大小
std::cout << "Size: " << q.size() << std::endl; // 2

```

`queue` 适用于“先来先服务”的场景，如任务调度、广度优先搜索、消息队列等。

在 RoboMaster 开发中，队列可用于缓存传感器数据、管理待处理的命令等：

```

std::queue<SensorData> sensorQueue;

// 生产者线程
void sensorCallback(const SensorData& data) {
    sensorQueue.push(data);
}

// 消费者线程
void processLoop() {
    while (running) {
        if (!sensorQueue.empty()) {
            SensorData data = sensorQueue.front();
            sensorQueue.pop();
            process(data);
        }
    }
}

```

1.5.17.10. priority_queue：优先队列

`std::priority_queue` 是一种特殊的队列，每次取出的是优先级最高的元素（默认是最大值）。它基于堆实现。

```
#include <queue>

// 默认是最大堆
std::priority_queue<int> maxPq;
maxPq.push(30);
maxPq.push(10);
maxPq.push(20);

std::cout << maxPq.top() << std::endl; // 30
maxPq.pop();
std::cout << maxPq.top() << std::endl; // 20

// 最小堆需要指定比较函数
std::priority_queue<int, std::vector<int>, std::greater<int>> minPq;
minPq.push(30);
minPq.push(10);
minPq.push(20);

std::cout << minPq.top() << std::endl; // 10
```

优先队列在 RoboMaster 开发中可用于目标优先级排序：

```
struct Target {
    int id;
    double priority;

    // 定义比较（优先级高的排前面）
    bool operator<(const Target& other) const {
        return priority < other.priority; // 注意: priority_queue 是最大堆
    }
};

std::priority_queue<Target> targetQueue;

void addTarget(const Target& t) {
    targetQueue.push(t);
}

Target getHighestPriorityTarget() {
    Target t = targetQueue.top();
    targetQueue.pop();
    return t;
}
```

1.5.17.11. 如何选择容器

选择合适的容器是编写高效代码的关键。以下是一些指导原则：

如果需要随机访问元素，且主要在末尾添加删除，选择 `vector`。它是最通用的容器，在大多数情况下都是首选。`vector` 的内存连续，缓存友好，遍历性能极佳。

如果需要在两端频繁添加删除，选择 `deque`。它在头尾操作都是常数时间，适合实现双端队列。

如果需要在中间频繁插入删除，且不需要随机访问，选择 `list`。但要注意 `list` 的缓存性能较差，实际上在很多场景下 `vector`（即使需要移动元素）反而更快。

如果需要元素自动排序且唯一，选择 `set`。如果允许重复元素，选择 `multiset`。

如果需要键值对且按键排序，选择 `map`。如果允许重复键，选择 `multimap`。

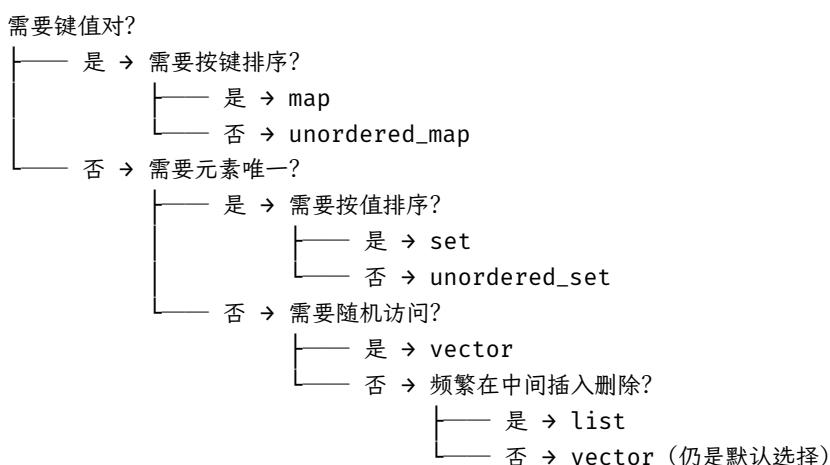
如果只需要快速查找、不关心顺序，选择 `unordered_set` 或 `unordered_map`。它们的平均性能优于有序版本。

如果需要后进先出语义，选择 `stack`。

如果需要先进先出语义，选择 `queue`。

如果需要按优先级处理元素，选择 `priority_queue`。

以下是一个简单的决策流程：



1.5.17.12. 容器的通用操作

STL 容器共享许多通用操作，这种一致性使得学习新容器变得简单：

```
// 以 vector 为例，但这些操作适用于大多数容器

std::vector<int> v = {1, 2, 3, 4, 5};

// 大小相关
v.size();      // 元素数量
v.empty();     // 是否为空
v.max_size();  // 理论最大容量

// 迭代器
v.begin();    // 指向第一个元素
v.end();       // 指向最后一个元素之后
v.rbegin();   // 反向迭代器，指向最后一个元素
v.rend();     // 反向迭代器，指向第一个元素之前

// 修改
v.clear();    // 清空所有元素
v.swap(other); // 与另一个容器交换内容

// 比较（字典序）
v < other;
v > other;
```

```
v1 == v2;      // 相等
v1 < v2;       // 小于
// 等等
```

这种统一的接口设计是 STL 的重要特征，它使得通用算法可以作用于任何容器。

1.5.17.13. 实践中的性能考虑

选择容器时，除了功能匹配，还要考虑性能特征。以下是各种操作的时间复杂度对比：

操作	vector	deque	list	set/map	unordered_set/map
随机访问	O(1)	O(1)	O(n)	O(log n)	O(1) 平均
头部插入	O(n)	O(1)	O(1)	-	-
尾部插入	O(1)*	O(1)	O(1)	-	-
中间插入	O(n)	O(n)	O(1)**	O(log n)	O(1) 平均
查找	O(n)	O(n)	O(n)	O(log n)	O(1) 平均

* 摊销常数时间

** 已有迭代器的情况下

然而，时间复杂度只是故事的一部分。`vector` 的缓存局部性使得它在很多场景下即使时间复杂度较高（如 O(n) 的中间插入），实际性能也优于理论上更快的 `list`。现代 CPU 的缓存机制使得连续内存访问比离散内存访问快得多。

在 RoboMaster 开发中的一般建议是：从 `vector` 开始，只有在性能分析表明它是瓶颈时才考虑其他容器。过早优化选择复杂的数据结构可能反而降低性能，同时增加代码复杂性。

STL 容器是 C++ 程序员最强大的工具之一。它们经过精心设计和优化，提供了丰富的功能和优异的性能。掌握它们，可以让你专注于解决实际问题，而不是重复实现基础数据结构。下一节我们将学习 STL 算法和迭代器，它们与容器配合使用，提供了强大的数据处理能力。

1.5.18. STL 算法与迭代器

上一节我们学习了 STL 容器，它们提供了多种数据存储方式。然而，仅有容器是不够的——我们还需要对容器中的数据进行各种操作：排序、查找、变换、统计等。如果每种容器都需要单独实现这些操作，代码量将会非常庞大。STL 的设计者采用了一种优雅的方案：通过迭代器 (iterator) 将容器与算法解耦。迭代器是一种抽象的“指针”，提供了统一的元素访问方式；算法则基于迭代器工作，不关心底层容器的具体类型。这种设计使得少量的算法可以作用于任意容器，极大地提高了代码的复用性。

1.5.18.1. 迭代器的概念

迭代器是容器与算法之间的桥梁。从概念上讲，迭代器是一种行为类似指针的对象，它指向容器中的某个元素，可以通过解引用访问该元素，通过递增移动到下一个元素。

每个 STL 容器都定义了自己的迭代器类型，可以通过 `begin()` 和 `end()` 成员函数获取：

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = {10, 20, 30, 40, 50};

    // 获取迭代器
    std::vector<int>::iterator it = v.begin(); // 指向第一个元素
```

```

    std::vector<int>::iterator end = v.end(); // 指向最后一个元素之后

    // 使用 auto 简化类型声明
    auto it2 = v.begin();

    // 解引用访问元素
    std::cout << *it << std::endl; // 10

    // 递增移动到下一个元素
    ++it;
    std::cout << *it << std::endl; // 20

    // 遍历容器
    for (auto iter = v.begin(); iter != v.end(); ++iter) {
        std::cout << *iter << " ";
    }
    std::cout << std::endl; // 10 20 30 40 50

    return 0;
}

```

`begin()` 返回指向第一个元素的迭代器，`end()` 返回指向“尾后”位置的迭代器——这是一个不存在的位置，表示容器的结束。这种半开区间 `[begin, end)` 的设计使得空容器可以用 `begin() == end()` 来判断，也使得遍历逻辑更加统一。

迭代器支持的操作取决于其类别。STL 定义了五种迭代器类别，从弱到强依次是：

输入迭代器（Input Iterator）只能向前移动，只能读取元素，且只能遍历一次。典型例子是从输入流读取数据的迭代器。

输出迭代器（Output Iterator）只能向前移动，只能写入元素。典型例子是向输出流写入数据的迭代器。

前向迭代器（Forward Iterator）可以多次遍历，支持读写。`forward_list` 的迭代器属于这一类。

双向迭代器（Bidirectional Iterator）可以向前和向后移动。`list`、`set`、`map` 的迭代器属于这一类。

随机访问迭代器（Random Access Iterator）支持所有指针运算，包括直接跳转到任意位置。`vector`、`deque`、`array` 的迭代器以及原生指针属于这一类。

```

std::vector<int> v = {10, 20, 30, 40, 50};
auto it = v.begin();

// 随机访问迭代器支持的操作
++it; // 前进一步
--it; // 后退一步
it += 3; // 前进三步
it -= 2; // 后退两步
auto it2 = it + 2; // 获取前方第二个位置的迭代器
int diff = it2 - it; // 计算两个迭代器之间的距离
bool less = it < it2; // 比较位置

std::cout << *it << std::endl; // 20
std::cout << it[2] << std::endl; // 40 (相当于 *(it + 2))

// list 的双向迭代器不支持随机访问

```

```

std::list<int> lst = {10, 20, 30};
auto lit = lst.begin();
++lit; // 正确
--lit; // 正确
// lit += 2; // 错误！双向迭代器不支持 +=
// lit[1]; // 错误！双向迭代器不支持 []

```

不同算法对迭代器有不同的要求。例如，`std::sort` 需要随机访问迭代器，因此不能用于 `list`；而 `std::find` 只需要输入迭代器，可以用于任何容器。

1.5.18.2. 迭代器的种类

除了普通迭代器，STL 还提供了几种特殊的迭代器。

常量迭代器（`const_iterator`）不允许通过迭代器修改元素。当容器是 `const` 的，或者只需要读取元素时，应使用常量迭代器：

```

std::vector<int> v = {10, 20, 30};

// 普通迭代器，可以修改元素
std::vector<int>::iterator it = v.begin();
*it = 100; // 正确

// 常量迭代器，不能修改元素
std::vector<int>::const_iterator cit = v.cbegin();
// *cit = 200; // 错误！不能通过 const_iterator 修改

// const 容器只能使用常量迭代器
const std::vector<int>& cv = v;
auto cit2 = cv.begin(); // 自动推导为 const_iterator

```

`cbegin()` 和 `cend()` 显式返回常量迭代器，即使容器本身不是 `const` 的。

反向迭代器（`reverse_iterator`）从后向前遍历容器：

```

std::vector<int> v = {10, 20, 30, 40, 50};

// 反向遍历
for (auto rit = v.rbegin(); rit != v.rend(); ++rit) {
    std::cout << *rit << " ";
}
std::cout << std::endl; // 50 40 30 20 10

// 常量反向迭代器
for (auto crit = v.crbegin(); crit != v.crend(); ++crit) {
    std::cout << *crit << " ";
}

```

插入迭代器（`insert iterator`）在写入时自动向容器插入新元素，而非覆盖现有元素：

```

#include <iostream>

std::vector<int> v = {1, 2, 3};
std::vector<int> result;

// back_inserter：在末尾插入
std::copy(v.begin(), v.end(), std::back_inserter(result));
// result = {1, 2, 3}

```

```

// front_inserter: 在开头插入 (需要支持 push_front 的容器)
std::list<int> lst;
std::copy(v.begin(), v.end(), std::front_inserter(lst));
// lst = {3, 2, 1} (注意顺序)

// inserter: 在指定位置插入
std::vector<int> v2 = {10, 20};
std::copy(v.begin(), v.end(), std::inserter(v2, v2.begin() + 1));
// v2 = {10, 1, 2, 3, 20}

```

1.5.18.3. 常用算法概览

STL 算法定义在 `<algorithm>` 头文件中 (部分数值算法在 `<numeric>` 中)。它们接受迭代器作为参数，对指定范围内的元素进行操作。

让我们从最常用的算法开始。

排序算法 `std::sort` 对元素进行升序排序：

```

#include <algorithm>
#include <vector>

std::vector<int> v = {30, 10, 50, 20, 40};

std::sort(v.begin(), v.end()); // 升序排序
// v = {10, 20, 30, 40, 50}

// 降序排序：使用比较函数
std::sort(v.begin(), v.end(), std::greater<int>());
// v = {50, 40, 30, 20, 10}

// 自定义比较函数
std::sort(v.begin(), v.end(), [](int a, int b) {
    return a > b; // 降序
});

// 对部分范围排序
std::vector<int> v2 = {5, 3, 1, 4, 2};
std::sort(v2.begin() + 1, v2.begin() + 4); // 只排序索引 1-3
// v2 = {5, 1, 3, 4, 2}

```

`std::sort` 使用快速排序的变体 (IntroSort)，平均时间复杂度 $O(n \log n)$ ，需要随机访问迭代器。对于 `list`，应使用其成员函数 `list::sort()`。

稳定排序 `std::stable_sort` 保持相等元素的相对顺序：

```

struct Student {
    std::string name;
    int score;
};

std::vector<Student> students = {
    {"Alice", 90}, {"Bob", 85}, {"Charlie", 90}, {"David", 85}
};

// 按分数排序，相同分数保持原顺序
std::stable_sort(students.begin(), students.end(),
    [](&const Student& a, &const Student& b) {

```

```

        return a.score > b.score;
    });
// Alice(90), Charlie(90), Bob(85), David(85)
// 相同分数的学生保持了原来的相对顺序

查找算法 std::find 在范围内查找特定值:

std::vector<int> v = {10, 20, 30, 40, 50};

auto it = std::find(v.begin(), v.end(), 30);
if (it != v.end()) {
    std::cout << "找到: " << *it << std::endl;
    std::cout << "索引: " << (it - v.begin()) << std::endl; // 2
} else {
    std::cout << "未找到" << std::endl;
}

// 查找满足条件的第一个元素
auto it2 = std::find_if(v.begin(), v.end(), [](int x) {
    return x > 25;
});
if (it2 != v.end()) {
    std::cout << "第一个大于25的数: " << *it2 << std::endl; // 30
}

// 查找不满足条件的第一个元素
auto it3 = std::find_if_not(v.begin(), v.end(), [](int x) {
    return x < 35;
});
// *it3 = 40

```

对于已排序的容器，`std::binary_search` 使用二分查找，效率更高：

```

std::vector<int> v = {10, 20, 30, 40, 50}; // 必须已排序

bool found = std::binary_search(v.begin(), v.end(), 30); // true
bool notFound = std::binary_search(v.begin(), v.end(), 35); // false

// 获取位置使用 lower_bound / upper_bound
auto lb = std::lower_bound(v.begin(), v.end(), 30); // 指向 30
auto ub = std::upper_bound(v.begin(), v.end(), 30); // 指向 40

```

计数算法 `std::count` 统计特定值的出现次数：

```

std::vector<int> v = {1, 2, 3, 2, 4, 2, 5};

int count = std::count(v.begin(), v.end(), 2);
std::cout << "2 出现了 " << count << " 次" << std::endl; // 3

// 统计满足条件的元素数量
int countEven = std::count_if(v.begin(), v.end(), [](int x) {
    return x % 2 == 0;
});
std::cout << "偶数有 " << countEven << " 个" << std::endl; // 4

```

1.5.18.4. 变换与修改算法

`std::transform` 对每个元素应用函数，将结果写入目标范围：

```

std::vector<int> v = {1, 2, 3, 4, 5};
std::vector<int> result(v.size());

// 每个元素乘以 2
std::transform(v.begin(), v.end(), result.begin(),
    [](int x) { return x * 2; });
// result = {2, 4, 6, 8, 10}

// 原地变换
std::transform(v.begin(), v.end(), v.begin(),
    [](int x) { return x * x; });
// v = {1, 4, 9, 16, 25}

// 两个范围的元素配对运算
std::vector<int> a = {1, 2, 3};
std::vector<int> b = {4, 5, 6};
std::vector<int> sum(3);

std::transform(a.begin(), a.end(), b.begin(), sum.begin(),
    [](int x, int y) { return x + y; });
// sum = {5, 7, 9}

std::for_each 对每个元素执行操作 (不产生新值):
std::vector<int> v = {1, 2, 3, 4, 5};

// 打印每个元素
std::for_each(v.begin(), v.end(), [](int x) {
    std::cout << x << " ";
});
std::cout << std::endl;

// 修改元素 (通过引用)
std::for_each(v.begin(), v.end(), [](int& x) {
    x *= 2;
});
// v = {2, 4, 6, 8, 10}

std::copy 复制元素到另一个范围:
std::vector<int> src = {1, 2, 3, 4, 5};
std::vector<int> dst(5);

std::copy(src.begin(), src.end(), dst.begin());
// dst = {1, 2, 3, 4, 5}

// 条件复制
std::vector<int> evens;
std::copy_if(src.begin(), src.end(), std::back_inserter(evens),
    [](int x) { return x % 2 == 0; });
// evens = {2, 4}

std::fill 将范围内所有元素设为指定值:
std::vector<int> v(10);
std::fill(v.begin(), v.end(), 42);
// v = {42, 42, 42, 42, 42, 42, 42, 42, 42, 42}

// 只填充部分

```

```

std::fill(v.begin(), v.begin() + 5, 0);
// v = {0, 0, 0, 0, 0, 42, 42, 42, 42, 42}

std::replace 替换特定值:
std::vector<int> v = {1, 2, 3, 2, 4, 2};

std::replace(v.begin(), v.end(), 2, 99);
// v = {1, 99, 3, 99, 4, 99}

// 条件替换
std::vector<int> v2 = {1, 2, 3, 4, 5};
std::replace_if(v2.begin(), v2.end(),
    [](int x) { return x % 2 == 0; }, 0);
// v2 = {1, 0, 3, 0, 5}

std::remove 和 std::remove_if “移除”满足条件的元素。注意，它们实际上不会改变容器大小，只是将不需要移除的元素移到前面，返回新的逻辑结尾:
std::vector<int> v = {1, 2, 3, 2, 4, 2, 5};

// remove 返回新的逻辑结尾
auto newEnd = std::remove(v.begin(), v.end(), 2);
// v 可能是 {1, 3, 4, 5, ?, ?, ?}, newEnd 指向第一个 ?

// 真正删除需要配合 erase
v.erase(newEnd, v.end());
// v = {1, 3, 4, 5}

// 常见的 erase-remove 惯用法
std::vector<int> v2 = {1, 2, 3, 4, 5, 6};
v2.erase(std::remove_if(v2.begin(), v2.end(),
    [](int x) { return x % 2 == 0; }), v2.end());
// v2 = {1, 3, 5}

std::unique 移除连续的重复元素:
std::vector<int> v = {1, 1, 2, 2, 2, 3, 3, 4};

auto newEnd = std::unique(v.begin(), v.end());
v.erase(newEnd, v.end());
// v = {1, 2, 3, 4}

// 对于非连续的重复，需要先排序
std::vector<int> v2 = {3, 1, 2, 1, 3, 2, 1};
std::sort(v2.begin(), v2.end());
v2.erase(std::unique(v2.begin(), v2.end()), v2.end());
// v2 = {1, 2, 3}

std::reverse 反转元素顺序:
std::vector<int> v = {1, 2, 3, 4, 5};
std::reverse(v.begin(), v.end());
// v = {5, 4, 3, 2, 1}

```

1.5.18.5. 数值算法

`<numeric>` 头文件提供了一些数值算法。

`std::accumulate` 计算范围内元素的累积值：

```

#include <numeric>

std::vector<int> v = {1, 2, 3, 4, 5};

// 求和
int sum = std::accumulate(v.begin(), v.end(), 0); // 15

// 求积
int product = std::accumulate(v.begin(), v.end(), 1,
    [] (int acc, int x) { return acc * x; }); // 120

// 字符串连接
std::vector<std::string> words = {"Hello", " ", "World"};
std::string sentence = std::accumulate(words.begin(), words.end(),
    std::string("")); // "Hello World"

std::inner_product 计算两个范围的内积:
std::vector<double> a = {1.0, 2.0, 3.0};
std::vector<double> b = {4.0, 5.0, 6.0};

double dot = std::inner_product(a.begin(), a.end(), b.begin(), 0.0);
// 1*4 + 2*5 + 3*6 = 32

std::partial_sum 计算前缀和:
std::vector<int> v = {1, 2, 3, 4, 5};
std::vector<int> prefix(v.size());

std::partial_sum(v.begin(), v.end(), prefix.begin());
// prefix = {1, 3, 6, 10, 15}

std::iota 填充递增序列:
std::vector<int> v(10);
std::iota(v.begin(), v.end(), 1);
// v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

```

1.5.18.6. 范围 for 循环

C++11 引入的范围 for 循环 (range-based for loop) 大大简化了容器遍历的语法。它内部使用迭代器，但隐藏了迭代器的细节：

```

std::vector<int> v = {1, 2, 3, 4, 5};

// 传统迭代器方式
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
}

// 范围 for 循环
for (int x : v) {
    std::cout << x << " ";
}

```

范围 for 循环有几种形式，适用于不同场景：

```

std::vector<std::string> names = {"Alice", "Bob", "Charlie"};

// 值拷贝：适用于小型类型，不修改原容器
for (std::string name : names) {

```

```

        std::cout << name << std::endl;
    }

// 常量引用：避免拷贝，只读访问（推荐用于大型对象）
for (const std::string& name : names) {
    std::cout << name << std::endl;
}

// 引用：需要修改元素时使用
for (std::string& name : names) {
    name = "Mr. " + name;
}
// names = {"Mr. Alice", "Mr. Bob", "Mr. Charlie"}

// auto 自动推导类型
for (const auto& name : names) {
    std::cout << name << std::endl;
}

```

范围 for 循环可以用于任何提供 begin() 和 end() 的对象，包括：

```

// STL 容器
std::vector<int> vec = {1, 2, 3};
for (int x : vec) { /* ... */ }

// 原生数组
int arr[] = {1, 2, 3, 4, 5};
for (int x : arr) { /* ... */ }

// 初始化列表
for (int x : {1, 2, 3, 4, 5}) { /* ... */ }

// std::string
std::string str = "Hello";
for (char c : str) { /* ... */ }

// map 等关联容器
std::map<std::string, int> ages = {"Alice", 25}, {"Bob", 30};
for (const auto& pair : ages) {
    std::cout << pair.first << ":" << pair.second << std::endl;
}

// C++17 结构化绑定
for (const auto& [name, age] : ages) {
    std::cout << name << ":" << age << std::endl;
}

```

范围 for 循环的一个限制是无法直接获取当前索引或迭代器。如果需要索引，可以使用传统循环或额外维护一个计数器：

```

std::vector<int> v = {10, 20, 30, 40, 50};

// 需要索引时的选择
// 方式一：传统循环
for (size_t i = 0; i < v.size(); i++) {
    std::cout << i << ":" << v[i] << std::endl;
}

```

```
// 方式二：手动维护索引
size_t index = 0;
for (const auto& x : v) {
    std::cout << index << ":" << x << std::endl;
    index++;
}
```

1.5.18.7. 算法与谓词

许多 STL 算法接受谓词 (predicate) 作为参数。谓词是返回布尔值的函数或函数对象，用于自定义算法的行为。

谓词可以是普通函数：

```
bool isPositive(int x) {
    return x > 0;
}

std::vector<int> v = {-2, -1, 0, 1, 2};
int count = std::count_if(v.begin(), v.end(), isPositive); // 2
```

也可以是函数对象：

```
struct GreaterThan {
    int threshold;
    GreaterThan(int t) : threshold(t) {}
    bool operator()(int x) const {
        return x > threshold;
    }
};

std::vector<int> v = {1, 5, 3, 8, 2, 9, 4};
auto it = std::find_if(v.begin(), v.end(), GreaterThan(6));
// *it = 8
```

最常用的是 lambda 表达式，它提供了简洁的内联定义方式：

```
std::vector<int> v = {1, 5, 3, 8, 2, 9, 4};

// 基本 lambda
auto it = std::find_if(v.begin(), v.end(), [] (int x) {
    return x > 6;
});

// 捕获外部变量
int threshold = 6;
auto it2 = std::find_if(v.begin(), v.end(), [threshold] (int x) {
    return x > threshold;
});

// 按引用捕获
int sum = 0;
std::for_each(v.begin(), v.end(), [&sum] (int x) {
    sum += x;
});

// 捕获所有局部变量（按值）
// [=] (int x) { ... }
```

```
// 捕获所有局部变量（按引用）
// [&](int x) { ... }
```

STL 还提供了一些预定义的函数对象，定义在 `<functional>` 头文件中：

```
#include <functional>

std::vector<int> v = {30, 10, 50, 20, 40};

// 降序排序
std::sort(v.begin(), v.end(), std::greater<int>());
// v = {50, 40, 30, 20, 10}

// 其他常用函数对象
// std::less<T>      小于比较（默认）
// std::greater<T>    大于比较
// std::plus<T>       加法
// std::minus<T>      减法
// std::multiplies<T>  乘法
// std::divides<T>    除法
// std::negate<T>     取负
// std::equal_to<T>   相等比较
// std::not_equal_to<T> 不等比较
```

1.5.18.8. 在 RoboMaster 开发中的应用

让我们看几个 STL 算法在 RoboMaster 开发中的实际应用。

目标筛选与排序：

```
struct Target {
    int id;
    double distance;
    double confidence;
    bool isEnemy;
};

std::vector<Target> targets;

// 筛选敌方目标
std::vector<Target> enemies;
std::copy_if(targets.begin(), targets.end(), std::back_inserter(enemies),
[](const Target& t) { return t.isEnemy; });

// 按距离排序
std::sort(enemies.begin(), enemies.end(),
[](const Target& a, const Target& b) {
    return a.distance < b.distance;
});

// 找最高置信度的目标
auto bestTarget = std::max_element(enemies.begin(), enemies.end(),
[](const Target& a, const Target& b) {
    return a.confidence < b.confidence;
});

if (bestTarget != enemies.end()) {
```

```
        std::cout << "最佳目标 ID: " << bestTarget->id << std::endl;
    }
```

// 移除低置信度目标

```
enemies.erase(
    std::remove_if(enemies.begin(), enemies.end(),
        [](const Target& t) { return t.confidence < 0.5; }), 
    enemies.end());
```

传感器数据处理:

```
struct SensorReading {
    double timestamp;
    double value;
};

std::vector<SensorReading> readings;

// 计算平均值
double sum = std::accumulate(readings.begin(), readings.end(), 0.0,
    [](double acc, const SensorReading& r) {
        return acc + r.value;
});
double average = readings.empty() ? 0.0 : sum / readings.size();

// 找最大最小值
auto [minIt, maxIt] = std::minmax_element(readings.begin(), readings.end(),
    [](const SensorReading& a, const SensorReading& b) {
        return a.value < b.value;
});

// 检查是否所有读数都在有效范围内
bool allValid = std::all_of(readings.begin(), readings.end(),
    [](const SensorReading& r) {
        return r.value >= 0.0 && r.value <= 100.0;
});

// 检查是否有任何异常值
bool hasAnomaly = std::any_of(readings.begin(), readings.end(),
    [average](const SensorReading& r) {
        return std::abs(r.value - average) > 3.0 * stddev; // 假设 stddev 已
计算
});
```

电机控制数组操作:

```
std::array<double, 4> motorSpeeds = {1000, 2000, 1500, 1800};
std::array<double, 4> targetSpeeds = {1200, 2200, 1600, 2000};

// 计算误差
std::array<double, 4> errors;
std::transform(targetSpeeds.begin(), targetSpeeds.end(),
    motorSpeeds.begin(), errors.begin(),
    [] (double target, double current) {
        return target - current;
});

// 限幅处理
```

```

const double maxSpeed = 5000.0;
std::transform(motorSpeeds.begin(), motorSpeeds.end(), motorSpeeds.begin(),
[&maxSpeed](double speed) {
    return std::clamp(speed, -maxSpeed, maxSpeed);
});

// 检查是否所有电机都接近目标
bool allOnTarget = std::all_of(errors.begin(), errors.end(),
[](double e) { return std::abs(e) < 10.0; });

```

1.5.18.9. 算法的效率考虑

使用 STL 算法时，了解其复杂度很重要：

算法	时间复杂度	说明
sort	$O(n \log n)$	快速排序变体
stable_sort	$O(n \log n)$	归并排序
find	$O(n)$	线性查找
binary_search	$O(\log n)$	需要已排序
count	$O(n)$	线性扫描
transform	$O(n)$	线性变换
copy	$O(n)$	线性复制
unique	$O(n)$	线性扫描
reverse	$O(n)$	线性反转
min/max_element	$O(n)$	线性扫描
accumulate	$O(n)$	线性累加

一些优化建议：

对于需要频繁查找的数据，考虑使用 `set`、`map` 或无序容器，而非在 `vector` 上使用 `find`。

如果需要多次查找，先排序再使用 `binary_search` 或 `lower_bound`。

避免在循环中重复调用 `size()` 或 `end()`，虽然现代编译器通常会优化这些调用。

使用 `reserve()` 预分配空间，避免多次重新分配。

对于大型对象，使用移动语义（后续章节介绍）减少拷贝开销。

1.5.18.10. 自定义类型与算法

要让自定义类型与 STL 算法配合使用，需要满足某些要求。

对于排序和查找算法，类型需要支持比较操作：

```

struct Point {
    double x, y;

    // 方式一：重载 < 运算符
    bool operator<(const Point& other) const {
        if (x != other.x) return x < other.x;
        return y < other.y;
    }

    // 方式二：重载 == 运算符（用于 find 等）
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }
};

```

```

std::vector<Point> points = {{1, 2}, {3, 1}, {2, 3}};
std::sort(points.begin(), points.end()); // 使用 operator<

Point target = {3, 1};
auto it = std::find(points.begin(), points.end(), target); // 使用 operator==

```

也可以不修改类型，而是提供自定义比较器：

```

struct Point {
    double x, y;
};

// 按到原点距离排序
std::sort(points.begin(), points.end(),
    [] (const Point& a, const Point& b) {
        double distA = a.x * a.x + a.y * a.y;
        double distB = b.x * b.x + b.y * b.y;
        return distA < distB;
});

```

STL 算法与迭代器的组合提供了强大而灵活的数据处理能力。通过学习这些工具，你可以用更少的代码完成更多的工作，同时保持代码的清晰和效率。算法表达了“做什么”，而不是“怎么做”，这使得代码更易于理解和维护。下一节我们将学习模板基础，了解 STL 容器和算法背后的泛型编程技术。

1.5.19. 模板基础

在前面的章节中，我们大量使用了 STL 容器和算法：`vector<int>`、`map<string, double>`、`sort`、`find` 等。你可能已经注意到，这些组件可以处理任意类型的数据——`vector` 可以存储 `int`、`double`、`string` 甚至自定义类型，`sort` 可以对任何支持比较操作的类型进行排序。这种“一次编写，处理多种类型”的能力正是 C++ 模板（template）提供的。模板是 C++ 泛型编程的基础，它允许我们编写与类型无关的代码，由编译器在使用时根据具体类型生成相应的实现。理解模板不仅能帮助你更好地使用 STL，还能让你编写出更加通用、可复用的代码。

1.5.19.1. 为什么需要模板

假设我们需要编写一个函数来交换两个整数的值：

```

void swapInt(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

```

现在又需要交换两个 `double` 类型的值。由于 C++ 是强类型语言，我们必须编写另一个函数：

```

void swapDouble(double& a, double& b) {
    double temp = a;
    a = b;
    b = temp;
}

```

如果还需要交换 `string`、`Vector3D`、`Motor` 等类型呢？每种类型都写一个几乎相同的函数，这显然不是好的做法——代码重复、维护困难、容易出错。

函数重载可以让这些函数同名：

```

void swap(int& a, int& b) { /* ... */ }
void swap(double& a, double& b) { /* ... */ }
void swap(std::string& a, std::string& b) { /* ... */ }

```

但这仍然没有解决代码重复的问题。而且，对于用户自定义的类型，我们无法预先提供重载版本。

模板提供了根本性的解决方案：编写一个通用的“模板”，让编译器根据实际使用的类型自动生成具体的函数。

1.5.19.2. 函数模板

函数模板使用 `template` 关键字定义，后跟模板参数列表。模板参数通常是类型参数，用 `typename` 或 `class` 关键字声明：

```

template <typename T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

```

这里 `T` 是类型参数，代表任意类型。当我们调用这个函数时，编译器会根据实参类型推导出 `T` 的具体类型，并生成相应的函数实例：

```

int main() {
    int x = 10, y = 20;
    swap(x, y); // T 被推导为 int, 生成 swap<int>

    double a = 1.5, b = 2.5;
    swap(a, b); // T 被推导为 double, 生成 swap<double>

    std::string s1 = "hello", s2 = "world";
    swap(s1, s2); // T 被推导为 std::string

    return 0;
}

```

编译器为每种使用到的类型生成一个独立的函数版本，这个过程称为模板实例化（instantiation）。生成的代码与手写的类型特定版本完全相同，没有任何运行时开销。

也可以显式指定模板参数：

```

swap<int>(x, y); // 显式指定 T 为 int
swap<double>(a, b); // 显式指定 T 为 double

```

当编译器无法自动推导类型，或者需要强制使用特定类型时，显式指定是必要的。

函数模板可以有多个类型参数：

```

template <typename T, typename U>
void printPair(const T& first, const U& second) {
    std::cout << "(" << first << ", " << second << ")" << std::endl;
}

int main() {
    printPair(10, 3.14); // T=int, U=double
    printPair("hello", 42); // T=const char*, U=int
    printPair(std::string("hi"), std::vector<int>{1,2,3}); // 复杂类型也可以
}

```

```
        return 0;
    }
```

1.5.19.3. 类模板

类模板允许定义通用的类，其成员的类型可以参数化。STL 容器就是类模板的典型例子。

定义一个简单的类模板：

```
template <typename T>
class Container {
public:
    Container(const T& value) : data(value) {}

    T getData() const { return data; }
    void setData(const T& value) { data = value; }

private:
    T data;
};
```

使用类模板时，必须显式指定类型参数（C++17 之前）：

```
int main() {
    Container<int> intContainer(42);
    std::cout << intContainer.getData() << std::endl; // 42

    Container<std::string> strContainer("hello");
    std::cout << strContainer.getData() << std::endl; // hello

    Container<double> dblContainer(3.14);
    dblContainer.setData(2.71);

    return 0;
}
```

C++17 引入了类模板参数推导（CTAD），允许编译器从构造函数参数推导类型：

```
// C++17 及以后
Container intContainer(42);           // 推导为 Container<int>
Container strContainer("hello");       // 推导为 Container<const char*>
Container dblContainer(3.14);          // 推导为 Container<double>
```

让我们实现一个更实用的类模板——简化版的动态数组：

```
template <typename T>
class DynamicArray {
public:
    DynamicArray() : data(nullptr), size(0), capacity(0) {}

    ~DynamicArray() {
        delete[] data;
    }

    void push_back(const T& value) {
        if (size >= capacity) {
            grow();
        }
        data[size++] = value;
    }
```

```

T& operator[](size_t index) {
    return data[index];
}

const T& operator[](size_t index) const {
    return data[index];
}

size_t getSize() const { return size; }
bool empty() const { return size == 0; }

private:
    void grow() {
        size_t newCapacity = (capacity == 0) ? 1 : capacity * 2;
        T* newData = new T[newCapacity];
        for (size_t i = 0; i < size; i++) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
        capacity = newCapacity;
    }

    T* data;
    size_t size;
    size_t capacity;
};

int main() {
    DynamicArray<int> numbers;
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    for (size_t i = 0; i < numbers.getSize(); i++) {
        std::cout << numbers[i] << " ";
    }
    std::cout << std::endl; // 10 20 30

    DynamicArray<std::string> words;
    words.push_back("hello");
    words.push_back("world");

    return 0;
}

```

这个简化版展示了类模板的基本结构。真正的 `std::vector` 实现要复杂得多，包括异常安全、移动语义、分配器支持等。

1.5.19.4. 成员函数模板

类（无论是否是模板类）可以拥有成员函数模板：

```

class Printer {
public:

```

```

template <typename T>
void print(const T& value) {
    std::cout << value << std::endl;
}

template <typename T, typename U>
void printPair(const T& first, const U& second) {
    std::cout << first << ", " << second << std::endl;
}

int main() {
    Printer p;
    p.print(42);           // 实例化 print<int>
    p.print(3.14);         // 实例化 print<double>
    p.print("hello");      // 实例化 print<const char*>
    p.printPair(1, "one"); // 实例化 printPair<int, const char*>
    return 0;
}

```

类模板也可以有成员函数模板，这在实现通用接口时很有用：

```

template <typename T>
class Container {
public:
    Container(const T& value) : data(value) {}

    // 成员函数模板：允许从不同类型的 Container 构造
    template <typename U>
    Container(const Container<U>& other) : data(other.getData()) {}

    T getData() const { return data; }

private:
    T data;
};

int main() {
    Container<int> intCont(42);
    Container<double> dblCont(intCont); // 从 Container<int> 构造
    std::cout << dblCont.getData() << std::endl; // 42.0
    return 0;
}

```

1.5.19.5. 非类型模板参数

除了类型参数，模板还可以有非类型参数（non-type template parameter）。非类型参数必须是编译时常量，可以是整数、枚举、指针或引用：

```

template <typename T, size_t N>
class FixedArray {
public:
    T& operator[](size_t index) {
        return data[index];
    }

```

```

        const T& operator[](size_t index) const {
            return data[index];
        }

        constexpr size_t size() const { return N; }

private:
    T data[N];
};

int main() {
    FixedArray<int, 5> arr1;      // 5 个 int 的数组
    FixedArray<double, 10> arr2; // 10 个 double 的数组

    for (size_t i = 0; i < arr1.size(); i++) {
        arr1[i] = i * 10;
    }

    // arr1 和 arr2 是不同的类型
    // FixedArray<int, 5> 与 FixedArray<int, 6> 也是不同的类型

    return 0;
}

```

`std::array` 就是使用非类型模板参数的典型例子：`std::array<int, 10>` 表示包含 10 个整数的固定大小数组。

非类型参数还可以用于编译时计算：

```

template <int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
    static constexpr int value = 1;
};

int main() {
    constexpr int fact5 = Factorial<5>::value; // 编译时计算 5! = 120
    std::cout << "5! = " << fact5 << std::endl;
    return 0;
}

```

这种技术称为模板元编程 (template metaprogramming)，它在编译时进行计算，可以实现许多高级功能。

1.5.19.6. 默认模板参数

模板参数可以有默认值：

```

template <typename T = int, size_t N = 10>
class Buffer {
public:
    T& operator[](size_t index) { return data[index]; }
    size_t size() const { return N; }
private:

```

```

    T data[N];
};

int main() {
    Buffer<> buf1;           // T=int, N=10
    Buffer<double> buf2;     // T=double, N=10
    Buffer<char, 256> buf3;  // T=char, N=256
    return 0;
}

```

STL 容器广泛使用默认模板参数。例如，`std::vector` 的完整声明是：

```

template <typename T, typename Allocator = std::allocator<T>>
class vector;

```

大多数情况下我们只需要指定 `T`，分配器使用默认值。

1.5.19.7. 模板特化

有时候，通用的模板实现对某些特定类型不适用或效率不高。模板特化（template specialization）允许为特定类型提供定制实现。

全特化（full specialization）为所有模板参数指定具体类型：

```

// 通用版本
template <typename T>
class TypeInfo {
public:
    static std::string name() { return "unknown"; }
};

// 针对 int 的特化
template <>
class TypeInfo<int> {
public:
    static std::string name() { return "int"; }
};

// 针对 double 的特化
template <>
class TypeInfo<double> {
public:
    static std::string name() { return "double"; }
};

// 针对 std::string 的特化
template <>
class TypeInfo<std::string> {
public:
    static std::string name() { return "std::string"; }
};

int main() {
    std::cout << TypeInfo<int>::name() << std::endl;           // int
    std::cout << TypeInfo<double>::name() << std::endl;         // double
    std::cout << TypeInfo<std::string>::name() << std::endl; // std::string
    std::cout << TypeInfo<char>::name() << std::endl;          // unknown
}

```

```
        return 0;
    }
```

偏特化 (partial specialization) 只特化部分参数，或对参数施加约束：

```
// 通用版本
template <typename T>
class Container {
public:
    void info() { std::cout << "Generic container" << std::endl; }

};

// 针对指针类型的偏特化
template <typename T*>
class Container<T*> {
public:
    void info() { std::cout << "Pointer container" << std::endl; }

};

// 针对数组类型的偏特化
template <typename T, size_t N>
class Container<T[N]> {
public:
    void info() { std::cout << "Array container of size " << N << std::endl; }

};

int main() {
    Container<int> c1;
    Container<int*> c2;
    Container<double[10]> c3;

    c1.info(); // Generic container
    c2.info(); // Pointer container
    c3.info(); // Array container of size 10

    return 0;
}
```

注意：函数模板不支持偏特化，但可以通过重载达到类似效果。

1.5.19.8. 模板与编译

理解模板的编译模型对于避免常见错误很重要。

模板本身不是代码，而是生成代码的蓝图。只有当模板被实例化时，编译器才会生成实际的代码。这意味着：

模板定义通常放在头文件中。由于编译器需要看到完整的模板定义才能实例化，模板的声明和定义通常都放在头文件中，而不像普通函数那样声明在头文件、定义在源文件。

```
// MyTemplate.h
#ifndef MY_TEMPLATE_H
#define MY_TEMPLATE_H

template <typename T>
class MyTemplate {
public:
    void foo();
```

```

    void bar();
};

// 成员函数定义也在头文件中
template <typename T>
void MyTemplate<T>::foo() {
    // 实现
}

template <typename T>
void MyTemplate<T>::bar() {
    // 实现
}

#endif

```

未使用的模板代码不会被编译。如果模板中有语法错误，但该模板从未被实例化，错误可能不会被发现。

类型检查发生在实例化时。模板代码对类型的要求（如需要 `operator<`）只有在实例化时才会被检查：

```

template <typename T>
T maximum(const T& a, const T& b) {
    return (a > b) ? a : b; // 要求 T 支持 operator>
}

struct NoCompare {
    int value;
};

int main() {
    maximum(10, 20); // 正确, int 支持 >

    NoCompare nc1{1}, nc2{2};
    // maximum(nc1, nc2); // 错误! NoCompare 不支持 >

    return 0;
}

```

1.5.19.9. 实际应用示例

让我们看几个模板在 RoboMaster 开发中的实际应用。

通用的数值范围限制函数：

```

template <typename T>
T clamp(const T& value, const T& minVal, const T& maxVal) {
    if (value < minVal) return minVal;
    if (value > maxVal) return maxVal;
    return value;
}

// 使用
double speed = clamp(rawSpeed, -1000.0, 1000.0);
int pwm = clamp(calculatedPwm, 0, 255);

```

通用的 PID 控制器：

```

template <typename T>
class PIDController {
public:
    PIDController(T kp, T ki, T kd)
        : kp(kp), ki(ki), kd(kd), integral(0), lastError(0) {}

    T compute(T setpoint, T measurement, T dt) {
        T error = setpoint - measurement;
        integral += error * dt;
        T derivative = (error - lastError) / dt;
        lastError = error;
        return kp * error + ki * integral + kd * derivative;
    }

    void reset() {
        integral = 0;
        lastError = 0;
    }

private:
    T kp, ki, kd;
    T integral;
    T lastError;
};

// 使用 double 精度
PIDController<double> speedController(1.5, 0.1, 0.05);

// 使用 float 精度（嵌入式系统中常见）
PIDController<float> angleController(2.0f, 0.0f, 0.1f);

```

通用的环形缓冲区：

```

template <typename T, size_t N>
class RingBuffer {
public:
    RingBuffer() : head(0), tail(0), count(0) {}

    bool push(const T& value) {
        if (count >= N) return false; // 缓冲区满
        data[tail] = value;
        tail = (tail + 1) % N;
        count++;
        return true;
    }

    bool pop(T& value) {
        if (count == 0) return false; // 缓冲区空
        value = data[head];
        head = (head + 1) % N;
        count--;
        return true;
    }

    bool empty() const { return count == 0; }
    bool full() const { return count >= N; }
    size_t size() const { return count; }

```

```

private:
    T data[N];
    size_t head;
    size_t tail;
    size_t count;
};

// 用于存储最近 100 个传感器读数
RingBuffer<SensorReading, 100> sensorHistory;

// 用于通信数据缓冲
RingBuffer<uint8_t, 256> rxBuffer;

```

通用的单例模式：

```

template <typename T>
class Singleton {
public:
    static T& getInstance() {
        static T instance;
        return instance;
    }

    // 禁止拷贝和赋值
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

protected:
    Singleton() = default;
    ~Singleton() = default;
};

// 使用单例
class ConfigManager : public Singleton<ConfigManager> {
    friend class Singleton<ConfigManager>;
public:
    void loadConfig(const std::string& filename);
    int getValue(const std::string& key);
private:
    ConfigManager() = default;
    std::map<std::string, int> config;
};

// 访问单例
ConfigManager::getInstance().loadConfig("robot.cfg");

```

1.5.19.10. 模板的优缺点

模板是强大的工具，但也有其局限性。

优点包括：

代码复用：一次编写，适用于多种类型。

类型安全：编译时类型检查，不像 `void*` 那样丢失类型信息。

零运行时开销：模板实例化发生在编译时，生成的代码与手写的特定类型代码效率相同。

强大的抽象能力：可以表达复杂的类型关系和约束。

缺点包括：

编译时间增加：每个翻译单元都可能实例化相同的模板，增加编译时间。

代码膨胀：每种类型实例化都会生成独立的代码，可能增加可执行文件大小。

错误信息复杂：模板相关的编译错误信息通常很长且难以理解。

调试困难：模板代码的调试比普通代码更具挑战性。

1.5.19.11. 现代 C++ 中的模板改进

C++11 及以后的标准对模板进行了许多改进。

变参模板（variadic templates）允许模板接受任意数量的参数：

```
// 递归终止
void print() {
    std::cout << std::endl;
}

// 变参模板
template <typename T, typename... Args>
void print(const T& first, const Args&... rest) {
    std::cout << first << " ";
    print(rest...); // 递归展开
}

int main() {
    print(1, 2.5, "hello", 'c'); // 1 2.5 hello c
    return 0;
}
```

auto 和 decltype 简化了复杂类型的声明：

```
template <typename T, typename U>
auto add(const T& a, const U& b) -> decltype(a + b) {
    return a + b;
}

// C++14 可以省略尾置返回类型
template <typename T, typename U>
auto multiply(const T& a, const U& b) {
    return a * b;
}
```

C++20 引入了概念（concepts），可以对模板参数施加约束，提供更好的错误信息：

```
// C++20 概念
template <typename T>
concept Comparable = requires(T a, T b) {
    { a < b } -> std::convertible_to<bool>;
    { a > b } -> std::convertible_to<bool>;
};

template <Comparable T>
T maximum(const T& a, const T& b) {
    return (a > b) ? a : b;
}
```

模板是 C++ 最强大的特性之一，也是 STL 的基础。通过本节的学习，你应该能够理解 STL 容器和算法的工作原理，并能够编写自己的模板代码。模板的高级用法（如 SFINAE、模板元编程、类型萃取等）超出了本教程的范围，但掌握基础知识已经足以应对大多数实际需求。下一节我们将学习智能指针，它们是现代 C++ 内存管理的核心工具。

1.5.20. 智能指针

在前面的章节中，我们多次提到动态内存管理的挑战：使用 `new` 分配的内存必须用 `delete` 释放，否则会造成内存泄漏；如果释放得太早或重复释放，程序会崩溃或产生未定义行为。手动管理内存不仅繁琐，还极易出错，特别是在存在多个执行路径（如异常、提前返回）的复杂代码中。

C++11 引入的智能指针（smart pointer）从根本上解决了这个问题。智能指针是一种封装了原始指针的类模板，它在适当的时机自动释放所管理的内存，遵循 RAII 原则。使用智能指针，你几乎不再需要直接调用 `new` 和 `delete`，内存管理变得安全而自动。现代 C++ 代码中，智能指针已经成为管理动态分配对象的标准方式。

1.5.20.1. 原始指针的问题

让我们先回顾一下原始指针管理动态内存时面临的问题：

```
void processData() {
    int* data = new int[1000];

    // 处理数据...

    if (someError) {
        return; // 忘记 delete! 内存泄漏
    }

    // 更多处理...

    if (anotherError) {
        throw std::runtime_error("处理失败"); // 异常导致 delete 不执行
    }

    delete[] data; // 只有正常执行到这里才会释放
}
```

这段代码有多个问题：提前返回和异常都会导致内存泄漏。要正确处理，需要在每个退出点都添加 `delete`：

```
void processData() {
    int* data = new int[1000];

    try {
        if (someError) {
            delete[] data;
            return;
        }

        // ...

        if (anotherError) {
            delete[] data;
        }
    }
}
```

```

        throw std::runtime_error("处理失败");
    }

    delete[] data;
} catch (...) {
    delete[] data;
    throw;
}
}

```

代码变得冗长且容易出错。智能指针通过 RAII 优雅地解决了这个问题：当智能指针离开作用域时，析构函数自动释放资源，无论是正常退出、提前返回还是异常抛出。

1.5.20.2. unique_ptr：独占所有权

`std::unique_ptr` 表示对动态分配对象的独占所有权。同一时刻只有一个 `unique_ptr` 可以指向特定对象，当 `unique_ptr` 被销毁时，它所管理的对象也随之被删除。

基本用法：

```

#include <memory>
#include <iostream>

class Motor {
public:
    Motor(int id) : id(id) {
        std::cout << "Motor " << id << " 创建" << std::endl;
    }
    ~Motor() {
        std::cout << "Motor " << id << " 销毁" << std::endl;
    }
    void run() {
        std::cout << "Motor " << id << " 运行中" << std::endl;
    }
private:
    int id;
};

int main() {
    // 创建 unique_ptr
    std::unique_ptr<Motor> motor1(new Motor(1));

    // 推荐：使用 make_unique (C++14)
    auto motor2 = std::make_unique<Motor>(2);

    // 使用智能指针（像普通指针一样）
    motor1->run();
    motor2->run();

    // 检查是否为空
    if (motor1) {
        std::cout << "motor1 有效" << std::endl;
    }

    return 0;
} // motor1 和 motor2 离开作用域，自动销毁所管理的对象

```

输出：

```
Motor 1 创建
Motor 2 创建
Motor 1 运行中
Motor 2 运行中
motor1 有效
Motor 2 销毁
Motor 1 销毁
```

`std::make_unique` 是创建 `unique_ptr` 的推荐方式，它更安全（异常安全）、更简洁，且避免了显式使用 `new`。

`unique_ptr` 的核心特性是不可复制，这确保了独占所有权的语义：

```
auto ptr1 = std::make_unique<Motor>(1);

// std::unique_ptr<Motor> ptr2 = ptr1; // 错误！不能复制
// auto ptr3 = ptr1; // 错误！不能复制

// 但可以移动
std::unique_ptr<Motor> ptr2 = std::move(ptr1); // 所有权转移
// 现在 ptr1 为空，ptr2 拥有对象

if (!ptr1) {
    std::cout << "ptr1 现在为空" << std::endl;
}
```

移动后，原来的 `unique_ptr` 变为空 (`nullptr`)，所有权完全转移到新的 `unique_ptr`。

`unique_ptr` 常用的操作：

```
auto ptr = std::make_unique<Motor>(1);

// 获取原始指针（不转移所有权）
Motor* raw = ptr.get();

// 释放所有权，返回原始指针（调用者负责删除）
Motor* released = ptr.release();
// ptr 现在为空，调用者必须手动 delete released

// 重置：删除当前对象，可选地管理新对象
ptr.reset(); // 删除并置空
ptr.reset(new Motor(2)); // 删除旧对象，管理新对象

// 交换
auto ptr2 = std::make_unique<Motor>(3);
ptr.swap(ptr2);
```

`unique_ptr` 可以管理数组：

```
// 管理动态数组
auto arr = std::make_unique<int[]>(100);
arr[0] = 10;
arr[99] = 20;
// 自动调用 delete[]
```

在函数参数和返回值中使用 `unique_ptr`：

```
// 工厂函数：返回 unique_ptr 表示调用者获得所有权
std::unique_ptr<Motor> createMotor(int id) {
```

```

        return std::make_unique<Motor>(id);
    }

// 接收 unique_ptr: 转移所有权 (接收者负责生命周期)
void takeOwnership(std::unique_ptr<Motor> motor) {
    motor->run();
} // motor 在这里被销毁

// 只使用不获取所有权: 传递引用或原始指针
void useMotor(Motor& motor) {
    motor.run();
}

void useMotorPtr(Motor* motor) {
    if (motor) motor->run();
}

int main() {
    auto motor = createMotor(1);

    useMotor(*motor);           // 传递引用
    useMotorPtr(motor.get());   // 传递原始指针

    takeOwnership(std::move(motor)); // 转移所有权
    // motor 现在为空

    return 0;
}

```

1.5.20.3. shared_ptr: 共享所有权

`std::shared_ptr` 允许多个指针共享同一个对象的所有权。它使用引用计数来跟踪有多少个 `shared_ptr` 指向同一对象，当最后一个 `shared_ptr` 被销毁时，对象才被删除。

```

#include <memory>
#include <iostream>

class Sensor {
public:
    Sensor(const std::string& name) : name(name) {
        std::cout << "Sensor " << name << " 创建" << std::endl;
    }
    ~Sensor() {
        std::cout << "Sensor " << name << " 销毁" << std::endl;
    }
    std::string getName() const { return name; }
private:
    std::string name;
};

int main() {
    std::shared_ptr<Sensor> sensor1 = std::make_shared<Sensor>("IMU");
    std::cout << "引用计数: " << sensor1.use_count() << std::endl; // 1

    {
        std::shared_ptr<Sensor> sensor2 = sensor1; // 共享所有权
    }
}

```

```

    std::cout << "引用计数: " << sensor1.use_count() << std::endl; // 2

    std::shared_ptr<Sensor> sensor3 = sensor1;
    std::cout << "引用计数: " << sensor1.use_count() << std::endl; // 3

} // sensor2 和 sensor3 销毁, 但对象还在

std::cout << "引用计数: " << sensor1.use_count() << std::endl; // 1

return 0;
} // sensor1 销毁, 引用计数变为 0, 对象被删除

```

输出:

```

Sensor IMU 创建
引用计数: 1
引用计数: 2
引用计数: 3
引用计数: 1
Sensor IMU 销毁

```

与 unique_ptr 不同, shared_ptr 可以复制:

```

auto ptr1 = std::make_shared<Sensor>("Camera");
auto ptr2 = ptr1; // 复制, 共享所有权
auto ptr3 = ptr1; // 再复制

// 三个指针指向同一个对象
std::cout << ptr1.get() << std::endl;
std::cout << ptr2.get() << std::endl; // 相同地址
std::cout << ptr3.get() << std::endl; // 相同地址

```

std::make_shared 是创建 shared_ptr 的推荐方式。它比直接使用 new 更高效, 因为它只进行一次内存分配 (同时分配对象和控制块), 而 shared_ptr<T>(new T(...)) 需要两次分配:

```

// 推荐
auto ptr1 = std::make_shared<Sensor>("Lidar");

// 不推荐 (两次内存分配)
std::shared_ptr<Sensor> ptr2(new Sensor("Radar"));

```

shared_ptr 在多个对象需要共同访问同一资源时非常有用:

```

class TargetTracker {
public:
    void setTarget(std::shared_ptr<Target> t) {
        target = t;
    }

    void track() {
        if (target) {
            // 跟踪目标
        }
    }
}

private:
    std::shared_ptr<Target> target;
};

```

```

class WeaponSystem {
public:
    void setTarget(std::shared_ptr<Target> t) {
        target = t;
    }

    void aim() {
        if (target) {
            // 瞄准目标
        }
    }
}

private:
    std::shared_ptr<Target> target;
};

int main() {
    auto target = std::make_shared<Target>(100, 200, 50);

    TargetTracker tracker;
    WeaponSystem weapon;

    // 两个系统共享同一个目标
    tracker.setTarget(target);
    weapon.setTarget(target);

    // 即使 target 变量被销毁, tracker 和 weapon 仍持有目标
    target.reset();

    // 目标仍然有效, 直到 tracker 和 weapon 都释放
    tracker.track();
    weapon.aim();

    return 0;
}

```

1.5.20.4. weak_ptr: 弱引用

`std::weak_ptr` 是 `shared_ptr` 的补充。它持有对象的非拥有（“弱”）引用——不增加引用计数，不影响对象的生命周期。`weak_ptr` 主要用于解决 `shared_ptr` 的循环引用问题，以及在不延长对象生命周期的情况下观察对象。

循环引用问题：

```

class Node {
public:
    std::string name;
    std::shared_ptr<Node> next; // 指向下一个节点
    std::shared_ptr<Node> prev; // 指向上一个节点——问题所在！

    Node(const std::string& n) : name(n) {
        std::cout << "Node " << name << " 创建" << std::endl;
    }
    ~Node() {
        std::cout << "Node " << name << " 销毁" << std::endl;
    }
}

```

```

};

int main() {
    auto node1 = std::make_shared<Node>("A");
    auto node2 = std::make_shared<Node>("B");

    node1->next = node2;
    node2->prev = node1; // 循环引用!

    return 0;
} // node1 和 node2 离开作用域, 但对象不会被销毁!
// 因为 node1 被 node2->prev 引用, node2 被 node1->next 引用
// 两个对象的引用计数都是 1, 永远不会变成 0

```

使用 `weak_ptr` 打破循环:

```

class Node {
public:
    std::string name;
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev; // 使用 weak_ptr 打破循环

    Node(const std::string& n) : name(n) {
        std::cout << "Node " << name << " 创建" << std::endl;
    }
    ~Node() {
        std::cout << "Node " << name << " 销毁" << std::endl;
    }
};

int main() {
    auto node1 = std::make_shared<Node>("A");
    auto node2 = std::make_shared<Node>("B");

    node1->next = node2;
    node2->prev = node1; // weak_ptr 不增加引用计数

    return 0;
} // 正确销毁

```

输出:

```

Node A 创建
Node B 创建
Node B 销毁
Node A 销毁

```

`weak_ptr` 不能直接访问对象, 必须先转换为 `shared_ptr`。这是因为 `weak_ptr` 所指向的对象可能已经被删除:

```

std::weak_ptr<Sensor> weakSensor;

{
    auto sensor = std::make_shared<Sensor>("GPS");
    weakSensor = sensor;

    // 方式一: lock() 返回 shared_ptr, 如果对象已删除则返回空
    if (auto sp = weakSensor.lock()) {

```

```

        std::cout << "Sensor 有效: " << sp->getName() << std::endl;
    }

    // 方式二: expired() 检查对象是否已删除
    if (!weakSensor.expired()) {
        auto sp = weakSensor.lock();
        // 使用 sp...
    }

} // sensor 销毁

// 现在对象已删除
if (weakSensor.expired()) {
    std::cout << "Sensor 已失效" << std::endl;
}

if (auto sp = weakSensor.lock()) {
    // 不会执行
} else {
    std::cout << "无法获取 Sensor" << std::endl;
}

```

`weak_ptr` 的另一个用途是缓存:持有对象的弱引用,如果对象仍存在则复用,否则重新创建:

```

class ResourceCache {
public:
    std::shared_ptr<Texture> getTexture(const std::string& name) {
        // 检查缓存
        auto it = cache.find(name);
        if (it != cache.end()) {
            // 尝试获取已缓存的纹理
            if (auto sp = it->second.lock()) {
                std::cout << "缓存命中: " << name << std::endl;
                return sp;
            }
        }
    }

    // 缓存未命中或已过期, 加载新纹理
    std::cout << "加载纹理: " << name << std::endl;
    auto texture = std::make_shared<Texture>(name);
    cache[name] = texture;
    return texture;
}

private:
    std::unordered_map<std::string, std::weak_ptr<Texture>> cache;
};

```

1.5.20.5. 智能指针与自定义删除器

默认情况下,智能指针使用 `delete` 或 `delete[]` 释放资源。但有时需要自定义释放方式,比如关闭文件句柄、释放网络连接等。可以通过自定义删除器实现:

```

// 自定义删除器: 函数
void closeFile(FILE* fp) {
    if (fp) {
        std::cout << "关闭文件" << std::endl;

```

```

        fclose(fp);
    }
}

// 使用函数作为删除器
std::unique_ptr<FILE, decltype(&closeFile)> file(fopen("data.txt", "r"),
closeFile);

// 使用 lambda 作为删除器
auto file2 = std::unique_ptr<FILE, void(*)(FILE*)>(
    fopen("log.txt", "w"),
    [](FILE* fp) {
        if (fp) {
            std::cout << "关闭日志文件" << std::endl;
            fclose(fp);
        }
    }
);

// shared_ptr 的删除器更简单（类型擦除）
std::shared_ptr<FILE> file3(
    fopen("config.txt", "r"),
    [](FILE* fp) {
        if (fp) fclose(fp);
    }
);

```

在 RoboMaster 开发中，自定义删除器可用于管理硬件资源：

```

class SerialPort {
public:
    static std::unique_ptr<SerialPort, void(*)(SerialPort*)> open(const
std::string& device) {
        SerialPort* port = new SerialPort(device);
        if (!port->isOpen()) {
            delete port;
            return {nullptr, [](SerialPort*){}};
        }
        return {port, [](SerialPort* p) {
            if (p) {
                p->close();
                delete p;
            }
        }};
    }
};

private:
    SerialPort(const std::string& device);
    void close();
    bool isOpen() const;
    int fd;
};

// 使用
auto port = SerialPort::open("/dev/ttyUSB0");
if (port) {

```

```
// 使用串口...
} // 自动关闭并释放
```

1.5.20.6. enable_shared_from_this

有时对象内部需要获取指向自身的 `shared_ptr`。直接从 `this` 创建 `shared_ptr` 是错误的，因为这会创建独立的引用计数：

```
class BadExample {
public:
    std::shared_ptr<BadExample> getShared() {
        return std::shared_ptr<BadExample>(this); // 错误!
    }
};

auto ptr1 = std::make_shared<BadExample>();
auto ptr2 = ptr1->getShared(); // ptr2 有独立的引用计数
// 当 ptr1 或 ptr2 销毁时，对象会被重复删除!
```

正确的做法是继承 `std::enable_shared_from_this`：

```
class GoodExample : public std::enable_shared_from_this<GoodExample> {
public:
    std::shared_ptr<GoodExample> getShared() {
        return shared_from_this(); // 正确!
    }

    void registerCallback() {
        // 将自身注册到某个系统
        callbackManager.register(shared_from_this());
    }
};

auto ptr1 = std::make_shared<GoodExample>();
auto ptr2 = ptr1->getShared(); // ptr2 共享 ptr1 的引用计数
std::cout << ptr1.use_count() << std::endl; // 2
```

注意：`shared_from_this()` 只能在对象已经被 `shared_ptr` 管理时调用。在构造函数中调用会导致异常。

1.5.20.7. 如何选择智能指针

选择正确的智能指针对于编写清晰、高效的代码至关重要。

使用 `unique_ptr` 当：

- 只有一个所有者需要管理对象的生命周期
- 需要工厂函数返回动态分配的对象
- 需要将对象存储在容器中，但保持独占所有权
- 作为类的成员变量，管理该类独占的资源

```
class Robot {
public:
    Robot() : controller(std::make_unique<PIDController>(1.0, 0.1, 0.05)) {}

private:
    std::unique_ptr<PIDController> controller; // Robot 独占 controller
};
```

```
std::unique_ptr<Motor> createMotor(int id) {
    return std::make_unique<Motor>(id); // 工厂函数
}
```

使用 `shared_ptr` 当:

- 多个对象需要共享同一资源的所有权
- 不确定哪个对象最后使用资源
- 需要将指针存储在多个容器或对象中

```
class Scene {
public:
    void addObject(std::shared_ptr<GameObject> obj) {
        objects.push_back(obj);
    }

private:
    std::vector<std::shared_ptr<GameObject>> objects;
};

// 同一个对象可以被多个场景共享
auto player = std::make_shared<Player>();
scene1.addObject(player);
scene2.addObject(player);
```

使用 `weak_ptr` 当:

- 需要打破 `shared_ptr` 的循环引用
- 需要观察对象但不想延长其生命周期
- 实现缓存，允许对象在不再使用时被释放

```
class Observer {
public:
    void observe(std::shared_ptr<Subject> subject) {
        this->subject = subject; // weak_ptr 不延长 subject 的生命周期
    }

    void notify() {
        if (auto sp = subject.lock()) {
            // subject 仍然存在
        }
    }
}

private:
    std::weak_ptr<Subject> subject;
};
```

使用原始指针（非拥有）当:

- 函数只是使用对象，不管理其生命周期
- 生命周期由调用者明确保证
- 需要表示可选的引用（可为 `nullptr`）

```
void processMotor(Motor* motor) { // 只使用，不拥有
    if (motor) {
        motor->run();
    }
}
```

```
}

auto motor = std::make_unique<Motor>(1);
processMotor(motor.get()); // 传递原始指针
```

使用引用当：

- 对象一定存在（不可为空）
- 只是使用对象，不管理生命周期

```
void processMotor(Motor& motor) { // 对象必须存在
    motor.run();
}
```

```
auto motor = std::make_unique<Motor>(1);
processMotor(*motor); // 传递引用
```

1.5.20.8. 性能考虑

智能指针有一些性能开销需要了解：

`unique_ptr` 几乎零开销。它的大小通常与原始指针相同（除非使用了有状态的删除器），所有操作都可以被编译器优化为与原始指针等效的代码。

`shared_ptr` 有一定开销：

- 额外的内存用于存储引用计数（控制块）
- 复制时需要原子操作更新引用计数（线程安全但有开销）
- 比 `unique_ptr` 多一次间接寻址

```
// 大小比较
std::cout << sizeof(int*) << std::endl; // 8 (64位系统)
std::cout << sizeof(std::unique_ptr<int>) << std::endl; // 8
std::cout << sizeof(std::shared_ptr<int>) << std::endl; // 16 (包含控制块指针)
std::cout << sizeof(std::weak_ptr<int>) << std::endl; // 16
```

优化建议：

- 优先使用 `unique_ptr`，只在确实需要共享所有权时使用 `shared_ptr`
- 传递智能指针时，如果不涉及所有权转移，传递引用或原始指针
- 避免不必要的 `shared_ptr` 复制，使用 `const shared_ptr<T>&` 传参
- 使用 `make_unique` 和 `make_shared` 而非直接 `new`

```
// 不好：不需要地复制 shared_ptr
void process(std::shared_ptr<Data> data) { // 复制，增加引用计数
    // ...
}
```

```
// 好：如果不需要共享所有权，传递引用
void process(const std::shared_ptr<Data>& data) { // 不复制
    // ...
}
```

```
// 更好：如果只是使用，传递原始引用
void process(Data& data) { // 不涉及智能指针
    // ...
}
```

1.5.20.9. 实际应用示例

让我们看一个综合示例，展示智能指针在 RoboMaster 开发中的应用：

```
#include <memory>
#include <vector>
#include <iostream>

// 前向声明
class RobotSystem;

// 传感器基类
class Sensor {
public:
    virtual ~Sensor() = default;
    virtual void update() = 0;
    virtual std::string getName() const = 0;
};

// 具体传感器
class IMU : public Sensor {
public:
    IMU() { std::cout << "IMU 初始化" << std::endl; }
    ~IMU() { std::cout << "IMU 关闭" << std::endl; }

    void update() override {
        std::cout << "IMU 更新数据" << std::endl;
    }

    std::string getName() const override { return "IMU"; }
};

class Camera : public Sensor {
public:
    Camera() { std::cout << "Camera 初始化" << std::endl; }
    ~Camera() { std::cout << "Camera 关闭" << std::endl; }

    void update() override {
        std::cout << "Camera 捕获图像" << std::endl;
    }

    std::string getName() const override { return "Camera"; }
};

// 目标（可能被多个系统共享）
class Target : public std::enable_shared_from_this<Target> {
public:
    Target(double x, double y) : x(x), y(y) {
        std::cout << "Target 创建于 (" << x << ", " << y << ")" << std::endl;
    }

    ~Target() {
        std::cout << "Target 销毁" << std::endl;
    }

    void updatePosition(double newX, double newY) {
```

```

        x = newX;
        y = newY;
    }

    std::shared_ptr<Target> getPtr() {
        return shared_from_this();
    }

private:
    double x, y;
};

// 追踪器（观察目标但不拥有）
class Tracker {
public:
    void setTarget(std::shared_ptr<Target> t) {
        target = t; // weak_ptr, 不延长目标生命周期
    }

    void track() {
        if (auto t = target.lock()) {
            std::cout << "正在追踪目标" << std::endl;
        } else {
            std::cout << "目标已丢失" << std::endl;
        }
    }

private:
    std::weak_ptr<Target> target;
};

// 机器人系统
class RobotSystem {
public:
    RobotSystem() {
        std::cout << "==== 机器人系统启动 ===" << std::endl;
    }

    ~RobotSystem() {
        std::cout << "==== 机器人系统关闭 ===" << std::endl;
    }

// 添加传感器（独占所有权）
void addSensor(std::unique_ptr<Sensor> sensor) {
    std::cout << "添加传感器: " << sensor->getName() << std::endl;
    sensors.push_back(std::move(sensor));
}

// 设置当前目标（共享所有权）
void setTarget(std::shared_ptr<Target> t) {
    currentTarget = t;
    tracker.setTarget(t);
}

// 更新所有传感器

```

```

void updateSensors() {
    for (auto& sensor : sensors) {
        sensor->update();
    }
}

// 追踪目标
void trackTarget() {
    tracker.track();
}

private:
    std::vector<std::unique_ptr<Sensor>> sensors; // 独占传感器
    std::shared_ptr<Target> currentTarget; // 共享目标
    Tracker tracker;
};

int main() {
    auto robot = std::make_unique<RobotSystem>();

    // 添加传感器（转移所有权给 robot）
    robot->addSensor(std::make_unique<IMU>());
    robot->addSensor(std::make_unique<Camera>());

    // 创建共享目标
    auto target = std::make_shared<Target>(100, 200);
    robot->setTarget(target);

    // 运行
    robot->updateSensors();
    robot->trackTarget();

    std::cout << "\n--- 目标引用计数: " << target.use_count() << " ---\n" <<
    std::endl;

    // 释放外部目标引用
    target.reset();
    std::cout << "外部目标引用已释放" << std::endl;

    // 目标仍然存在 (robot 持有)
    robot->trackTarget();

    std::cout << "\n--- 关闭机器人 ---\n" << std::endl;
    robot.reset(); // 销毁机器人，所有资源自动释放

    return 0;
}

```

输出：

```

==== 机器人系统启动 ====
IMU 初始化
添加传感器: IMU
Camera 初始化
添加传感器: Camera
Target 创建于 (100, 200)

```

```
IMU 更新数据
Camera 捕获图像
正在追踪目标
```

```
--- 目标引用计数: 2 ---
```

```
外部目标引用已释放
正在追踪目标
```

```
--- 关闭机器人 ---
```

```
==== 机器人系统关闭 ===
```

```
Target 销毁
Camera 关闭
IMU 关闭
```

这个示例展示了三种智能指针的典型用法：`unique_ptr` 管理独占资源（传感器），`shared_ptr` 管理共享资源（目标），`weak_ptr` 观察资源而不影响生命周期（追踪器）。

智能指针是现代 C++ 内存管理的基石。通过使用智能指针，你可以编写出更安全、更清晰的代码，几乎完全消除内存泄漏和悬空指针的风险。记住：优先使用 `unique_ptr`，需要共享时使用 `shared_ptr`，打破循环或观察时使用 `weak_ptr`。下一节我们将学习移动语义，它与智能指针密切相关，是理解现代 C++ 性能优化的关键。

1.5.21. 右值引用与移动语义

在前面的章节中，我们多次提到“移动”这个概念：`unique_ptr` 不能复制但可以移动，`std::move` 可以将对象的所有权转移给另一个对象。那么，移动到底是什么？为什么需要移动？这一节将深入探讨 C++11 引入的右值引用（rvalue reference）和移动语义（move semantics），它们是现代 C++ 性能优化的核心技术。

1.5.21.1. 从问题出发

假设有一个存储大量数据的类：

```
class BigData {
public:
    BigData(size_t size) : size(size), data(new int[size]) {
        std::cout << "构造: 分配 " << size << " 个整数" << std::endl;
    }

    // 拷贝构造函数
    BigData(const BigData& other) : size(other.size), data(new int[other.size])
    {
        std::copy(other.data, other.data + size, data);
        std::cout << "拷贝构造: 复制 " << size << " 个整数" << std::endl;
    }

    ~BigData() {
        delete[] data;
        std::cout << "析构: 释放内存" << std::endl;
    }

private:
    size_t size;
```

```

    int* data;
};

BigData createBigData() {
    BigData temp(1000000); // 创建临时对象
    return temp;           // 返回时会拷贝
}

int main() {
    BigData data = createBigData(); // 又一次拷贝?
    return 0;
}

```

在 C++11 之前，这段代码可能涉及多次昂贵的深拷贝。`createBigData` 返回时，`temp` 的内容被复制到返回值；然后返回值又被复制到 `data`。对于包含百万个整数的对象，这是巨大的浪费。

问题的关键在于：`temp` 是一个即将销毁的临时对象，我们完全可以“偷走”它的数据，而不是复制。这就是移动语义的核心思想——对于即将消亡的对象，转移其资源而非复制。

1.5.21.2. 左值与右值

要理解移动语义，首先需要理解左值（lvalue）和右值（rvalue）的概念。

简单来说，左值是有持久身份的表达式——它有名字，可以取地址，在表达式结束后仍然存在。右值是临时的、即将消亡的表达式——通常没有名字，不能取地址，表达式结束后就不存在了。

```

int x = 10;           // x 是左值
int y = x + 5;       // x 是左值, x + 5 是右值, y 是左值
int z = x + y;       // x 和 y 是左值, x + y 是右值

int* p = &x;          // 正确, x 是左值, 可以取地址
// int* q = &(x + 5); // 错误! x + 5 是右值, 不能取地址

std::string s1 = "hello";           // s1 是左值
std::string s2 = s1 + " world";     // s1 是左值, s1 + " world" 是右值
std::string s3 = std::string("hi"); // std::string("hi") 是右值

```

函数返回值（非引用）是右值：

```

std::string getName() {
    return "Robot";
}

std::string name = getName(); // getName() 返回的是右值

```

直观判断方法：能放在赋值运算符左边的通常是左值，只能放在右边的是右值。能取地址的是左值，不能取地址的是右值。

1.5.21.3. 右值引用

C++11 引入了右值引用，用 `&&` 表示。右值引用只能绑定到右值：

```

int x = 10;

int& lref = x;      // 左值引用, 绑定到左值
// int& lref2 = 10; // 错误! 左值引用不能绑定到右值

int&& rref = 10;    // 右值引用, 绑定到右值

```

```

int&& rref2 = x + 5; // 正确, x + 5 是右值
// int&& rref3 = x; // 错误! 右值引用不能绑定到左值

// 特殊情况: const 左值引用可以绑定到右值
const int& cref = 10; // 正确, 这是历史遗留特性

```

右值引用的意义在于：它让我们能够区分“传入的是即将消亡的临时对象”和“传入的是需要保留的持久对象”。基于这个区分，我们可以对临时对象采取不同的处理策略——移动而非复制。

1.5.21.4. 移动构造函数与移动赋值运算符

有了右值引用，我们可以为类定义移动构造函数和移动赋值运算符。它们接受右值引用参数，在实现中“窃取”源对象的资源：

```

class BigData {
public:
    // 普通构造函数
    BigData(size_t size) : size(size), data(new int[size]) {
        std::cout << "构造: 分配 " << size << " 个整数" << std::endl;
    }

    // 拷贝构造函数
    BigData(const BigData& other) : size(other.size), data(new int[other.size])
    {
        std::copy(other.data, other.data + size, data);
        std::cout << "拷贝构造: 复制 " << size << " 个整数" << std::endl;
    }

    // 移动构造函数
    BigData(BigData&& other) noexcept
        : size(other.size), data(other.data) // 直接接管资源
    {
        other.size = 0;
        other.data = nullptr; // 将源对象置于有效但空的状态
        std::cout << "移动构造: 转移资源" << std::endl;
    }

    // 拷贝赋值运算符
    BigData& operator=(const BigData& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new int[size];
            std::copy(other.data, other.data + size, data);
            std::cout << "拷贝赋值: 复制 " << size << " 个整数" << std::endl;
        }
        return *this;
    }

    // 移动赋值运算符
    BigData& operator=(BigData&& other) noexcept {
        if (this != &other) {
            delete[] data; // 释放自己的资源
            size = other.size; // 接管源对象的资源
            data = other.data;
            other.size = 0; // 将源对象置于有效但空的状态
        }
        return *this;
    }
}

```

```

        other.data = nullptr;
        std::cout << "移动赋值: 转移资源" << std::endl;
    }
    return *this;
}

~BigData() {
    delete[] data;
    if (size > 0) {
        std::cout << "析构: 释放 " << size << " 个整数" << std::endl;
    } else {
        std::cout << "析构: 空对象" << std::endl;
    }
}

private:
    size_t size;
    int* data;
};

```

移动操作的关键点：

1. 参数是右值引用 `T&&`
2. 直接接管源对象的资源（指针、句柄等），而不是复制
3. 将源对象置于“有效但未指定”的状态（通常是空状态），确保其析构函数能安全执行
4. 标记为 `noexcept`，这对于标准库容器的优化很重要

现在让我们看看移动语义的效果：

```

BigData createBigData() {
    BigData temp(1000000);
    return temp;
}

int main() {
    std::cout << "==== 创建临时对象 ===" << std::endl;
    BigData data = createBigData();

    std::cout << "\n==== 从右值移动 ===" << std::endl;
    BigData data2 = BigData(500000); // 临时对象，触发移动构造

    std::cout << "\n==== 移动赋值 ===" << std::endl;
    data2 = BigData(200000); // 临时对象，触发移动赋值

    std::cout << "\n==== 程序结束 ===" << std::endl;
    return 0;
}

```

移动操作只是转移指针，时间复杂度 $O(1)$ ，而拷贝操作需要分配内存和复制数据，时间复杂度 $O(n)$ 。对于大型对象，性能差异是巨大的。

1.5.21.5. `std::move`

前面的例子中，移动语义自动应用于临时对象（右值）。但有时我们想对左值也使用移动语义——比如我们知道某个对象之后不再使用，想把它的资源转移出去。

`std::move` 的作用是将左值转换为右值引用，从而允许移动操作：

```

#include <utility> // std::move

int main() {
    BigData data1(1000000);

    // BigData data2 = data1; // 拷贝构造

    BigData data2 = std::move(data1); // 移动构造
    // data1 现在处于有效但未指定的状态（通常是空）
    // 不应该再使用 data1 的值

    return 0;
}

```

理解 `std::move` 的关键：它本身不移动任何东西！它只是一个类型转换，将表达式的类型从左值转换为右值引用。真正的移动发生在移动构造函数或移动赋值运算符中。

```

// std::move 的简化实现
template <typename T>
typename std::remove_reference<T>::type&& move(T&& arg) noexcept {
    return static_cast<typename std::remove_reference<T>::type&&>(arg);
}

```

使用 `std::move` 后的对象处于“有效但未指定”的状态。这意味着：

- 对象仍然有效，可以被销毁、被赋新值
- 但不应该依赖它的值，因为值已经被转移走了

```

std::string s1 = "Hello, World!";
std::string s2 = std::move(s1);

std::cout << s2 << std::endl; // "Hello, World!"
// std::cout << s1 << std::endl; // 可能输出空字符串，但不要依赖这个行为

s1 = "New Value"; // 可以重新赋值
std::cout << s1 << std::endl; // "New Value"

```

1.5.21.6. 何时使用 `std::move`

`std::move` 应该谨慎使用。以下是一些适当使用的场景：

将对象传递给会“消费”它的函数：

```

void takeOwnership(std::unique_ptr<Motor> motor) {
    // 使用 motor...
}

auto motor = std::make_unique<Motor>(1);
takeOwnership(std::move(motor)); // 转移所有权
// motor 现在为空

```

从函数返回局部对象时通常不需要 `std::move`，编译器会自动应用返回值优化（RVO）或移动：

```

BigData createData() {
    BigData data(1000);
    return data; // 不要写 return std::move(data)，这可能阻止 RVO
}

```

将容器中的元素移出：

```

std::vector<std::string> names = {"Alice", "Bob", "Charlie"};
std::string first = std::move(names[0]);
// names[0] 现在是空字符串（或未指定状态）

```

在类的成员函数中转移成员：

```

class Container {
public:
    std::vector<int> extractData() {
        return std::move(data); // 转移而非复制
        // 之后 data 为空，对象状态改变
    }

private:
    std::vector<int> data;
};

```

不应该对 `const` 对象使用 `std::move`——结果仍然是 `const`，会匹配拷贝构造函数而非移动构造函数：

```

const std::string s1 = "Hello";
std::string s2 = std::move(s1); // 实际上是拷贝！

```

1.5.21.7. 完美转发与万能引用

在模板编程中，`T&&` 不一定是右值引用。当 `T` 是模板参数时，`T&&` 是“万能引用”（也称“转发引用”），它可以绑定到左值或右值：

```

template <typename T>
void wrapper(T&& arg) { // T&& 是万能引用
    // arg 是左值还是右值取决于调用时传入的参数
}

int x = 10;
wrapper(x); // T 推导为 int&, arg 类型是 int& (左值引用)
wrapper(10); // T 推导为 int, arg 类型是 int&& (右值引用)
wrapper(x + 5); // T 推导为 int, arg 类型是 int&&

```

在模板中，为了保持参数的左值/右值属性进行转发，使用 `std::forward`：

```

template <typename T>
void relay(T&& arg) {
    process(std::forward<T>(arg)); // 完美转发
}

void process(int& x) { std::cout << "左值版本" << std::endl; }
void process(int&& x) { std::cout << "右值版本" << std::endl; }

int main() {
    int x = 10;
    relay(x); // 调用 process(int&)
    relay(10); // 调用 process(int&&)
    relay(x + 5); // 调用 process(int&&)
    return 0;
}

```

`std::forward` 与 `std::move` 的区别：

- `std::move` 无条件地将参数转为右值引用
- `std::forward` 根据模板参数有条件地转换，保持原来的左值/右值属性

这在实现通用工厂函数或包装器时非常有用：

```
template <typename T, typename... Args>
std::unique_ptr<T> makeUnique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

1.5.21.8. 移动语义与标准库

标准库的容器和类型都支持移动语义，这使得它们在很多场景下效率更高。

`std::vector` 在扩容时，如果元素类型的移动构造函数是 `noexcept` 的，会使用移动而非复制：

```
std::vector<BigData> vec;
vec.reserve(2);

vec.push_back(BigData(1000));    // 移动临时对象
vec.push_back(BigData(2000));    // 移动临时对象

// 如果需要扩容，且 BigData 的移动构造是 noexcept
// 则现有元素会被移动而非复制
vec.push_back(BigData(3000));
```

这就是为什么移动构造函数应该标记为 `noexcept`——它允许标准库进行优化。

`std::string` 和其他标准库类型也支持移动：

```
std::string s1 = "A very long string that exceeds small string optimization";
std::string s2 = std::move(s1); // O(1) 移动，而非 O(n) 复制

std::vector<int> v1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<int> v2 = std::move(v1); // O(1) 移动
```

1.5.21.9. 五法则

C++11 之后，“三法则”扩展为“五法则”：如果你需要自定义析构函数、拷贝构造函数或拷贝赋值运算符中的任何一个，你可能也需要自定义其他四个（包括移动构造函数和移动赋值运算符）。

完整的五法则示例：

```
class Resource {
public:
    // 构造函数
    Resource(size_t size) : size(size), data(new int[size]) {}

    // 析构函数
    ~Resource() {
        delete[] data;
    }

    // 拷贝构造函数
    Resource(const Resource& other)
        : size(other.size), data(new int[other.size])
    {
        std::copy(other.data, other.data + size, data);
    }

    // 拷贝赋值运算符
    Resource& operator=(const Resource& other) {
        if (this != &other) {
```

```

        delete[] data;
        size = other.size;
        data = new int[size];
        std::copy(other.data, other.data + size, data);
    }
    return *this;
}

// 移动构造函数
Resource(Resource&& other) noexcept
    : size(other.size), data(other.data)
{
    other.size = 0;
    other.data = nullptr;
}

// 移动赋值运算符
Resource& operator=(Resource&& other) noexcept {
    if (this != &other) {
        delete[] data;
        size = other.size;
        data = other.data;
        other.size = 0;
        other.data = nullptr;
    }
    return *this;
}

private:
    size_t size;
    int* data;
};

```

或者使用 copy-and-swap 惯用法统一处理拷贝和移动赋值：

```

class Resource {
public:
    Resource(size_t size) : size(size), data(new int[size]) {}
    ~Resource() { delete[] data; }

    Resource(const Resource& other)
        : size(other.size), data(new int[other.size])
    {
        std::copy(other.data, other.data + size, data);
    }

    Resource(Resource&& other) noexcept
        : size(other.size), data(other.data)
    {
        other.size = 0;
        other.data = nullptr;
    }

    // 统一的赋值运算符（拷贝和移动）
    Resource& operator=(Resource other) noexcept { // 按值传递
        swap(*this, other);
        return *this;
    }
}

```

```

    }

    friend void swap(Resource& a, Resource& b) noexcept {
        using std::swap;
        swap(a.size, b.size);
        swap(a.data, b.data);
    }

private:
    size_t size;
    int* data;
};

```

1.5.21.10. 实际应用示例

让我们看看移动语义在 RoboMaster 开发中的应用。

高效的图像处理管道：

```

class Frame {
public:
    Frame(int width, int height)
        : width(width), height(height),
          data(new uint8_t[width * height * 3])
    {
        std::cout << "分配帧: " << width << "x" << height << std::endl;
    }

    Frame(Frame&& other) noexcept
        : width(other.width), height(other.height), data(other.data)
    {
        other.width = 0;
        other.height = 0;
        other.data = nullptr;
        std::cout << "移动帧" << std::endl;
    }

    Frame& operator=(Frame&& other) noexcept {
        if (this != &other) {
            delete[] data;
            width = other.width;
            height = other.height;
            data = other.data;
            other.width = 0;
            other.height = 0;
            other.data = nullptr;
        }
        return *this;
    }

    // 禁止拷贝（图像数据太大）
    Frame(const Frame&) = delete;
    Frame& operator=(const Frame&) = delete;

    ~Frame() { delete[] data; }

private:

```

```

        int width, height;
        uint8_t* data;
    };

    class FrameProcessor {
public:
    Frame process(Frame frame) { // 按值传递, 自动移动
        // 处理帧...
        return frame; // 返回处理后的帧, 自动移动
    }
};

// 使用
Frame captureFrame() {
    return Frame(1920, 1080);
}

int main() {
    FrameProcessor processor;

    Frame frame = captureFrame(); // 移动 (或 RVO)
    frame = processor.process(std::move(frame)); // 移动进, 移动出

    return 0;
}

```

消息队列的高效传递:

```

class Message {
public:
    Message(std::string topic, std::vector<uint8_t> payload)
        : topic(std::move(topic)), payload(std::move(payload)),
          timestamp(std::chrono::steady_clock::now())
    {
    }

    // 默认移动操作足够
    Message(Message&&) = default;
    Message& operator=(Message&&) = default;

    // 禁止拷贝
    Message(const Message&) = delete;
    Message& operator=(const Message&) = delete;

    const std::string& getTopic() const { return topic; }
    const std::vector<uint8_t>& getPayload() const { return payload; }

private:
    std::string topic;
    std::vector<uint8_t> payload;
    std::chrono::steady_clock::time_point timestamp;
};

class MessageQueue {
public:
    void push(Message msg) { // 按值传递, 调用者可以移动或拷贝

```

```

        std::lock_guard<std::mutex> lock(mutex);
        queue.push(std::move(msg)); // 移动到队列
    }

    std::optional<Message> pop() {
        std::lock_guard<std::mutex> lock(mutex);
        if (queue.empty()) return std::nullopt;

        Message msg = std::move(queue.front()); // 移动出队列
        queue.pop();
        return msg; // 移动返回
    }

private:
    std::queue<Message> queue;
    std::mutex mutex;
};

// 使用
void producer(MessageQueue& mq) {
    std::vector<uint8_t> data(1024);
    mq.push(Message("sensor imu", std::move(data))); // 高效传递
}

void consumer(MessageQueue& mq) {
    if (auto msg = mq.pop()) {
        // 处理消息...
    }
}

```

高效的返回复杂对象：

```

struct DetectionResult {
    std::vector<Target> targets;
    Frame processedFrame;
    double processingTime;

    // 默认移动操作
    DetectionResult(DetectionResult&&) = default;
    DetectionResult& operator=(DetectionResult&&) = default;
};

DetectionResult detectTargets(Frame frame) {
    DetectionResult result;

    // 处理...
    result.processedFrame = std::move(frame);
    result.targets.emplace_back(/* ... */);
    result.processingTime = 0.016;

    return result; // 移动返回 (或 NRVO)
}

int main() {
    Frame frame = captureFrame();
    DetectionResult result = detectTargets(std::move(frame));
    // frame 已被移动, result 包含处理结果
}

```

```
        return 0;
    }
```

1.5.21.11. 常见错误与最佳实践

不要过度使用 `std::move`。以下情况不需要 `std::move`:

```
// 错误: 返回局部变量不需要 move, 可能阻止 RVO
std::string getName() {
    std::string name = "Robot";
    return std::move(name); // 不要这样做!
}
```

```
// 正确
std::string getName() {
    std::string name = "Robot";
    return name; // 编译器会自动优化
}
```

不要移动后继续使用:

```
std::vector<int> v1 = {1, 2, 3};
std::vector<int> v2 = std::move(v1);

// 危险! v1 的状态未指定
// for (int x : v1) { ... } // 不要这样做

// 可以重新赋值
v1 = {4, 5, 6}; // 现在 v1 又有效了
```

记得标记 `noexcept`:

```
// 好: 标记 noexcept, 允许标准库优化
Resource(Resource&& other) noexcept;

// 不好: 没有 noexcept, vector 扩容时会复制而非移动
Resource(Resource&& other);
```

使用 `= default` 生成移动操作 (如果合适):

```
class Simple {
public:
    Simple() = default;
    Simple(Simple&&) = default; // 编译器生成移动构造
    Simple& operator=(Simple&&) = default; // 编译器生成移动赋值

private:
    std::string name;
    std::vector<int> data;
    // 所有成员都支持移动, 默认实现就够了
};
```

移动语义是现代 C++ 的重要特性, 它使得资源密集型对象的传递变得高效。理解左值、右值、右值引用和 `std::move` 的含义, 可以帮助你写出更高效的代码, 并更好地理解标准库的行为。在 RoboMaster 开发中, 图像处理、消息传递、大型数据结构等场景都能从移动语义中受益。

1.5.22. Lambda 表达式

在前面的章节中，我们已经多次使用 lambda 表达式：作为 STL 算法的谓词、作为线程的执行函数、作为条件变量的等待条件。Lambda 表达式是 C++11 引入的重要特性，它允许在代码中就地定义匿名函数，使代码更加简洁和直观。在 RoboMaster 开发中，lambda 表达式尤其重要——ROS 的回调机制大量使用 lambda 来处理消息、定时器和服务响应。掌握 lambda 表达式，是编写现代 C++ 代码和 ROS 程序的必备技能。

1.5.22.1. 为什么需要 Lambda

假设我们要对一组目标按距离排序。使用传统方法，需要单独定义比较函数：

```
bool compareByDistance(const Target& a, const Target& b) {
    return a.distance < b.distance;
}

void processTargets(std::vector<Target>& targets) {
    std::sort(targets.begin(), targets.end(), compareByDistance);
}
```

这种方式有几个问题：比较函数定义在使用位置之外，阅读代码时需要跳转查看；如果比较逻辑只用一次，单独定义函数显得冗余；如果需要访问局部变量（如阈值），函数就无法直接使用。

Lambda 表达式解决了这些问题：

```
void processTargets(std::vector<Target>& targets) {
    std::sort(targets.begin(), targets.end(),
        [] (const Target& a, const Target& b) {
            return a.distance < b.distance;
        });
}
```

比较逻辑就在使用的地方定义，代码意图一目了然。

1.5.22.2. Lambda 的基本语法

Lambda 表达式的完整语法如下：

```
[捕获列表](参数列表) mutable 异常说明 -> 返回类型 { 函数体 }
```

其中很多部分是可选的，最简形式只需要捕获列表和函数体：

```
[]{ std::cout << "Hello, Lambda!" << std::endl; }
```

让我们逐一了解各个部分：

```
// 完整示例
auto lambda = [x, &y](int a, int b) mutable noexcept -> int {
    x++; // mutable 允许修改按值捕获的变量
    return a + b + x + y;
};

// 各部分说明:
// [x, &y] - 捕获列表: x 按值捕获, y 按引用捕获
// (int a, int b) - 参数列表: 接受两个 int 参数
// mutable - 允许修改按值捕获的变量 (可选)
// noexcept - 异常说明 (可选)
// -> int - 返回类型 (通常可省略, 编译器自动推导)
// { ... } - 函数体
```

常见的 lambda 形式：

```
// 无参数、无捕获
auto greet = []() { std::cout << "Hello!" << std::endl; };
greet();

// 参数列表，无捕获
auto add = [](int a, int b) { return a + b; };
int sum = add(3, 4); // 7

// 捕获局部变量
int factor = 10;
auto multiply = [factor](int x) { return x * factor; };
int result = multiply(5); // 50

// 省略空参数列表
auto sayHi = [] { std::cout << "Hi!" << std::endl; };
sayHi();
```

返回类型通常可以省略，编译器会自动推导。但在某些复杂情况下需要显式指定：

```
// 编译器自动推导返回类型为 int
auto simple = [](int x) { return x * 2; };

// 多个 return 语句且类型不一致时，需要显式指定
auto conditional = [](bool flag) -> double {
    if (flag) return 1;           // int
    else return 3.14;           // double
};
```

1.5.22.3. 捕获列表

捕获列表是 lambda 最重要的特性之一，它决定了 lambda 如何访问外部作用域的变量。

按值捕获创建变量的副本，lambda 内部使用的是副本，原变量不受影响：

```
int x = 10;

auto byValue = [x]() {
    // x = 20; // 错误！默认情况下不能修改按值捕获的变量
    std::cout << x << std::endl; // 10
};

x = 100;
byValue(); // 仍然输出 10，因为捕获的是副本
```

按引用捕获使用变量的引用，修改会影响原变量：

```
int x = 10;

auto byRef = [&x]() {
    x = 20; // 修改原变量
};

byRef();
std::cout << x << std::endl; // 20
```

可以混合使用多种捕获方式：

```
int a = 1, b = 2, c = 3;
```

```

// 分别指定每个变量的捕获方式
auto mixed = [a, &b, c]() {
    // a 是值捕获
    // b 是引用捕获
    // c 是值捕获
};

// 默认按值捕获所有变量, b 按引用
auto defaultValue = [=, &b]() {
    // 所有变量按值捕获, 除了 b 按引用
};

// 默认按引用捕获所有变量, a 按值
auto defaultRef = [&, a]() {
    // 所有变量按引用捕获, 除了 a 按值
};

```

捕获列表的各种形式：

[]	// 不捕获任何变量
[x]	// 按值捕获 x
[&x]	// 按引用捕获 x
[=]	// 按值捕获所有使用的局部变量
[&]	// 按引用捕获所有使用的局部变量
[=, &x]	// 默认按值, x 按引用
[&, x]	// 默认按引用, x 按值
[this]	// 捕获当前对象的 this 指针
[*this]	// C++17: 按值捕获当前对象 (复制)

使用 `mutable` 关键字可以修改按值捕获的变量（修改的是副本）：

```

int x = 10;

auto mutableLambda = [x]() mutable {
    x++; // 允许修改, 但修改的是副本
    return x;
};

std::cout << mutableLambda() << std::endl; // 11
std::cout << mutableLambda() << std::endl; // 12 (lambda 内部的 x 保持状态)
std::cout << x << std::endl; // 10 (原变量不变)

```

1.5.22.4. 捕获的注意事项

捕获引用时要特别注意变量的生命周期。如果 lambda 存活时间超过被捕获变量，会产生悬空引用：

```

std::function<int()> createCounter() {
    int count = 0;
    // 危险! count 在函数返回后被销毁
    return [&count]() { return ++count; };
}

auto counter = createCounter();
// counter(); // 未定义行为! count 已经不存在

```

解决方案是按值捕获，或者使用 `mutable`：

```

std::function<int()> createCounter() {
    int count = 0;
    // 安全: 按值捕获, 使用 mutable 允许修改
    return [count]() mutable { return ++count; };
}

auto counter = createCounter();
std::cout << counter() << std::endl; // 1
std::cout << counter() << std::endl; // 2
std::cout << counter() << std::endl; // 3

```

在类的成员函数中, 捕获 this 可以访问成员变量:

```

class Robot {
public:
    void startMonitoring() {
        // 捕获 this, 可以访问成员变量
        auto monitor = [this]() {
            while (running) {
                std::cout << "Position: " << position << std::endl;
                std::this_thread::sleep_for(std::chrono::seconds(1));
            }
        };
        monitorThread = std::thread(monitor);
    }

    void stop() {
        running = false;
        if (monitorThread.joinable()) {
            monitorThread.join();
        }
    }
};

private:
    std::atomic<bool> running{true};
    double position = 0.0;
    std::thread monitorThread;
};

```

C++17 的 [*this] 按值捕获整个对象, 这在对象可能在 lambda 执行前被销毁的情况下很有用:

```

class Processor {
public:
    auto createTask() {
        // C++17: 复制整个对象
        return [*this]() {
            // 使用对象的副本, 即使原对象已销毁也安全
            return this->process();
        };
    }

private:
    int process() const { return data * 2; }
    int data = 42;
};

```

1.5.22.5. Lambda 的类型与存储

每个 lambda 表达式都有唯一的、无名的类型（闭包类型）。即使两个 lambda 看起来完全相同，它们的类型也不同：

```
auto lambda1 = [](int x) { return x * 2; };
auto lambda2 = [](int x) { return x * 2; };

// lambda1 和 lambda2 的类型不同!
// decltype(lambda1) != decltype(lambda2)
```

因此，通常使用 `auto` 来声明 lambda 变量。如果需要存储 lambda 或将其作为参数传递，可以使用 `std::function`：

```
#include <functional>

// std::function 可以存储任何可调用对象
std::function<int(int)> func;

func = [](int x) { return x * 2; };
std::cout << func(5) << std::endl; // 10

func = [](int x) { return x * x; };
std::cout << func(5) << std::endl; // 25

// 作为函数参数
void applyToAll(std::vector<int>& vec, std::function<int(int)> transform) {
    for (int& x : vec) {
        x = transform(x);
    }
}
```

不捕获任何变量的 lambda 可以转换为函数指针：

```
// 无捕获 lambda 可以转换为函数指针
int (*funcPtr)(int, int) = [](int a, int b) { return a + b; };

std::cout << funcPtr(3, 4) << std::endl; // 7

// 有捕获的 lambda 不能转换为函数指针
int factor = 2;
// int (*ptr)(int) = [factor](int x) { return x * factor; }; // 错误!
```

1.5.22.6. 泛型 Lambda (C++14)

C++14 允许 lambda 的参数使用 `auto`，创建泛型 lambda：

```
// 泛型 lambda: 参数类型自动推导
auto print = [](const auto& x) {
    std::cout << x << std::endl;
};

print(42);           // int
print(3.14);         // double
print("hello");       // const char*
print(std::string("world")); // std::string

// 多个 auto 参数
auto add = [](auto a, auto b) {
```

```

        return a + b;
    };

    std::cout << add(1, 2) << std::endl;           // 3 (int)
    std::cout << add(1.5, 2.5) << std::endl;     // 4.0 (double)
    std::cout << add(std::string("Hello, "), "World!") << std::endl; // "Hello,
    World!"

```

泛型 lambda 本质上是带有模板化 operator() 的函数对象。

1.5.22.7. ROS 回调中的 Lambda

在 ROS (Robot Operating System) 开发中，lambda 表达式是处理回调的首选方式。相比传统的成员函数回调，lambda 更加灵活和直观。

ROS 2 订阅器回调：

```

#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/imu.hpp>
#include <geometry_msgs/msg/twist.hpp>

class RobotNode : public rclcpp::Node {
public:
    RobotNode() : Node("robot_node") {
        // 使用 lambda 作为订阅回调
        imu_sub_ = this->create_subscription<sensor_msgs::msg::Imu>(
            "imu/data", 10,
            [this](const sensor_msgs::msg::Imu::SharedPtr msg) {
                // 直接访问成员变量
                latest_imu_ = *msg;
                processIMU(msg);
            }
        );
    }

    // 另一个订阅器
    cmd_sub_ = this->create_subscription<geometry_msgs::msg::Twist>(
        "cmd_vel", 10,
        [this](const geometry_msgs::msg::Twist::SharedPtr msg) {
            target_velocity_ = *msg;
        }
    );

    // 定时器回调
    timer_ = this->create_wall_timer(
        std::chrono::milliseconds(10),
        [this]() {
            controlLoop();
        }
    );
}

private:
    void processIMU(const sensor_msgs::msg::Imu::SharedPtr msg) {
        // 处理 IMU 数据
    }

    void controlLoop() {

```

```

        // 控制循环
    }

    sensor_msgs::msg::Imu latest_imu_;
    geometry_msgs::msg::Twist target_velocity_;

    rclcpp::Subscription<sensor_msgs::msg::Imu>::SharedPtr imu_sub_;
    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr cmd_sub_;
    rclcpp::TimerBase::SharedPtr timer_;
};


```

服务回调:

```

#include <example_interfaces/srv/add_two_ints.hpp>

class CalculatorNode : public rclcpp::Node {
public:
    CalculatorNode() : Node("calculator") {
        service_ = this->create_service<example_interfaces::srv::AddTwoInts>(
            "add_two_ints",
            [this]{
                const
                std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
                std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>
                response
            } {
                response->sum = request->a + request->b;
                RCLCPP_INFO(this->get_logger(), "Request: %ld + %ld = %ld",
                            request->a, request->b, response->sum);
            }
        );
    }

private:
    rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service_;
};


```

ROS 1 风格 (使用 roscpp):

```

#include <ros/ros.h>
#include <sensor_msgs/Image.h>

class VisionNode {
public:
    VisionNode() {
        ros::NodeHandle nh;

        // boost::bind 方式 (传统)
        // image_sub_ = nh.subscribe("camera/image", 1,
        // &VisionNode::imageCallback, this);

        // Lambda 方式 (更现代)
        image_sub_ = nh.subscribe<sensor_msgs::Image>(
            "camera/image", 1,
            [this](const sensor_msgs::ImageConstPtr& msg) {
                processImage(msg);
            }
        );
    }
};


```

```

        );
    }

private:
    void processImage(const sensor_msgs::ImageConstPtr& msg) {
        // 图像处理
    }

    ros::Subscriber image_sub_;
};
```

1.5.22.8. STL 算法中的 Lambda

Lambda 在 STL 算法中广泛使用。这里是一些常见模式：

```

std::vector<Target> targets = getTargets();

// 排序
std::sort(targets.begin(), targets.end(),
    [](const Target& a, const Target& b) {
        return a.priority > b.priority; // 按优先级降序
});

// 查找
auto it = std::find_if(targets.begin(), targets.end(),
    [](const Target& t) {
        return t.confidence > 0.9 && t.isEnemy;
});

// 计数
int highConfCount = std::count_if(targets.begin(), targets.end(),
    [](const Target& t) {
        return t.confidence > 0.8;
});

// 过滤 (移除不满足条件的)
targets.erase(
    std::remove_if(targets.begin(), targets.end(),
        [](const Target& t) {
            return t.confidence < 0.5;
        }),
    targets.end()
);

// 变换
std::vector<double> distances;
std::transform(targets.begin(), targets.end(), std::back_inserter(distances),
    [](const Target& t) {
        return t.distance;
});

// 累加
double totalPriority = std::accumulate(targets.begin(), targets.end(), 0.0,
    [](double sum, const Target& t) {
        return sum + t.priority;
});
```

```

// 全部满足条件?
bool allValid = std::all_of(targets.begin(), targets.end(),
    [](const Target& t) {
        return t.isValid();
    });

// 任一满足条件?
bool anyEnemy = std::any_of(targets.begin(), targets.end(),
    [](const Target& t) {
        return t.isEnemy();
    });

```

使用捕获访问外部变量:

```

double maxDistance = 100.0;
double minConfidence = 0.7;

auto validTargets = targets;
validTargets.erase(
    std::remove_if(validTargets.begin(), validTargets.end(),
        [maxDistance, minConfidence](const Target& t) {
            return t.distance > maxDistance || t.confidence < minConfidence;
        }),
    validTargets.end()
);

```

1.5.22.9. 异步操作与 Lambda

Lambda 与异步操作配合使用非常方便:

```

#include <future>

// std::async 与 lambda
auto future = std::async(std::launch::async, []() {
    // 耗时操作
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 42;
});

// 做其他事情...

int result = future.get(); // 等待并获取结果

// 并行处理多个任务
std::vector<std::future<DetectionResult>> futures;

for (const auto& frame : frames) {
    futures.push_back(std::async(std::launch::async,
        [&detector, frame]() {
            return detector.process(frame);
        })
    );
}

// 收集结果
std::vector<DetectionResult> results;

```

```

for (auto& f : futures) {
    results.push_back(f.get());
}

```

线程创建:

```

std::atomic<bool> running{true};
std::queue<Task> taskQueue;
std::mutex queueMutex;
std::condition_variable cv;

// 工作线程
std::thread worker([&]() {
    while (running) {
        Task task;
        {
            std::unique_lock<std::mutex> lock(queueMutex);
            cv.wait(lock, [&]() {
                return !taskQueue.empty() || !running;
            });
            if (!running && taskQueue.empty()) break;
            task = std::move(taskQueue.front());
            taskQueue.pop();
        }
        task.execute();
    }
});

```

1.5.22.10. 高级用法

立即调用的 Lambda (IIFE) 用于复杂的初始化:

```

// 复杂的 const 变量初始化
const std::vector<int> primes = []() {
    std::vector<int> result;
    // 计算质数...
    for (int n = 2; n < 100; n++) {
        bool isPrime = true;
        for (int i = 2; i * i <= n; i++) {
            if (n % i == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) result.push_back(n);
    }
    return result;
}(); // 注意末尾的 () —— 立即调用

// 条件初始化
const auto config = [&]() {
    if (useDefault) {
        return Config::getDefault();
    } else {
        return Config::loadFromFile(filename);
    }
};

```

```
    }
};
```

递归 Lambda 需要显式类型声明：

```
// 使用 std::function 实现递归
std::function<int(int)> factorial = [&factorial](int n) -> int {
    return n <= 1 ? 1 : n * factorial(n - 1);
};

std::cout << factorial(5) << std::endl; // 120

// C++14 的另一种方式：泛型 lambda 自引用
auto factorial2 = [] (auto self, int n) -> int {
    return n <= 1 ? 1 : n * self(self, n - 1);
};
std::cout << factorial2(factorial2, 5) << std::endl; // 120
```

Lambda 作为比较器用于容器：

```
// 自定义 set 排序
auto cmp = [] (const Target& a, const Target& b) {
    return a.priority > b.priority;
};

std::set<Target, decltype(cmp)> prioritySet(cmp);

// 自定义 priority_queue
auto pqCmp = [] (const Task& a, const Task& b) {
    return a.deadline > b.deadline; // 最早截止时间优先
};

std::priority_queue<Task, std::vector<Task>, decltype(pqCmp)> taskQueue(pqCmp);
```

1.5.22.11. 最佳实践

保持 Lambda 简短。如果 lambda 体超过几行，考虑提取为命名函数：

```
// 不好：lambda 过长
std::sort(targets.begin(), targets.end(),
    [] (const Target& a, const Target& b) {
        // 20 行复杂的比较逻辑...
    });

// 好：提取为命名函数
bool compareTargets(const Target& a, const Target& b) {
    // 20 行复杂的比较逻辑...
}

std::sort(targets.begin(), targets.end(), compareTargets);
```

注意捕获的生命周期。避免捕获引用后 lambda 存活时间超过被捕获变量：

```
// 危险
std::function<void()> createTask() {
    std::string data = "important";
    return [&data]() { // data 在函数返回后销毁!
        std::cout << data << std::endl;
    };
}
```

```
// 安全
std::function<void()> createTask() {
    std::string data = "important";
    return [data]() { // 按值捕获, 复制数据
        std::cout << data << std::endl;
    };
}
```

明确捕获意图。避免使用 [=] 或 [&] 捕获所有变量，显式列出需要的变量更清晰：

```
// 不推荐: 不清楚捕获了什么
auto lambda = [=](){ /* ... */ };

// 推荐: 明确捕获的变量
auto lambda = [config, &logger](){ /* ... */ };
```

在类成员函数中，优先捕获具体成员而非 this（如果可能）：

```
class Processor {
public:
    void process() {
        // 不太好: 捕获整个 this
        auto task = [this](){
            return data_ * factor_;
        };

        // 更好: 只捕获需要的成员
        auto task2 = [data_ = data_, factor_ = factor_](){
            return data_ * factor_;
        };
    }

private:
    int data_;
    int factor_;
};
```

Lambda 表达式是现代 C++ 中不可或缺的工具。它使代码更加简洁、更具表现力，特别适合回调、算法定制和异步编程场景。在 RoboMaster 开发中，无论是 ROS 的消息回调、STL 算法的使用，还是多线程编程，lambda 都会是你最常用的语法特性之一。熟练掌握 lambda 的各种用法，将显著提升你的 C++ 编程效率。

1.5.23. 多线程基础

在 RoboMaster 机器人系统中，同时发生着许多事情：相机以 60 帧每秒的速度捕获图像，IMU 以更高的频率更新姿态数据，控制算法需要实时计算电机输出，通信模块需要收发裁判系统和队友的信息。如果所有这些任务都在单一执行流中顺序进行，系统将无法满足实时性要求——当图像处理占用 CPU 时，控制循环就会停滞。

多线程编程允许程序同时执行多个任务。每个线程是一个独立的执行流，拥有自己的程序计数器和栈，但共享进程的内存空间。C++11 标准库提供了跨平台的多线程支持，包括线程创建、同步原语和原子操作。掌握多线程编程，是开发高性能机器人系统的关键技能。

1.5.23.1. 创建线程

`std::thread` 是 C++ 标准库提供的线程类，定义在 `<thread>` 头文件中。创建线程只需将可调用对象（函数、lambda、函数对象）传递给 `std::thread` 的构造函数：

```
#include <thread>
#include <iostream>

void printMessage(const std::string& msg) {
    std::cout << "线程消息: " << msg << std::endl;
}

int main() {
    // 创建线程，执行 printMessage 函数
    std::thread t(printMessage, "Hello from thread!");

    std::cout << "主线程继续执行" << std::endl;

    // 等待线程完成
    t.join();

    std::cout << "线程已结束" << std::endl;
    return 0;
}
```

`std::thread` 构造函数的第一个参数是要执行的函数，后续参数会被传递给该函数。线程创建后立即开始执行，与主线程并发运行。

每个 `std::thread` 对象在销毁前必须被 `join()` 或 `detach()`：

`join()` 阻塞当前线程，等待目标线程执行完成。这是最常用的方式，确保线程的工作完成后继续。

`detach()` 将线程分离，让它在后台独立运行。分离后的线程无法再被 `join`，它会在执行完后自动清理资源。

```
std::thread t1(someFunction);
t1.join();      // 等待 t1 完成

std::thread t2(backgroundTask);
t2.detach();   // 让 t2 在后台运行，不再管理它
```

如果 `std::thread` 对象在析构时既没有 `join` 也没有 `detach`，程序会调用 `std::terminate()` 终止。这是为了防止“悬空线程”——线程仍在运行但已无法访问。

使用 lambda 表达式创建线程更加灵活：

```
int main() {
    int result = 0;

    std::thread t([&result]() {
        // 执行一些计算
        result = 42;
        std::cout << "计算完成" << std::endl;
    });

    t.join();
    std::cout << "结果: " << result << std::endl; // 42
```

```
    return 0;
}
```

可以创建多个线程并发执行：

```
void worker(int id) {
    std::cout << "Worker " << id << " 开始工作" << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::cout << "Worker " << id << " 完成工作" << std::endl;
}

int main() {
    std::vector<std::thread> threads;

    // 创建 4 个工作线程
    for (int i = 0; i < 4; i++) {
        threads.emplace_back(worker, i);
    }

    // 等待所有线程完成
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "所有工作完成" << std::endl;
    return 0;
}
```

`std::this_thread` 命名空间提供了一些作用于当前线程的函数：

```
// 获取当前线程的 ID
std::thread::id myId = std::this_thread::get_id();

// 让当前线程休眠
std::this_thread::sleep_for(std::chrono::milliseconds(100));
std::this_thread::sleep_until(someTimePoint);

// 提示调度器可以切换到其他线程
std::this_thread::yield();
```

1.5.23.2. 数据竞争与互斥量

当多个线程同时访问共享数据，且至少一个线程在修改数据时，就会发生数据竞争（data race）。数据竞争导致未定义行为，程序可能产生错误结果、崩溃或表现出其他不可预测的行为。

```
int counter = 0;

void increment() {
    for (int i = 0; i < 100000; i++) {
        counter++; // 数据竞争!
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
```

```

        t1.join();
        t2.join();

        // 预期 200000, 但实际结果不确定
        std::cout << "Counter: " << counter << std::endl;

        return 0;
    }
}

```

`counter++` 看似是原子操作，实际上它包含三步：读取 `counter` 的值，加 1，写回 `counter`。两个线程可能同时读取相同的值，各自加 1 后写回，导致一次递增被“丢失”。

互斥量（mutex）是解决数据竞争的基本工具。它提供了互斥访问——同一时刻只有一个线程可以“持有”互斥量，其他线程必须等待。

```

#include <mutex>

int counter = 0;
std::mutex mtx; // 互斥量

void increment() {
    for (int i = 0; i < 100000; i++) {
        mtx.lock(); // 获取锁
        counter++; // 临界区：受保护的代码
        mtx.unlock(); // 释放锁
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Counter: " << counter << std::endl; // 200000

    return 0;
}

```

现在结果总是正确的 200000。但手动调用 `lock()` 和 `unlock()` 有风险：如果在临界区内抛出异常或提前返回，`unlock()` 可能不会执行，导致死锁。

1.5.23.3. lock_guard 与 unique_lock

`std::lock_guard` 是一个 RAII 风格的锁管理器。它在构造时自动获取锁，在析构时自动释放锁，确保锁总是被正确释放：

```

void increment() {
    for (int i = 0; i < 100000; i++) {
        std::lock_guard<std::mutex> lock(mtx); // 构造时加锁
        counter++;
    } // 析构时自动解锁，即使发生异常
}

```

C++17 引入了类模板参数推导，可以省略模板参数：

```
std::lock_guard lock(mtx); // C++17
```

`std::unique_lock` 提供了更灵活的锁管理。它支持延迟加锁、尝试加锁、定时加锁，以及在生命周期内多次加锁解锁：

```
std::mutex mtx;

void flexibleLocking() {
    // 延迟加锁
    std::unique_lock<std::mutex> lock(mtx, std::defer_lock);
    // 此时还没有加锁

    // 手动加锁
    lock.lock();
    // 临界区...
    lock.unlock();

    // 尝试加锁（非阻塞）
    if (lock.try_lock()) {
        // 成功获取锁
        // ...
    }

    // 定时尝试加锁
    if (lock.try_lock_for(std::chrono::milliseconds(100))) {
        // 在 100ms 内成功获取锁
        // ...
    }
} // 自动解锁（如果持有锁的话）
```

`unique_lock` 可以被移动（但不能复制），这在需要传递锁的场景下很有用。它也是条件变量所需的锁类型。

当需要同时锁定多个互斥量时，使用 `std::lock` 或 `std::scoped_lock` (C++17) 来避免死锁：

```
std::mutex mtx1, mtx2;

void safeMultiLock() {
    // C++17: scoped_lock 同时锁定多个互斥量，避免死锁
    std::scoped_lock lock(mtx1, mtx2);
    // 两个互斥量都被锁定
    // ...
} // 自动解锁

// C++11/14 方式
void safeMultiLockOld() {
    std::unique_lock<std::mutex> lock1(mtx1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(mtx2, std::defer_lock);
    std::lock(lock1, lock2); // 原子地锁定两个互斥量
    // ...
}
```

1.5.23.4. 条件变量

互斥量保护数据免受并发访问，但有时线程需要等待某个条件成立后才能继续。例如，消费者线程需要等待队列中有数据，生产者线程需要等待队列有空间。

条件变量 (`std::condition_variable`) 允许线程等待某个条件，并在条件可能变化时被其他线程唤醒。它必须与互斥量和 `unique_lock` 配合使用：

```
#include <condition_variable>
#include <queue>

std::queue<int> dataQueue;
std::mutex mtx;
std::condition_variable cv;

void producer() {
    for (int i = 0; i < 10; i++) {
        {
            std::lock_guard<std::mutex> lock(mtx);
            dataQueue.push(i);
            std::cout << "生产: " << i << std::endl;
        }
        cv.notify_one(); // 通知一个等待的消费者
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);

        // 等待直到队列非空
        cv.wait(lock, []() { return !dataQueue.empty(); });

        int data = dataQueue.front();
        dataQueue.pop();
        lock.unlock(); // 尽早释放锁

        std::cout << "消费: " << data << std::endl;

        if (data == 9) break; // 结束条件
    }
}

int main() {
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

`cv.wait(lock, predicate)` 的工作方式：

1. 检查谓词 (`predicate`)，如果为真则立即返回
2. 如果谓词为假，释放锁并阻塞当前线程
3. 当被 `notify_one()` 或 `notify_all()` 唤醒后，重新获取锁
4. 再次检查谓词，如果仍为假则继续等待

5. 谓词为真时返回，此时持有锁

使用谓词很重要，因为条件变量可能会“虚假唤醒”（spurious wakeup）——在没有通知的情况下被唤醒。谓词确保只有在条件真正满足时才继续执行。

`notify_one()` 唤醒一个等待的线程，`notify_all()` 唤醒所有等待的线程。选择哪个取决于场景：如果只有一个线程能处理条件变化，用 `notify_one()`；如果多个线程都需要响应，用 `notify_all()`。

条件变量还支持定时等待：

```
std::unique_lock<std::mutex> lock(mtx);

// 等待最多 100ms
if (cv.wait_for(lock, std::chrono::milliseconds(100), predicate)) {
    // 条件在时限内满足
} else {
    // 超时
}

// 等待直到特定时间点
if (cv.wait_until(lock, deadline, predicate)) {
    // 条件在截止时间前满足
} else {
    // 超时
}
```

1.5.23.5. 原子操作

对于简单的数值操作，使用互斥量可能过于重量级。`std::atomic` 提供了无锁的原子操作，性能更高：

```
#include <atomic>

std::atomic<int> counter(0);

void increment() {
    for (int i = 0; i < 100000; i++) {
        counter++; // 原子操作，无需互斥量
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Counter: " << counter << std::endl; // 200000
}

return 0;
}
```

`std::atomic` 支持多种操作：

```
std::atomic<int> value(0);
```

```

value.store(10);           // 原子存储
int v = value.load();     // 原子加载
int old = value.exchange(20); // 原子交换, 返回旧值

// 原子加减
value.fetch_add(5);       // 加 5, 返回旧值
value.fetch_sub(3);        // 减 3, 返回旧值
value += 10;               // 等价于 fetch_add(10)

// 比较并交换 (CAS)
int expected = 20;
bool success = value.compare_exchange_strong(expected, 30);
// 如果 value == expected, 则 value = 30, 返回 true
// 否则 expected = value, 返回 false

std::atomic<bool> 常用作标志:
std::atomic<bool> running(true);

void workerThread() {
    while (running) { // 原子读取
        // 执行工作...
    }
}

void stopWorker() {
    running = false; // 原子写入
}

```

1.5.23.6. 线程安全的数据结构

结合互斥量和条件变量, 可以构建线程安全的数据结构。以下是一个常用的线程安全队列:

```

template <typename T>
class ThreadSafeQueue {
public:
    void push(T value) {
        std::lock_guard<std::mutex> lock(mtx);
        queue.push(std::move(value));
        cv.notify_one();
    }

    // 阻塞式取出
    T pop() {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [this]() { return !queue.empty(); });
        T value = std::move(queue.front());
        queue.pop();
        return value;
    }

    // 非阻塞式尝试取出
    bool tryPop(T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        if (queue.empty()) return false;
        value = std::move(queue.front());
        queue.pop();
    }
}

```

```

        return true;
    }

    // 定时等待取出
    bool waitAndPop(T& value, std::chrono::milliseconds timeout) {
        std::unique_lock<std::mutex> lock(mtx);
        if (!cv.wait_for(lock, timeout, [this]() { return !queue.empty(); })) {
            return false;
        }
        value = std::move(queue.front());
        queue.pop();
        return true;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(mtx);
        return queue.empty();
    }

    size_t size() const {
        std::lock_guard<std::mutex> lock(mtx);
        return queue.size();
    }

private:
    mutable std::mutex mtx;
    std::queue<T> queue;
    std::condition_variable cv;
};
```

1.5.23.7. RoboMaster 多线程应用

在 RoboMaster 机器人系统中，多线程是必不可少的。让我们看几个典型的应用场景。

图像采集与处理分离：

```

class VisionSystem {
public:
    VisionSystem() : running(false) {}

    void start() {
        running = true;
        captureThread = std::thread(&VisionSystem::captureLoop, this);
        processThread = std::thread(&VisionSystem::processLoop, this);
    }

    void stop() {
        running = false;
        frameQueue.push(Frame{}); // 发送空帧唤醒处理线程

        if (captureThread.joinable()) captureThread.join();
        if (processThread.joinable()) processThread.join();
    }

    std::vector<Target> getLatestTargets() {
        std::lock_guard<std::mutex> lock(resultMutex);
        return latestTargets;
    }
```

```

    }

private:
    void captureLoop() {
        while (running) {
            Frame frame = camera.capture();
            frameQueue.push(std::move(frame));
        }
    }

    void processLoop() {
        while (running) {
            Frame frame = frameQueue.pop();
            if (!frame.isValid()) break; // 收到停止信号

            std::vector<Target> targets = detector.detect(frame);

            {
                std::lock_guard<std::mutex> lock(resultMutex);
                latestTargets = std::move(targets);
            }
        }
    }
}

Camera camera;
Detector detector;
ThreadSafeQueue<Frame> frameQueue;

std::vector<Target> latestTargets;
std::mutex resultMutex;

std::atomic<bool> running;
std::thread captureThread;
std::thread processThread;
};

```

控制循环与通信分离：

```

class RobotController {
public:
    void start() {
        running = true;
        controlThread = std::thread(&RobotController::controlLoop, this);
        commThread = std::thread(&RobotController::commLoop, this);
    }

    void stop() {
        running = false;
        if (controlThread.joinable()) controlThread.join();
        if (commThread.joinable()) commThread.join();
    }

    void setTargetVelocity(double vx, double vy, double omega) {
        std::lock_guard<std::mutex> lock(cmdMutex);
        targetVx = vx;
        targetVy = vy;
        targetOmega = omega;
    }
}

```

```

    }

private:
    void controlLoop() {
        auto lastTime = std::chrono::steady_clock::now();

        while (running) {
            auto now = std::chrono::steady_clock::now();
            double dt = std::chrono::duration<double>(now - lastTime).count();
            lastTime = now;

            // 读取目标速度
            double vx, vy, omega;
            {
                std::lock_guard<std::mutex> lock(cmdMutex);
                vx = targetVx;
                vy = targetVy;
                omega = targetOmega;
            }

            // 读取传感器
            IMUData imu = imuSensor.read();

            // 计算控制输出
            auto motorOutputs = controller.compute(vx, vy, omega, imu, dt);

            // 发送到电机
            for (size_t i = 0; i < 4; i++) {
                motors[i].setOutput(motorOutputs[i]);
            }

            // 控制频率 1000Hz
            std::this_thread::sleep_until(now + std::chrono::milliseconds(1));
        }
    }

    void commLoop() {
        while (running) {
            // 接收裁判系统数据
            if (referee.hasData()) {
                RefereeData data = referee.receive();
                processRefereeData(data);
            }

            // 发送状态
            sendStatus();

            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }

    std::atomic<bool> running;
    std::thread controlThread;
    std::thread commThread;
}

```

```

    std::mutex cmdMutex;
    double targetVx = 0, targetVy = 0, targetOmega = 0;

    IMUSensor imuSensor;
    Motor motors[4];
    ChassisController controller;
    RefereeSystem referee;
};

}

```

传感器数据融合：

```

class SensorFusion {
public:
    void start() {
        running = true;
        imuThread = std::thread(&SensorFusion::imuLoop, this);
        lidarThread = std::thread(&SensorFusion::lidarLoop, this);
        fusionThread = std::thread(&SensorFusion::fusionLoop, this);
    }

    Pose getEstimatedPose() {
        std::lock_guard<std::mutex> lock(poseMutex);
        return estimatedPose;
    }

private:
    void imuLoop() {
        while (running) {
            IMUData data = imu.read();
            {
                std::lock_guard<std::mutex> lock(imuMutex);
                latestIMU = data;
                imuUpdated = true;
            }
            imuCv.notify_one();
            std::this_thread::sleep_for(std::chrono::microseconds(500)); // 2000Hz
        }
    }

    void lidarLoop() {
        while (running) {
            LidarScan scan = lidar.scan();
            {
                std::lock_guard<std::mutex> lock(lidarMutex);
                latestScan = std::move(scan);
                lidarUpdated = true;
            }
            lidarCv.notify_one();
            // Lidar 自身频率, 通常 10-40Hz
        }
    }

    void fusionLoop() {
        while (running) {
            // 等待 IMU 数据 (高频)
        }
    }
}

```

```

        std::unique_lock<std::mutex> lock(imuMutex);
        imuCv.wait(lock, [this](){ { return imuUpdated || !running; }});
        if (!running) break;

        // 使用 IMU 进行预测
        ekf.predict(latestIMU);
        imuUpdated = false;
    }

    // 检查是否有 Lidar 数据 (低频)
    {
        std::lock_guard<std::mutex> lock(lidarMutex);
        if (lidarUpdated) {
            ekf.updateWithLidar(latestScan);
            lidarUpdated = false;
        }
    }

    // 更新估计位姿
    {
        std::lock_guard<std::mutex> lock(poseMutex);
        estimatedPose = ekf.getState();
    }
}

std::atomic<bool> running;
std::thread imuThread, lidarThread, fusionThread;

IMUSensor imu;
LidarSensor lidar;
ExtendedKalmanFilter ekf;

std::mutex imuMutex, lidarMutex, poseMutex;
std::condition_variable imuCv, lidarCv;

IMUData latestIMU;
LidarScan latestScan;
Pose estimatedPose;

bool imuUpdated = false;
bool lidarUpdated = false;
};


```

1.5.23.8. 常见陷阱与最佳实践

多线程编程容易出错，以下是一些常见陷阱和避免方法。

死锁：两个线程相互等待对方持有的锁。避免方法包括：始终以相同顺序获取多个锁，使用 `std::scoped_lock` 同时获取，避免在持有锁时调用未知代码。

```

// 死锁示例
void thread1() {
    std::lock_guard<std::mutex> lock1(mtx1);
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::lock_guard<std::mutex> lock2(mtx2); // 等待 mtx2
}

```

```

}

void thread2() {
    std::lock_guard<std::mutex> lock2(mtx2);
    std::this_thread::sleep_for(std::chrono::milliseconds(1));
    std::lock_guard<std::mutex> lock1(mtx1); // 等待 mtx1, 死锁!
}

// 解决方案
void thread1Fixed() {
    std::scoped_lock lock(mtx1, mtx2); // 同时获取, 避免死锁
}

```

忘记保护共享数据：每次访问共享数据都必须加锁，不仅是写入，读取也需要。

```

// 错误: 读取时没有加锁
void reader() {
    int value = sharedData; // 数据竞争!
}

// 正确
void reader() {
    std::lock_guard<std::mutex> lock(mtx);
    int value = sharedData;
}

```

锁的粒度：锁的范围应该尽可能小。持有锁的时间越长，其他线程等待的时间就越长。

```

// 不好: 锁的范围太大
void process() {
    std::lock_guard<std::mutex> lock(mtx);
    expensiveOperation(); // 不访问共享数据也持有锁
    sharedData = result;
}

// 好: 只在必要时持有锁
void process() {
    auto result = expensiveOperation(); // 不持有锁
    std::lock_guard<std::mutex> lock(mtx);
    sharedData = result; // 只在访问共享数据时持有锁
}

```

线程生命周期管理：确保线程在使用的资源之前不会结束，也不会在资源销毁后仍在运行。

```

// 危险: 局部变量可能在线程结束前被销毁
void dangerous() {
    int localData = 42;
    std::thread t([&localData]() {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << localData << std::endl; // localData 可能已销毁!
    });
    t.detach(); // 分离后无法控制线程生命周期
}

// 安全: 使用值捕获或确保数据生命周期
void safe() {
    int localData = 42;
    std::thread t([localData]{} { // 值捕获
        std::this_thread::sleep_for(std::chrono::seconds(1));
    });
}

```

```

        std::cout << localData << std::endl;
    });
    t.join(); // 或者等待线程完成
}

```

使用 RAII 管理线程：

```

class JoinThreads {
public:
    explicit JoinThreads(std::vector<std::thread>& threads)
        : threads(threads) {}

    ~JoinThreads() {
        for (auto& t : threads) {
            if (t.joinable()) t.join();
        }
    }

private:
    std::vector<std::thread>& threads;
};

void process() {
    std::vector<std::thread> threads;
    JoinThreads joiner(threads); // 确保所有线程被 join

    for (int i = 0; i < 4; i++) {
        threads.emplace_back(worker, i);
    }

    // 即使这里抛出异常，joiner 的析构函数也会 join 所有线程
}

```

多线程编程是复杂但强大的技术。在 RoboMaster 开发中，合理使用多线程可以充分利用多核处理器的能力，实现传感器采集、图像处理、控制算法、通信等任务的并行执行，从而满足机器人系统的实时性要求。但同时也要谨慎处理共享数据，避免数据竞争和死锁等问题。随着经验的积累，你会逐渐掌握编写正确、高效的多线程代码的技巧。

1.5.24. 文件操作

机器人系统需要与外部世界交换数据：读取配置文件来调整参数，记录日志以便调试和分析，保存标定数据和运行状态。C++ 标准库提供了 `fstream` 类族来处理文件输入输出，它们使用与 `cin/cout` 相同的流操作符，学习成本很低。掌握文件操作，可以让你的机器人程序更加灵活和可维护——参数可以在不重新编译的情况下修改，问题可以通过日志追溯分析。

1.5.24.1. `fstream` 基础

C++ 文件流定义在 `<fstream>` 头文件中，包含三个主要类：

- `std::ifstream`: 输入文件流，用于读取文件
- `std::ofstream`: 输出文件流，用于写入文件
- `std::fstream`: 双向文件流，可读可写

基本的文件写入：

```

#include <fstream>
#include <iostream>

int main() {
    // 创建并打开文件（如果文件已存在则清空）
    std::ofstream outFile("output.txt");

    if (!outFile) {
        std::cerr << "无法打开文件" << std::endl;
        return 1;
    }

    // 使用 << 写入，和 cout 用法相同
    outFile << "Hello, File!" << std::endl;
    outFile << "数值: " << 42 << std::endl;
    outFile << "浮点数: " << 3.14159 << std::endl;

    // 文件在 outFile 离开作用域时自动关闭
    return 0;
}

```

基本的文件读取：

```

#include <fstream>
#include <iostream>
#include <string>

int main() {
    std::ifstream inFile("output.txt");

    if (!inFile) {
        std::cerr << "无法打开文件" << std::endl;
        return 1;
    }

    // 逐行读取
    std::string line;
    while (std::getline(inFile, line)) {
        std::cout << "读取: " << line << std::endl;
    }

    return 0;
}

```

文件打开模式可以通过第二个参数指定：

```

// 常用打开模式
std::ios::in      // 读取 (ifstream 默认)
std::ios::out     // 写入 (ofstream 默认)
std::ios::app      // 追加模式，写入到文件末尾
std::ios::trunc   // 截断模式，清空现有内容 (out 的默认行为)
std::ios::binary  // 二进制模式
std::ios::ate      // 打开后定位到文件末尾

// 组合使用
std::ofstream logFile("log.txt", std::ios::app); // 追加写入
std::fstream dataFile("data.bin", std::ios::in | std::ios::out |
std::ios::binary);

```

检查文件状态:

```
std::ifstream file("data.txt");

// 检查是否成功打开
if (!file.is_open()) {
    std::cerr << "文件打开失败" << std::endl;
}

// 流状态检查
if (file.good()) // 一切正常
if (file.eof()) // 到达文件末尾
if (file.fail()) // 操作失败 (格式错误等)
if (file.bad()) // 严重错误 (如硬件故障)

// 清除错误状态
file.clear();

// 显式关闭 (通常不需要, 析构时自动关闭)
file.close();
```

1.5.24.2. 文本文件读写

读取格式化数据:

```
// 假设文件内容:
// Alice 25 3.8
// Bob 22 3.5
// Charlie 24 3.9

struct Student {
    std::string name;
    int age;
    double gpa;
};

std::vector<Student> readStudents(const std::string& filename) {
    std::vector<Student> students;
    std::ifstream file(filename);

    if (!file) {
        throw std::runtime_error("无法打开文件: " + filename);
    }

    Student s;
    while (file >> s.name >> s.age >> s.gpa) {
        students.push_back(s);
    }

    return students;
}
```

处理包含空格的字符串需要使用 `getline`:

```
// 文件内容 (逗号分隔):
// Alice Smith,25,3.8
// Bob Johnson,22,3.5
```

```

std::vector<Student> readCSV(const std::string& filename) {
    std::vector<Student> students;
    std::ifstream file(filename);
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream iss(line);
        Student s;

        std::getline(iss, s.name, ','); // 读到逗号为止
        iss >> s.age;
        iss.ignore(1); // 跳过逗号
        iss >> s.gpa;

        students.push_back(s);
    }

    return students;
}

```

写入格式化数据:

```

#include <iomanip>

void writeStudents(const std::string& filename,
                   const std::vector<Student>& students) {
    std::ofstream file(filename);

    if (!file) {
        throw std::runtime_error("无法创建文件: " + filename);
    }

    // 设置格式
    file << std::fixed << std::setprecision(2);

    // 写入表头
    file << std::left << std::setw(20) << "Name"
        << std::setw(10) << "Age"
        << std::setw(10) << "GPA" << std::endl;

    file << std::string(40, '-') << std::endl;

    // 写入数据
    for (const auto& s : students) {
        file << std::left << std::setw(20) << s.name
            << std::setw(10) << s.age
            << std::setw(10) << s.gpa << std::endl;
    }
}

```

1.5.24.3. 二进制文件读写

二进制模式直接读写内存中的字节，效率更高，但文件不可人工阅读：

```

struct SensorData {
    double timestamp;
    float acceleration[3];
    float gyroscope[3];

```

```

        uint32_t status;
    };

    // 写入二进制数据
    void saveSensorData(const std::string& filename,
                         const std::vector<SensorData>& data) {
        std::ofstream file(filename, std::ios::binary);

        if (!file) {
            throw std::runtime_error("无法创建文件");
        }

        // 写入数据数量
        size_t count = data.size();
        file.write(reinterpret_cast<const char*>(&count), sizeof(count));

        // 写入数据
        file.write(reinterpret_cast<const char*>(data.data()),
                   data.size() * sizeof(SensorData));
    }

    // 读取二进制数据
    std::vector<SensorData> loadSensorData(const std::string& filename) {
        std::ifstream file(filename, std::ios::binary);

        if (!file) {
            throw std::runtime_error("无法打开文件");
        }

        // 读取数量
        size_t count;
        file.read(reinterpret_cast<char*>(&count), sizeof(count));

        // 读取数据
        std::vector<SensorData> data(count);
        file.read(reinterpret_cast<char*>(data.data()),
                  count * sizeof(SensorData));

        return data;
    }
}

```

文件定位操作:

```

std::fstream file("data.bin", std::ios::in | std::ios::out | std::ios::binary);

// 获取当前位置
std::streampos pos = file.tellg(); // 读取位置
std::streampos wpos = file.tellp(); // 写入位置

// 定位
file.seekg(0); // 移动到开头
file.seekg(100); // 移动到第 100 字节
file.seekg(-10, std::ios::end); // 从末尾倒数第 10 字节
file.seekg(50, std::ios::cur); // 从当前位置前进 50 字节

// 读取特定记录

```

```

size_t recordIndex = 5;
file.seekg(sizeof(size_t) + recordIndex * sizeof(SensorData));
SensorData record;
file.read(reinterpret_cast<char*>(&record), sizeof(record));

```

1.5.24.4. 配置文件读写

在 RoboMaster 开发中，配置文件用于存储可调参数，方便在不重新编译的情况下修改机器人行为。

简单的键值对配置文件：

```

// config.txt 内容:
// # PID 参数
// kp = 1.5
// ki = 0.1
// kd = 0.05
//
// # 速度限制
// max_speed = 5000
// min_speed = -5000

class ConfigReader {
public:
    bool load(const std::string& filename) {
        std::ifstream file(filename);
        if (!file) return false;

        std::string line;
        while (std::getline(file, line)) {
            // 跳过空行和注释
            if (line.empty() || line[0] == '#') continue;

            // 解析键值对
            size_t eqPos = line.find('=');
            if (eqPos == std::string::npos) continue;

            std::string key = trim(line.substr(0, eqPos));
            std::string value = trim(line.substr(eqPos + 1));

            config_[key] = value;
        }

        return true;
    }

    std::string getString(const std::string& key,
                         const std::string& defaultValue = "") const {
        auto it = config_.find(key);
        return (it != config_.end()) ? it->second : defaultValue;
    }

    int getInt(const std::string& key, int defaultValue = 0) const {
        auto it = config_.find(key);
        if (it == config_.end()) return defaultValue;
        try {

```

```

        return std::stoi(it->second);
    } catch (...) {
        return defaultValue;
    }
}

double getDouble(const std::string& key, double defaultValue = 0.0) const {
    auto it = config_.find(key);
    if (it == config_.end()) return defaultValue;
    try {
        return std::stod(it->second);
    } catch (...) {
        return defaultValue;
    }
}

bool getBool(const std::string& key, bool defaultValue = false) const {
    auto it = config_.find(key);
    if (it == config_.end()) return defaultValue;
    std::string val = it->second;
    std::transform(val.begin(), val.end(), val.begin(), ::tolower);
    return (val == "true" || val == "1" || val == "yes");
}

private:
    static std::string trim(const std::string& s) {
        size_t start = s.find_first_not_of(" \t\r\n");
        if (start == std::string::npos) return "";
        size_t end = s.find_last_not_of(" \t\r\n");
        return s.substr(start, end - start + 1);
    }

    std::map<std::string, std::string> config_;
};

// 使用
int main() {
    ConfigReader config;
    if (!config.load("robot_config.txt")) {
        std::cerr << "配置文件加载失败" << std::endl;
        return 1;
    }

    double kp = config.getDouble("kp", 1.0);
    double ki = config.getDouble("ki", 0.0);
    double kd = config.getDouble("kd", 0.0);
    int maxSpeed = config.getInt("max_speed", 3000);

    std::cout << "PID: " << kp << ", " << ki << ", " << kd << std::endl;
    std::cout << "Max Speed: " << maxSpeed << std::endl;

    return 0;
}

```

保存配置文件：

```

class ConfigWriter {
public:
    void set(const std::string& key, const std::string& value) {
        config_[key] = value;
    }

    void set(const std::string& key, int value) {
        config_[key] = std::to_string(value);
    }

    void set(const std::string& key, double value) {
        std::ostringstream oss;
        oss << std::fixed << std::setprecision(6) << value;
        config_[key] = oss.str();
    }

    void set(const std::string& key, bool value) {
        config_[key] = value ? "true" : "false";
    }

    bool save(const std::string& filename) {
        std::ofstream file(filename);
        if (!file) return false;

        file << "# Auto-generated configuration file" << std::endl;
        file << "# Generated at: " << getCurrentTime() << std::endl;
        file << std::endl;

        for (const auto& [key, value] : config_) {
            file << key << " = " << value << std::endl;
        }

        return true;
    }

private:
    std::string getCurrentTime() {
        auto now = std::chrono::system_clock::now();
        auto time = std::chrono::system_clock::to_time_t(now);
        std::string str = std::ctime(&time);
        str.pop_back(); // 移除换行符
        return str;
    }

    std::map<std::string, std::string> config_;
};


```

1.5.24.5. INI 格式配置文件

INI 格式支持分节 (section)，更适合复杂的配置：

```

; robot_config.ini
[PID_Speed]
kp = 1.5
ki = 0.1
kd = 0.05

```

```

[PID_Angle]
kp = 2.0
ki = 0.0
kd = 0.2

[Limits]
max_speed = 5000
max_current = 10000

[Features]
auto_aim = true
debug_mode = false

```

INI 解析器实现：

```

class INIReader {
public:
    bool load(const std::string& filename) {
        std::ifstream file(filename);
        if (!file) return false;

        std::string line, currentSection;

        while (std::getline(file, line)) {
            line = trim(line);

            // 跳过空行和注释
            if (line.empty() || line[0] == ';' || line[0] == '#') continue;

            // 检测节
            if (line[0] == '[' && line.back() == ']') {
                currentSection = line.substr(1, line.size() - 2);
                continue;
            }

            // 解析键值对
            size_t eqPos = line.find('=');
            if (eqPos == std::string::npos) continue;

            std::string key = trim(line.substr(0, eqPos));
            std::string value = trim(line.substr(eqPos + 1));

            // 存储为 "section.key" 格式
            std::string fullKey = currentSection.empty() ?
                key : (currentSection + "." + key);
            data_[fullKey] = value;
        }

        return true;
    }

    double getDouble(const std::string& section,
                    const std::string& key,
                    double defaultValue = 0.0) const {
        std::string fullKey = section + "." + key;
        auto it = data_.find(fullKey);

```

```

        if (it == data_.end()) return defaultValue;
        try {
            return std::stod(it->second);
        } catch (...) {
            return defaultValue;
        }
    }

    int getInt(const std::string& section,
               const std::string& key,
               int defaultValue = 0) const {
        std::string fullKey = section + "." + key;
        auto it = data_.find(fullKey);
        if (it == data_.end()) return defaultValue;
        try {
            return std::stoi(it->second);
        } catch (...) {
            return defaultValue;
        }
    }

    bool getBool(const std::string& section,
                 const std::string& key,
                 bool defaultValue = false) const {
        std::string fullKey = section + "." + key;
        auto it = data_.find(fullKey);
        if (it == data_.end()) return defaultValue;
        std::string val = it->second;
        std::transform(val.begin(), val.end(), val.begin(), ::tolower);
        return (val == "true" || val == "1" || val == "yes");
    }

private:
    static std::string trim(const std::string& s) {
        size_t start = s.find_first_not_of(" \t\r\n");
        if (start == std::string::npos) return "";
        size_t end = s.find_last_not_of(" \t\r\n");
        return s.substr(start, end - start + 1);
    }

    std::map<std::string, std::string> data_;
};

// 使用
int main() {
    INIReader config;
    if (!config.load("robot_config.ini")) {
        std::cerr << "配置加载失败" << std::endl;
        return 1;
    }

    // 读取速度环 PID
    double speedKp = config.getDouble("PID_Speed", "kp", 1.0);
    double speedKi = config.getDouble("PID_Speed", "ki", 0.0);
    double speedKd = config.getDouble("PID_Speed", "kd", 0.0);
}

```

```

    // 读取角度环 PID
    double angleKp = config.getDouble("PID_Angle", "kp", 1.0);

    // 读取限制
    int maxSpeed = config.getInt("Limits", "max_speed", 3000);

    // 读取功能开关
    bool autoAim = config.getBool("Features", "auto_aim", false);

    return 0;
}

```

1.5.24.6. 日志记录

良好的日志系统对于调试和问题追踪至关重要。以下是一个简单但实用的日志类：

```

#include <fstream>
#include <iostream>
#include <chrono>
#include <iomanip>
#include <mutex>
#include <sstream>

enum class LogLevel {
    DEBUG,
    INFO,
    WARNING,
    ERROR,
    FATAL
};

class Logger {
public:
    static Logger& getInstance() {
        static Logger instance;
        return instance;
    }

    void setLogFile(const std::string& filename) {
        std::lock_guard<std::mutex> lock(mutex_);
        if (file_.is_open()) {
            file_.close();
        }
        file_.open(filename, std::ios::app);
        logToFile_ = file_.is_open();
    }

    void setLogLevel(LogLevel level) {
        minLevel_ = level;
    }

    void setConsoleOutput(bool enabled) {
        logToConsole_ = enabled;
    }
}

```

```

void log(LogLevel level, const std::string& message,
         const char* file = "", int line = 0) {
    if (level < minLevel_) return;

    std::lock_guard<std::mutex> lock(mutex_);

    std::string logLine = formatMessage(level, message, file, line);

    if (logToConsole_) {
        std::ostream& out = (level >= LogLevel::ERROR) ? std::cerr :
        std::cout;
        out << getColorCode(level) << logLine << "\033[0m" << std::endl;
    }

    if (logToFile_ && file_.is_open()) {
        file_ << logLine << std::endl;
        file_.flush(); // 确保立即写入
    }
}

// 便捷方法
void debug(const std::string& msg, const char* file = "", int line = 0) {
    log(LogLevel::DEBUG, msg, file, line);
}

void info(const std::string& msg, const char* file = "", int line = 0) {
    log(LogLevel::INFO, msg, file, line);
}

void warning(const std::string& msg, const char* file = "", int line = 0) {
    log(LogLevel::WARNING, msg, file, line);
}

void error(const std::string& msg, const char* file = "", int line = 0) {
    log(LogLevel::ERROR, msg, file, line);
}

void fatal(const std::string& msg, const char* file = "", int line = 0) {
    log(LogLevel::FATAL, msg, file, line);
}

private:
    Logger() = default;
    ~Logger() {
        if (file_.is_open()) {
            file_.close();
        }
    }

    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;

    std::string formatMessage(LogLevel level, const std::string& message,
                             const char* file, int line) {
        std::ostringstream oss;

```

```

// 时间戳
auto now = std::chrono::system_clock::now();
auto time = std::chrono::system_clock::to_time_t(now);
auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(
    now.time_since_epoch()) % 1000;

oss << std::put_time(std::localtime(&time), "%Y-%m-%d %H:%M:%S");
oss << '.' << std::setfill('0') << std::setw(3) << ms.count();

// 日志级别
oss << " [" << getLevelString(level) << "]";

// 文件和行号（如果提供）
if (file && file[0] != '\0') {
    oss << " [" << file << ":" << line << "]";
}

// 消息
oss << " " << message;

return oss.str();
}

const char* getLevelString(LogLevel level) {
    switch (level) {
        case LogLevel::DEBUG:   return "DEBUG";
        case LogLevel::INFO:   return "INFO ";
        case LogLevel::WARNING: return "WARN ";
        case LogLevel::ERROR:  return "ERROR";
        case LogLevel::FATAL:  return "FATAL";
        default:               return "?????";
    }
}

const char* getColorCode(LogLevel level) {
    switch (level) {
        case LogLevel::DEBUG:   return "\033[36m"; // 青色
        case LogLevel::INFO:   return "\033[32m"; // 绿色
        case LogLevel::WARNING: return "\033[33m"; // 黄色
        case LogLevel::ERROR:  return "\033[31m"; // 红色
        case LogLevel::FATAL:  return "\033[35m"; // 紫色
        default:               return "\033[0m";
    }
}

std::ofstream file_;
std::mutex mutex_;
LogLevel minLevel_ = LogLevel::DEBUG;
bool logToFile_ = false;
bool logToConsole_ = true;
};

// 便捷宏，自动添加文件名和行号
#define LOG_DEBUG(msg) Logger::getInstance().debug(msg, __FILE__, __LINE__)

```

```

#define LOG_INFO(msg)    Logger::getInstance().info(msg, __FILE__, __LINE__)
#define LOG_WARNING(msg) Logger::getInstance().warning(msg, __FILE__, __LINE__)
#define LOG_ERROR(msg)   Logger::getInstance().error(msg, __FILE__, __LINE__)
#define LOG_FATAL(msg)   Logger::getInstance().fatal(msg, __FILE__, __LINE__)

```

使用示例：

```

int main() {
    // 配置日志
    Logger::getInstance().setLogFile("robot.log");
    Logger::getInstance().setLogLevel(LogLevel::DEBUG);
    Logger::getInstance().setConsoleOutput(true);

    LOG_INFO("机器人系统启动");

    try {
        // 加载配置
        LOG_DEBUG("正在加载配置文件...");
        ConfigReader config;
        if (!config.load("config.txt")) {
            LOG_ERROR("配置文件加载失败");
            return 1;
        }
        LOG_INFO("配置加载成功");

        // 初始化硬件
        LOG_DEBUG("初始化电机控制器...");
        // ...

        LOG_INFO("系统初始化完成，进入主循环");

        while (running) {
            // 主循环
            if (someError) {
                LOG_WARNING("检测到异常状态：" + errorDescription);
            }
        }
    } catch (const std::exception& e) {
        LOG_FATAL(std::string("未捕获异常：" ) + e.what());
        return 1;
    }

    LOG_INFO("机器人系统关闭");
    return 0;
}

```

1.5.24.7. 带格式化的日志

支持类似 `printf` 风格的格式化日志：

```

class FormattedLogger : public Logger {
public:
    template<typename... Args>
    void log(LogLevel level, const char* format, Args... args) {
        char buffer[1024];
        snprintf(buffer, sizeof(buffer), format, args...);
    }
}

```

```

        log(level, std::string(buffer));
    }
};

// 或使用 C++20 的 std::format (如果可用)
#ifndef __cpp_lib_format
#include <format>

template<typename... Args>
void logFormatted(LogLevel level, std::format_string<Args...> fmt, Args&&...
args) {
    Logger::getInstance().log(level, std::format(fmt,
std::forward<Args>(args)...));
}
#endif

// 使用字符串流的方式
class LogStream {
public:
    LogStream(LogLevel level, const char* file, int line)
        : level_(level), file_(file), line_(line) {}

    ~LogStream() {
        Logger::getInstance().log(level_, stream_.str(), file_, line_);
    }

    template<typename T>
    LogStream& operator<<(const T& value) {
        stream_ << value;
        return *this;
    }
}

private:
    LogLevel level_;
    const char* file_;
    int line_;
    std::ostringstream stream_;
};

#define LOG(level) LogStream(level, __FILE__, __LINE__)

// 使用
LOG(LogLevel::INFO) << "电机速度: " << motorSpeed << " RPM, 温度: " << temp <<
"°C";

```

1.5.24.8. 日志文件轮转

长时间运行的系统需要日志轮转，避免日志文件过大：

```

class RotatingLogger {
public:
    RotatingLogger(const std::string& baseFilename,
                  size_t maxFileSize = 10 * 1024 * 1024, // 10MB
                  int maxFiles = 5)
        : baseFilename_(baseFilename),
          maxFileSize_(maxFileSize),

```

```

        maxFiles_(maxFiles) {
            openNewFile();
        }

        void log(const std::string& message) {
            std::lock_guard<std::mutex> lock(mutex_);

            if (currentSize_ >= maxFileSize_) {
                rotateFiles();
            }

            if (file_.is_open()) {
                file_ << message << std::endl;
                currentSize_ += message.size() + 1;
            }
        }

    private:
        void openNewFile() {
            if (file_.is_open()) {
                file_.close();
            }
            file_.open(baseFilename_, std::ios::app);

            // 获取当前文件大小
            file_.seekp(0, std::ios::end);
            currentSize_ = file_.tellp();
        }

        void rotateFiles() {
            file_.close();

            // 删除最旧的文件
            std::string oldestFile = baseFilename_ + "." +
            std::to_string(maxFiles_);
            std::remove(oldestFile.c_str());

            // 重命名现有文件
            for (int i = maxFiles_ - 1; i >= 1; i--) {
                std::string oldName = baseFilename_ + "." + std::to_string(i);
                std::string newName = baseFilename_ + "." + std::to_string(i + 1);
                std::rename(oldName.c_str(), newName.c_str());
            }

            // 当前文件变为 .1
            std::rename(baseFilename_.c_str(), (baseFilename_ + ".1").c_str());

            // 打开新文件
            file_.open(baseFilename_);
            currentSize_ = 0;
        }

        std::string baseFilename_;
        size_t maxFileSize_;
        int maxFiles_;

```

```
    std::ofstream file_;
    size_t currentSize_ = 0;
    std::mutex mutex_;
};
```

1.5.24.9. 数据记录与回放

在 RoboMaster 开发中，记录传感器数据用于离线分析和算法调试非常重要：

```
class DataRecorder {
public:
    DataRecorder(const std::string& filename)
        : file_(filename, std::ios::binary) {
        if (!file_) {
            throw std::runtime_error("无法创建数据文件");
        }

        // 写入文件头
        FileHeader header;
        header.magic = 0x524F424F; // "ROBO"
        header.version = 1;
        header.startTime =
            std::chrono::system_clock::now().time_since_epoch().count();

        file_.write(reinterpret_cast<const char*>(&header), sizeof(header));
    }

    template<typename T>
    void record(uint32_t type, const T& data) {
        std::lock_guard<std::mutex> lock(mutex_);

        RecordHeader recordHeader;
        recordHeader.timestamp = getTimestamp();
        recordHeader.type = type;
        recordHeader.size = sizeof(T);

        file_.write(reinterpret_cast<const char*>(&recordHeader),
sizeof(recordHeader));
        file_.write(reinterpret_cast<const char*>(&data), sizeof(T));
    }

    void flush() {
        std::lock_guard<std::mutex> lock(mutex_);
        file_.flush();
    }
}

private:
    struct FileHeader {
        uint32_t magic;
        uint32_t version;
        int64_t startTime;
    };

    struct RecordHeader {
        int64_t timestamp;
        uint32_t type;
    };
}
```

```

        uint32_t size;
    };

    int64_t getTimestamp() {
        return std::chrono::steady_clock::now().time_since_epoch().count();
    }

    std::ofstream file_;
    std::mutex mutex_;
};

// 数据类型定义
enum class DataType : uint32_t {
    IMU = 1,
    MOTOR = 2,
    TARGET = 3,
    COMMAND = 4
};

// 使用
int main() {
    DataRecorder recorder("match_2024_01_15.dat");

    while (running) {
        // 记录 IMU 数据
        IMUData imu = readIMU();
        recorder.record(static_cast<uint32_t>(DataType::IMU), imu);

        // 记录电机数据
        MotorData motor = readMotor();
        recorder.record(static_cast<uint32_t>(DataType::MOTOR), motor);

        // 记录目标检测结果
        if (hasTarget) {
            recorder.record(static_cast<uint32_t>(DataType::TARGET), target);
        }
    }

    return 0;
}

```

数据回放器：

```

class DataPlayer {
public:
    DataPlayer(const std::string& filename)
        : file_(filename, std::ios::binary) {
        if (!file_) {
            throw std::runtime_error("无法打开数据文件");
        }

        // 读取文件头
        file_.read(reinterpret_cast<char*>(&header_), sizeof(header_));

        if (header_.magic != 0x524F424F) {
            throw std::runtime_error("无效的数据文件格式");
        }
    }
};

```

```

        }
    }

    bool readNext(uint32_t& type, std::vector<char>& data, int64_t& timestamp)
{
    RecordHeader recordHeader;
    if (!file_.read(reinterpret_cast<char*>(&recordHeader),
sizeof(recordHeader))) {
        return false;
    }

    type = recordHeader.type;
    timestamp = recordHeader.timestamp;
    data.resize(recordHeader.size);

    return file_.read(data.data(), recordHeader.size).good();
}

private:
    struct FileHeader {
        uint32_t magic;
        uint32_t version;
        int64_t startTime;
    };

    struct RecordHeader {
        int64_t timestamp;
        uint32_t type;
        uint32_t size;
    };

    std::ifstream file_;
    FileHeader header_;
};

// 回放数据
void replayData(const std::string& filename) {
    DataPlayer player(filename);

    uint32_t type;
    std::vector<char> data;
    int64_t timestamp;

    while (player.readNext(type, data, timestamp)) {
        switch (static_cast<DataType>(type)) {
            case DataType::IMU: {
                IMUData* imu = reinterpret_cast<IMUData*>(data.data());
                std::cout << "IMU: " << imu->gyro[0] << ", "
                    << imu->gyro[1] << ", " << imu->gyro[2] << std::endl;
                break;
            }
            case DataType::MOTOR: {
                MotorData* motor = reinterpret_cast<MotorData*>(data.data());
                std::cout << "Motor: " << motor->speed << " RPM" << std::endl;
                break;
            }
        }
    }
}

```

```

        }
        // ... 其他类型
    }
}

```

文件操作是程序与持久化数据交互的桥梁。在 RoboMaster 开发中，合理使用配置文件可以让参数调整更加灵活，良好的日志系统可以帮助快速定位问题，数据记录功能则为算法分析和改进提供了基础。掌握这些技能，将使你的机器人程序更加健壮和易于维护。

1.5.25. Eigen 矩阵库

机器人的核心是数学，而数学的核心是线性代数。从坐标变换到姿态估计，从卡尔曼滤波到运动学求解，矩阵运算无处不在。Eigen 是 C++ 中最流行的线性代数库，它提供了高效的向量、矩阵运算以及丰富的几何变换功能。在 RoboMaster 开发中，无论是自瞄算法的坐标变换、底盘运动学解算，还是导航定位的状态估计，Eigen 都是不可或缺的工具。

Eigen 是一个纯头文件库，无需编译链接，只需包含头文件即可使用。它利用 C++ 模板和表达式模板技术，在编译时优化矩阵运算，性能可与手写的优化代码媲美。

1.5.25.1. 安装与配置

在 Ubuntu 系统上安装 Eigen：

```
sudo apt install libeigen3-dev
```

在 CMake 项目中使用 Eigen：

```
cmake_minimum_required(VERSION 3.10)
project(my_project)

find_package(Eigen3 REQUIRED)

add_executable(my_program main.cpp)
target_link_libraries(my_program Eigen3::Eigen)
```

使用时包含主头文件：

```
#include <Eigen/Dense> // 稠密矩阵运算
#include <Eigen/Geometry> // 几何变换（四元数、旋转等）

// 或者包含全部功能
#include <Eigen/Eigen>
```

1.5.25.2. 向量与矩阵基础

Eigen 使用模板类表示向量和矩阵。最常用的是固定大小的类型：

```
#include <Eigen/Dense>
#include <iostream>

int main() {
    // 向量（列向量）
    Eigen::Vector2d v2;           // 2 维 double 向量
    Eigen::Vector3d v3;           // 3 维 double 向量
    Eigen::Vector4d v4;           // 4 维 double 向量
    Eigen::Vector3f v3f;          // 3 维 float 向量
    Eigen::Vector3i v3i;          // 3 维 int 向量
```

```

    // 矩阵
    Eigen::Matrix2d m2;           // 2x2 double 矩阵
    Eigen::Matrix3d m3;           // 3x3 double 矩阵
    Eigen::Matrix4d m4;           // 4x4 double 矩阵
    Eigen::Matrix3f m3f;          // 3x3 float 矩阵

    // 通用矩阵类型
    Eigen::Matrix<double, 3, 4> m34; // 3x4 double 矩阵
    Eigen::Matrix<float, 6, 6> m66;  // 6x6 float 矩阵

    // 动态大小矩阵
    Eigen::MatrixXd mDynamic;     // 动态大小 double 矩阵
    Eigen::VectorXd vDynamic;     // 动态大小 double 向量

    return 0;
}

```

类型命名规则：Matrix 或 Vector + 维度（2/3/4/X） + 数据类型（d=double, f=float, i=int）。X 表示动态大小。

初始化向量和矩阵：

```

// 向量初始化
Eigen::Vector3d v1(1.0, 2.0, 3.0);
Eigen::Vector3d v2 = {4.0, 5.0, 6.0}; // C++11

// 使用逗号初始化器
Eigen::Vector3d v3;
v3 << 7.0, 8.0, 9.0;

// 矩阵初始化（按行填充）
Eigen::Matrix3d m;
m << 1, 2, 3,
      4, 5, 6,
      7, 8, 9;

// 特殊矩阵
Eigen::Matrix3d zero = Eigen::Matrix3d::Zero(); // 零矩阵
Eigen::Matrix3d ones = Eigen::Matrix3d::Ones(); // 全 1 矩阵
Eigen::Matrix3d identity = Eigen::Matrix3d::Identity(); // 单位矩阵
Eigen::Matrix3d random = Eigen::Matrix3d::Random(); // 随机矩阵 [-1, 1]
Eigen::Matrix3d constant = Eigen::Matrix3d::Constant(5.0); // 常数矩阵

// 动态矩阵需要指定大小
Eigen::MatrixXd dynMat(3, 4); // 3 行 4 列
dynMat.setZero(); // 设为零矩阵
dynMat.setIdentity(); // 设为单位矩阵（对角线为 1）
dynMat.setRandom(); // 设为随机值

```

访问元素：

```

Eigen::Vector3d v(1, 2, 3);
Eigen::Matrix3d m = Eigen::Matrix3d::Random();

// 向量元素访问
double x = v(0); // 第一个元素
double y = v(1); // 第二个元素

```

```

double z = v(2);           // 第三个元素

// 向量专用访问器
double x2 = v.x();        // 等价于 v(0)
double y2 = v.y();        // 等价于 v(1)
double z2 = v.z();        // 等价于 v(2)

// 矩阵元素访问
double elem = m(1, 2);   // 第 2 行, 第 3 列 (0-indexed)

// 修改元素
v(0) = 10.0;
m(1, 2) = 5.0;

// 获取行和列
Eigen::Vector3d col0 = m.col(0); // 第一列
Eigen::RowVector3d row1 = m.row(1); // 第二行

// 矩阵块操作
Eigen::Matrix2d block = m.block<2, 2>(0, 0); // 左上角 2x2 块
Eigen::Vector2d top2 = v.head<2>();           // 前 2 个元素
Eigen::Vector2d bottom2 = v.tail<2>();          // 后 2 个元素

```

1.5.25.3. 基本运算

向量运算：

```

Eigen::Vector3d v1(1, 2, 3);
Eigen::Vector3d v2(4, 5, 6);

// 加减法
Eigen::Vector3d sum = v1 + v2;
Eigen::Vector3d diff = v1 - v2;

// 标量乘除
Eigen::Vector3d scaled = v1 * 2.0;
Eigen::Vector3d divided = v1 / 2.0;

// 点积
double dot = v1.dot(v2); // 1*4 + 2*5 + 3*6 = 32

// 叉积（仅限 3D 向量）
Eigen::Vector3d cross = v1.cross(v2);

// 范数
double norm = v1.norm();           // L2 范数（长度）
double squaredNorm = v1.squaredNorm(); // 范数的平方（避免开方，更快）

// 归一化
Eigen::Vector3d normalized = v1.normalized(); // 返回单位向量，不修改 v1
v1.normalize(); // 原地归一化

// 元素级运算
Eigen::Vector3d elemMul = v1.cwiseProduct(v2); // 逐元素乘法
Eigen::Vector3d elemDiv = v1.cwiseQuotient(v2); // 逐元素除法
Eigen::Vector3d elemAbs = v1.cwiseAbs();         // 逐元素取绝对值

```

```

// 统计
double maxVal = v1.maxCoeff();
double minVal = v1.minCoeff();
double sum_all = v1.sum();
double mean = v1.mean();

矩阵运算:

Eigen::Matrix3d A = Eigen::Matrix3d::Random();
Eigen::Matrix3d B = Eigen::Matrix3d::Random();
Eigen::Vector3d v = Eigen::Vector3d::Random();

// 加减法
Eigen::Matrix3d sum = A + B;
Eigen::Matrix3d diff = A - B;

// 标量乘法
Eigen::Matrix3d scaled = A * 2.0;

// 矩阵乘法
Eigen::Matrix3d product = A * B;

// 矩阵-向量乘法
Eigen::Vector3d result = A * v;

// 转置
Eigen::Matrix3d At = A.transpose();
A.transposeInPlace(); // 原地转置

// 逆矩阵
Eigen::Matrix3d Ainv = A.inverse();

// 行列式
double det = A.determinant();

// 迹 (对角线元素之和)
double tr = A.trace();

// 对角线
Eigen::Vector3d diag = A.diagonal();
Eigen::Matrix3d diagMat = v.asDiagonal(); // 向量转对角矩阵

```

求解线性方程组:

```

// 求解 Ax = b
Eigen::Matrix3d A;
A << 1, 2, 3,
      4, 5, 6,
      7, 8, 10;

Eigen::Vector3d b(3, 6, 9);

// 方法 1: 直接求逆 (小矩阵可以, 大矩阵不推荐)
Eigen::Vector3d x1 = A.inverse() * b;

// 方法 2: LU 分解 (通用, 较快)
Eigen::Vector3d x2 = A.lu().solve(b);

```

```

// 方法 3: QR 分解 (更稳定)
Eigen::Vector3d x3 = A.colPivHouseholderQr().solve(b);

// 方法 4: LDLT 分解 (对称正定矩阵最快)
Eigen::Matrix3d S = A.transpose() * A; // 对称正定
Eigen::Vector3d x4 = S.ldlt().solve(A.transpose() * b);

// 检验解的精度
double error = (A * x2 - b).norm();
std::cout << "误差: " << error << std::endl;

```

特征值分解:

```

Eigen::Matrix3d A;
A << 1, 2, 1,
      2, 4, 2,
      1, 2, 3;

// 自伴随矩阵 (对称矩阵) 的特征分解
Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> solver(A);

if (solver.info() == Eigen::Success) {
    Eigen::Vector3d eigenvalues = solver.eigenvalues();
    Eigen::Matrix3d eigenvectors = solver.eigenvectors();

    std::cout << "特征值:\n" << eigenvalues << std::endl;
    std::cout << "特征向量:\n" << eigenvectors << std::endl;
}

```

1.5.25.4. 姿态表示

机器人的姿态（朝向）可以用多种方式表示：旋转矩阵、欧拉角、轴角、四元数。每种表示有其优缺点，Eigen 的 Geometry 模块提供了完整支持。

旋转矩阵是最直观的表示， 3×3 正交矩阵，行列式为 1：

```

#include <Eigen/Geometry>

// 绕 X 轴旋转
double angle = M_PI / 4; // 45 度
Eigen::Matrix3d Rx;
Rx << 1, 0, 0,
      0, cos(angle), -sin(angle),
      0, sin(angle), cos(angle);

// 使用 AngleAxis 更方便
Eigen::AngleAxisd rotX(angle, Eigen::Vector3d::UnitX()); // 绕 X 轴
Eigen::AngleAxisd rotY(angle, Eigen::Vector3d::UnitY()); // 绕 Y 轴
Eigen::AngleAxisd rotZ(angle, Eigen::Vector3d::UnitZ()); // 绕 Z 轴

// 转换为旋转矩阵
Eigen::Matrix3d R = rotZ.toRotationMatrix();

// 组合旋转 (注意顺序: 先应用右边的)
Eigen::Matrix3d Rzyx = rotZ.toRotationMatrix() *
                      rotY.toRotationMatrix() *
                      rotX.toRotationMatrix();

```

```
// 应用旋转
Eigen::Vector3d v(1, 0, 0);
Eigen::Vector3d v_rotated = R * v;
```

欧拉角用三个角度表示旋转，直观但有万向锁问题：

```
// 欧拉角 (ZYX 顺序, 常用于航空: yaw-pitch-roll)
double yaw = 0.5;      // 偏航角
double pitch = 0.2;    // 俯仰角
double roll = 0.1;     // 滚转角

// 欧拉角转旋转矩阵
Eigen::Matrix3d R;
R = Eigen::AngleAxisd(yaw, Eigen::Vector3d::UnitZ()) *
    Eigen::AngleAxisd(pitch, Eigen::Vector3d::UnitY()) *
    Eigen::AngleAxisd(roll, Eigen::Vector3d::UnitX());

// 旋转矩阵转欧拉角
Eigen::Vector3d euler = R.eulerAngles(2, 1, 0); // ZYX 顺序
// euler(0) = yaw, euler(1) = pitch, euler(2) = roll
```

四元数是最推荐的姿态表示，避免万向锁，插值平滑，计算高效：

```
// 创建四元数
Eigen::Quaterniond q1 = Eigen::Quaterniond::Identity(); // 单位四元数 (无旋转)

// 从轴角创建
Eigen::Quaterniond q2(Eigen::AngleAxisd(M_PI / 4, Eigen::Vector3d::UnitZ()));

// 从旋转矩阵创建
Eigen::Matrix3d R = Eigen::Matrix3d::Identity();
Eigen::Quaterniond q3(R);

// 直接指定分量 (w, x, y, z)
Eigen::Quaterniond q4(1.0, 0.0, 0.0, 0.0); // w, x, y, z

// 四元数运算
Eigen::Quaterniond q_product = q1 * q2; // 组合旋转
Eigen::Quaterniond q_inv = q2.inverse(); // 逆 (共轭)
Eigen::Quaterniond q_conj = q2.conjugate(); // 共轭

// 归一化
q2.normalize();
Eigen::Quaterniond q_norm = q2.normalized();

// 应用旋转
Eigen::Vector3d v(1, 0, 0);
Eigen::Vector3d v_rotated = q2 * v;

// 四元数转旋转矩阵
Eigen::Matrix3d R2 = q2.toRotationMatrix();

// 四元数插值 (球面线性插值)
double t = 0.5; // 插值参数 [0, 1]
Eigen::Quaterniond q_interp = q1.slerp(t, q2);

// 访问分量
```

```

double w = q2.w();
double x = q2.x();
double y = q2.y();
double z = q2.z();
Eigen::Vector3d xyz = q2.vec(); // 向量部分 (x, y, z)

```

轴角表示：

```

// 轴角：旋转轴 + 旋转角度
Eigen::Vector3d axis(0, 0, 1); // Z 轴
double angle = M_PI / 3; // 60 度

Eigen::AngleAxisd aa(angle, axis);

// 转换
Eigen::Matrix3d R = aa.toRotationMatrix();
Eigen::Quaterniond q(aa);

// 从旋转矩阵恢复
Eigen::AngleAxisd aa2(R);
std::cout << "轴: " << aa2.axis().transpose() << std::endl;
std::cout << "角: " << aa2.angle() << std::endl;

```

1.5.25.5. 坐标变换

齐次变换矩阵结合了旋转和平移：

```

// 创建变换
Eigen::Isometry3d T = Eigen::Isometry3d::Identity();

// 设置旋转
T.rotate(Eigen::AngleAxisd(M_PI / 4, Eigen::Vector3d::UnitZ()));
// 或
T.linear() = Eigen::Matrix3d::Identity(); // 直接设置旋转矩阵部分

// 设置平移
T.pretranslate(Eigen::Vector3d(1, 2, 3));
// 或
T.translation() = Eigen::Vector3d(1, 2, 3);

// 获取旋转和平移
Eigen::Matrix3d R = T.rotation();
Eigen::Vector3d t = T.translation();

// 变换点
Eigen::Vector3d p(1, 0, 0);
Eigen::Vector3d p_transformed = T * p;

// 组合变换
Eigen::Isometry3d T1 = Eigen::Isometry3d::Identity();
Eigen::Isometry3d T2 = Eigen::Isometry3d::Identity();
T1.translate(Eigen::Vector3d(1, 0, 0));
T2.rotate(Eigen::AngleAxisd(M_PI / 2, Eigen::Vector3d::UnitZ()));

Eigen::Isometry3d T_combined = T1 * T2; // 先 T2 后 T1

// 逆变换

```

```

Eigen::Isometry3d T_inv = T.inverse();

// 转换为 4x4 矩阵
Eigen::Matrix4d mat = T.matrix();

```

坐标系之间的变换：

```

// 相机坐标系到世界坐标系的变换
Eigen::Isometry3d T_world_camera = Eigen::Isometry3d::Identity();
T_world_camera.rotate(Eigen::Quaterniond(0.707, 0, 0.707, 0)); // 旋转
T_world_camera.pretranslate(Eigen::Vector3d(0, 0, 1.5)); // 相机高度

// 目标在相机坐标系中的位置
Eigen::Vector3d p_camera(2, 0.5, 5);

// 转换到世界坐标系
Eigen::Vector3d p_world = T_world_camera * p_camera;

// 从世界坐标系转换到相机坐标系
Eigen::Isometry3d T_camera_world = T_world_camera.inverse();
Eigen::Vector3d p_camera_back = T_camera_world * p_world;

```

1.5.25.6. RoboMaster 应用实例

自瞄系统坐标变换：

```

class CoordinateTransformer {
public:
    CoordinateTransformer() {
        // 相机到云台的变换（固定安装参数）
        T_gimbal_camera_ = Eigen::Isometry3d::Identity();
        T_gimbal_camera_.translate(Eigen::Vector3d(0.05, 0, -0.03)); // 相机偏移

        // 云台到底盘的变换（会随云台角度变化）
        T_chassis_gimbal_ = Eigen::Isometry3d::Identity();
    }

    // 更新云台姿态
    void updateGimbalPose(double yaw, double pitch) {
        T_chassis_gimbal_ = Eigen::Isometry3d::Identity();
        T_chassis_gimbal_.rotate(
            Eigen::AngleAxisd(yaw, Eigen::Vector3d::UnitZ()) *
            Eigen::AngleAxisd(pitch, Eigen::Vector3d::UnitY())
        );
    }

    // 将相机坐标系中的目标转换到底盘坐标系
    Eigen::Vector3d cameraToChassisFrame(const Eigen::Vector3d& p_camera) {
        Eigen::Isometry3d T_chassis_camera = T_chassis_gimbal_ *
        T_gimbal_camera_;
        return T_chassis_camera * p_camera;
    }

    // 计算瞄准角度
    Eigen::Vector2d calculateAimAngles(const Eigen::Vector3d& target_camera) {
        // 转换到云台坐标系

```

```

Eigen::Vector3d target_gimbal = T_gimbal_camera_ * target_camera;

// 计算需要的 yaw 和 pitch 角度
double yaw = atan2(target_gimbal.y(), target_gimbal.x());
double pitch = atan2(-target_gimbal.z(),
                     sqrt(target_gimbal.x() * target_gimbal.x() +
                           target_gimbal.y() * target_gimbal.y()));

return Eigen::Vector2d(yaw, pitch);
}

private:
Eigen::Isometry3d T_gimbal_camera_;
Eigen::Isometry3d T_chassis_gimbal_;
};

```

麦克纳姆轮底盘运动学：

```

class MecanumKinematics {
public:
    MecanumKinematics(double wheelRadius, double lx, double ly)
        : r_(wheelRadius), lx_(lx), ly_(ly) {
        // 正运动学矩阵：轮速 -> 底盘速度
        // [vx, vy, omega]^T = J * [w1, w2, w3, w4]^T
        double k = r_ / 4.0;
        J_forward_ << k,   k,   k,   k,
                    -k,   k,   k,   -k,
                    -k/(lx_+ly_), k/(lx_+ly_), -k/(lx_+ly_), k/(lx_+ly_);
        // 逆运动学矩阵：底盘速度 -> 轮速
        // [w1, w2, w3, w4]^T = J_inv * [vx, vy, omega]^T
        double l = lx_ + ly_;
        J_inverse_ << 1, -1, -l,
                    1,  1,  l,
                    1,  1, -l,
                    1, -1,  l;
        J_inverse_ /= r_;
    }

    // 底盘速度 -> 轮速
    Eigen::Vector4d inverseKinematics(double vx, double vy, double omega) {
        Eigen::Vector3d chassisVel(vx, vy, omega);
        return J_inverse_ * chassisVel;
    }

    // 轮速 -> 底盘速度
    Eigen::Vector3d forwardKinematics(const Eigen::Vector4d& wheelSpeeds) {
        return J_forward_ * wheelSpeeds;
    }

private:
    double r_;    // 轮子半径
    double lx_;   // 轮子到中心的 x 距离
    double ly_;   // 轮子到中心的 y 距离
    Eigen::Matrix<double, 3, 4> J_forward_;

```

```

Eigen::Matrix<double, 4, 3> J_inverse_;
};

// 使用
int main() {
    MecanumKinematics kinematics(0.076, 0.2, 0.2); // 轮径 76mm

    // 期望底盘速度
    double vx = 1.0; // 前进 1 m/s
    double vy = 0.5; // 左移 0.5 m/s
    double omega = 0.3; // 逆时针旋转 0.3 rad/s

    // 计算轮速
    Eigen::Vector4d wheelSpeeds = kinematics.inverseKinematics(vx, vy, omega);
    std::cout << "轮速 (rad/s): " << wheelSpeeds.transpose() << std::endl;

    return 0;
}

```

扩展卡尔曼滤波器：

```

class EKF {
public:
    EKF() {
        // 状态: [x, y, theta, vx, vy, omega]
        state_ = Eigen::VectorXd::Zero(6);

        // 协方差矩阵
        P_ = Eigen::MatrixXd::Identity(6, 6) * 0.1;

        // 过程噪声
        Q_ = Eigen::MatrixXd::Identity(6, 6) * 0.01;
        Q_(3, 3) = Q_(4, 4) = Q_(5, 5) = 0.1; // 速度噪声更大

        // 观测噪声 (位置观测)
        R_ = Eigen::MatrixXd::Identity(3, 3) * 0.05;
    }

    // 预测步骤
    void predict(double dt) {
        // 状态转移
        double theta = state_(2);
        double vx = state_(3);
        double vy = state_(4);
        double omega = state_(5);

        // 预测状态
        Eigen::VectorXd state_pred(6);
        state_pred(0) = state_(0) + (vx * cos(theta) - vy * sin(theta)) * dt;
        state_pred(1) = state_(1) + (vx * sin(theta) + vy * cos(theta)) * dt;
        state_pred(2) = state_(2) + omega * dt;
        state_pred(3) = state_(3);
        state_pred(4) = state_(4);
        state_pred(5) = state_(5);

        // 雅可比矩阵
    }
}

```

```

Eigen::MatrixXd F = Eigen::MatrixXd::Identity(6, 6);
F(0, 2) = (-vx * sin(theta) - vy * cos(theta)) * dt;
F(0, 3) = cos(theta) * dt;
F(0, 4) = -sin(theta) * dt;
F(1, 2) = (vx * cos(theta) - vy * sin(theta)) * dt;
F(1, 3) = sin(theta) * dt;
F(1, 4) = cos(theta) * dt;
F(2, 5) = dt;

// 预测协方差
P_ = F * P_ * F.transpose() + Q_;
state_ = state_pred;
}

// 更新步骤 (位置观测)
void updatePosition(const Eigen::Vector3d& z) {
    // 观测矩阵
    Eigen::MatrixXd H = Eigen::MatrixXd::Zero(3, 6);
    H(0, 0) = 1;
    H(1, 1) = 1;
    H(2, 2) = 1;

    // 卡尔曼增益
    Eigen::MatrixXd S = H * P_ * H.transpose() + R_;
    Eigen::MatrixXd K = P_ * H.transpose() * S.inverse();

    // 更新状态
    Eigen::Vector3d y = z - H * state_;

    // 角度归一化
    while (y(2) > M_PI) y(2) -= 2 * M_PI;
    while (y(2) < -M_PI) y(2) += 2 * M_PI;

    state_ = state_ + K * y;

    // 更新协方差
    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(6, 6);
    P_ = (I - K * H) * P_;
}

Eigen::VectorXd getState() const { return state_; }
Eigen::Vector3d getPosition() const { return state_.head<3>(); }
Eigen::Vector3d getVelocity() const { return state_.tail<3>(); }

private:
    Eigen::VectorXd state_;
    Eigen::MatrixXd P_;
    Eigen::MatrixXd Q_;
    Eigen::MatrixXd R_;
};


```

弹道解算：

```

class BallisticSolver {
public:
    BallisticSolver(double bulletSpeed, double gravity = 9.8)

```

```

        : v0_(bulletSpeed), g_(gravity) {}

    // 计算击中目标需要的发射仰角
    // target: 目标相对于枪口的位置 (x: 前方距离, z: 高度差)
    double solve(double x, double z) {
        // 使用迭代法求解
        double pitch = atan2(z, x); // 初始猜测

        for (int i = 0; i < 20; i++) {
            double t = x / (v0_ * cos(pitch)); // 飞行时间
            double z_hit = v0_ * sin(pitch) * t - 0.5 * g_ * t * t; // 落点高度
            double error = z - z_hit;

            if (abs(error) < 0.001) break; // 收敛

            // 调整仰角
            pitch += error * 0.1 / x;
        }

        return pitch;
    }

    // 考虑空气阻力的弹道解算 (使用数值积分)
    Eigen::Vector2d solveWithDrag(const Eigen::Vector3d& target, double
dragCoef) {
        double distance = target.head<2>().norm();
        double yaw = atan2(target.y(), target.x());
        double z = target.z();

        // 二分搜索发射仰角
        double low = -M_PI / 4, high = M_PI / 4;
        double pitch = 0;

        for (int iter = 0; iter < 50; iter++) {
            pitch = (low + high) / 2;
            double z_hit = simulateTrajectory(distance, pitch, dragCoef);

            if (abs(z_hit - z) < 0.001) break;

            if (z_hit < z) {
                low = pitch;
            } else {
                high = pitch;
            }
        }

        return Eigen::Vector2d(yaw, pitch);
    }

private:
    double simulateTrajectory(double targetDist, double pitch, double drag) {
        Eigen::Vector3d pos = Eigen::Vector3d::Zero();
        Eigen::Vector3d vel(v0_ * cos(pitch), 0, v0_ * sin(pitch));
        double dt = 0.0001;
    }
}

```

```

        while (pos.x() < targetDist && pos.z() > -10) {
            // 空气阻力
            double v = vel.norm();
            Eigen::Vector3d dragForce = -drag * v * vel;

            // 重力
            Eigen::Vector3d gravity(0, 0, -g_);

            // 更新速度和位置
            Eigen::Vector3d acc = dragForce + gravity;
            vel += acc * dt;
            pos += vel * dt;
        }

        return pos.z();
    }

    double v0_; // 初速度
    double g_; // 重力加速度
};

```

1.5.25.7. 性能优化技巧

Eigen 提供了多种优化手段：

```

// 1. 使用固定大小矩阵（编译时优化）
Eigen::Matrix3d m3; // 比 MatrixXd 快

// 2. 避免不必要的临时对象
// 不好
Eigen::Matrix3d result = A * B * C * D; // 可能产生临时对象

// 好
Eigen::Matrix3d result;
result.noalias() = A * B * C * D; // 避免别名检查

// 3. 使用 .eval() 强制求值
auto temp = (A * B).eval(); // 立即计算，避免惰性求值的问题

// 4. 内存对齐 (SIMD 优化)
// Eigen 默认对固定大小矩阵进行 16 字节对齐
// 在类中使用时需要特殊宏
class MyClass {
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW // 确保 new 时正确对齐

    Eigen::Vector4d vec; // 需要 32 字节对齐
    Eigen::Matrix4d mat;
};

// 5. 使用 Map 包装现有数组（零拷贝）
double data[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
Eigen::Map<Eigen::Matrix3d> mapped(data);
// mapped 直接操作 data, 无拷贝

// 6. 块操作避免拷贝

```

```

Eigen::Matrix4d big;
big.topLeftCorner<3, 3>() = Eigen::Matrix3d::Identity(); // 直接写入

// 7. 并行计算 (OpenMP)
#ifndef EIGEN_HAS_OPENMP
Eigen::setNbThreads(4); // 设置线程数
#endif

```

1.5.25.8. 常见错误

避免这些常见陷阱:

```

// 1. 混淆固定大小和动态大小
Eigen::Vector3d v3;
Eigen::VectorXd vx(3);
// v3 = vx; // 可能出问题, 建议明确

// 2. 矩阵乘法维度不匹配
Eigen::Matrix<double, 3, 2> A;
Eigen::Matrix<double, 3, 2> B;
// auto C = A * B; // 错误! 3x2 不能乘 3x2

// 3. 自赋值问题
Eigen::MatrixXd A(3, 3);
// A = A * A; // 可能出问题
A = (A * A).eval(); // 正确

// 4. 归一化零向量
Eigen::Vector3d v = Eigen::Vector3d::Zero();
// v.normalize(); // 未定义行为
if (v.norm() > 1e-10) v.normalize(); // 安全

// 5. 忘记初始化
Eigen::Matrix3d m; // 未初始化, 包含垃圾值!
Eigen::Matrix3d m2 = Eigen::Matrix3d::Zero(); // 正确

```

Eigen 是 RoboMaster 开发中最常用的数学库之一。从简单的向量运算到复杂的状态估计, Eigen 提供了高效、易用的接口。熟练掌握 Eigen, 可以让你更专注于算法本身, 而不是底层的数学实现细节。建议多阅读 Eigen 的官方文档, 其中包含了更多高级功能和优化技巧。

1.5.26. Ceres 非线性优化库

在机器人系统中, 许多问题可以归结为优化问题: 相机标定需要找到最小化重投影误差的内参和畸变系数, 状态估计需要找到最符合传感器观测的位姿, 弹道模型需要拟合最接近实际轨迹的参数。这些问题的共同特点是: 目标函数是非线性的, 需要找到使目标函数最小化的参数。

Ceres Solver 是 Google 开发的开源 C++ 库, 专门用于求解非线性最小二乘问题。它在机器人、计算机视觉和机器学习领域广泛使用, 是 SLAM 后端优化、相机标定、传感器融合等应用的标准工具。Ceres 提供了强大的自动求导功能, 让你专注于定义问题本身, 而无需手动计算复杂的雅可比矩阵。

1.5.26.1. 安装与配置

在 Ubuntu 系统上安装 Ceres:

```

# 安装依赖
sudo apt install libgoogle-glog-dev libgflags-dev
sudo apt install libatlas-base-dev libeigen3-dev libsuitesparse-dev

# 安装 Ceres
sudo apt install libceres-dev

```

如果需要最新版本，可以从源码编译：

```

git clone https://ceres-solver.googlesource.com/ceres-solver
cd ceres-solver
mkdir build && cd build
cmake ..
make -j4
sudo make install

```

在 CMake 项目中使用 Ceres：

```

cmake_minimum_required(VERSION 3.10)
project(my_project)

find_package(Ceres REQUIRED)

add_executable(my_program main.cpp)
target_link_libraries(my_program Ceres::ceres)

```

1.5.26.2. 非线性最小二乘问题

Ceres 求解的是如下形式的优化问题：

$$\min_x \sum_{i=1}^n \rho_i(\|f_i(x)\|^2)$$

其中 x 是待优化的参数向量， $f_{i(x)}$ 是残差函数（residual function）， ρ_i 是损失函数（loss function，用于处理异常值）。最简单的情况下， ρ_i 是恒等函数，问题就是最小化残差平方和。

让我们从一个简单的例子开始——拟合曲线 $y = e^{mx+c}$ 到一组带噪声的数据点：

```

#include <ceres/ceres.h>
#include <iostream>
#include <vector>

// 定义残差函数（代价函数）
struct ExponentialResidual {
    ExponentialResidual(double x, double y) : x_(x), y_(y) {}

    // 模板函数，支持自动求导
    template <typename T>
    bool operator()(const T* const m, const T* const c, T* residual) const {
        // residual = y - exp(m * x + c)
        residual[0] = y_ - exp(m[0] * x_ + c[0]);
        return true;
    }

private:
    const double x_;
    const double y_;
};

```

```

int main() {
    // 真实参数
    const double m_true = 0.3;
    const double c_true = 0.1;

    // 生成带噪声的数据
    std::vector<double> x_data, y_data;
    for (int i = 0; i < 100; i++) {
        double x = i * 0.1;
        double y = exp(m_true * x + c_true) + 0.1 * ((rand() % 100) / 100.0
- 0.5);
        x_data.push_back(x);
        y_data.push_back(y);
    }

    // 初始参数猜测
    double m = 0.0;
    double c = 0.0;

    // 构建问题
    ceres::Problem problem;

    for (size_t i = 0; i < x_data.size(); i++) {
        // 添加残差块
        problem.AddResidualBlock(
            // 使用自动求导, 残差维度=1, m维度=1, c维度=1
            new ceres::AutoDiffCostFunction<ExponentialResidual, 1, 1, 1>(
                new ExponentialResidual(x_data[i], y_data[i])
            ),
            nullptr, // 不使用损失函数
            &m, &c // 参数块
        );
    }

    // 配置求解器
    ceres::Solver::Options options;
    options.linear_solver_type = ceres::DENSE_QR;
    options.minimizer_progress_to_stdout = true;

    // 求解
    ceres::Solver::Summary summary;
    ceres::Solve(options, &problem, &summary);

    // 输出结果
    std::cout << summary.BriefReport() << std::endl;
    std::cout << "真实值: m = " << m_true << ", c = " << c_true << std::endl;
    std::cout << "估计值: m = " << m << ", c = " << c << std::endl;

    return 0;
}

```

这个例子展示了 Ceres 的基本使用流程：定义残差函数、构建问题、添加残差块、配置求解器、求解。

1.5.26.3. 核心概念

理解 Ceres 的几个核心概念对于正确使用它至关重要。

残差块 (Residual Block) 是优化问题的基本单元。每个残差块包含一个代价函数和一组参数块。代价函数计算残差向量，Ceres 会最小化所有残差的平方和。

代价函数 (Cost Function) 定义了如何从参数计算残差。Ceres 支持三种方式定义代价函数：

```
// 方式 1: 自动求导 (推荐)
// Ceres 自动计算雅可比矩阵
struct AutoDiffCost {
    template <typename T>
    bool operator()(const T* const x, T* residual) const {
        residual[0] = x[0] * x[0] - 10.0;
        return true;
    }
};

ceres::CostFunction* cost1 =
    new ceres::AutoDiffCostFunction<AutoDiffCost, 1, 1>(new AutoDiffCost);

// 方式 2: 数值求导
// 使用有限差分近似雅可比
struct NumericCost {
    bool operator()(const double* const x, double* residual) const {
        residual[0] = x[0] * x[0] - 10.0;
        return true;
    }
};

ceres::CostFunction* cost2 =
    new ceres::NumericDiffCostFunction<NumericCost, ceres::CENTRAL, 1, 1>(
        new NumericCost);

// 方式 3: 解析求导
// 手动提供雅可比矩阵 (最快但最繁琐)
class AnalyticCost : public ceres::SizedCostFunction<1, 1> {
public:
    bool Evaluate(double const* const* parameters,
                  double* residuals,
                  double** jacobians) const override {
        double x = parameters[0][0];
        residuals[0] = x * x - 10.0;

        if (jacobians != nullptr && jacobians[0] != nullptr) {
            jacobians[0][0] = 2.0 * x; // d(x^2 - 10) / dx = 2x
        }
        return true;
    }
};
```

自动求导是最常用的方式，它利用 C++ 模板和运算符重载，在编译时生成求导代码，精度高且效率接近手写。

参数块 (Parameter Block) 是优化变量的容器。一个参数块可以包含一个或多个标量：

```

double pose[6];           // 6 自由度位姿作为一个参数块
double point[3];          // 3D 点作为一个参数块
double intrinsics[4];     // 相机内参作为一个参数块

problem.AddResidualBlock(cost_function, nullptr, pose, point);

```

损失函数 (Loss Function) 用于降低异常值的影响。当数据中存在错误匹配或噪声异常值时，平方损失会被严重影响。鲁棒损失函数可以减轻这种影响：

```

// 常用损失函数
ceres::LossFunction* huber = new ceres::HuberLoss(1.0);
ceres::LossFunction* cauchy = new ceres::CauchyLoss(0.5);
ceres::LossFunction* tukey = new ceres::TukeyLoss(1.0);

problem.AddResidualBlock(cost_function, huber, &x); // 使用 Huber 损失

```

1.5.26.4. 参数化与流形

某些参数有特殊的约束。例如，四元数必须是单位四元数，旋转矩阵必须是正交矩阵。Ceres 通过流形 (Manifold, 旧版本称为 LocalParameterization) 来处理这些约束：

```

// 四元数参数化 (保持单位长度)
double quaternion[4] = {1, 0, 0, 0}; // w, x, y, z

problem.AddParameterBlock(quaternion, 4);
problem.SetManifold(quaternion, new ceres::EigenQuaternionManifold());

// 或者使用 Sophus 库的 SE3 参数化
// problem.SetManifold(pose, new ceres::ProductManifold<
//   ceres::EigenQuaternionManifold,
//   ceres::EuclideanManifold<3>());

```

固定参数：

```

// 固定某些参数不优化
problem.SetParameterBlockConstant(intrinsics);

// 固定参数块的部分维度
std::vector<int> constant_indices = {0, 1}; // 固定前两个维度
problem.SetManifold(pose,
  new ceres::SubsetManifold(6, constant_indices));

```

参数边界：

```

// 设置参数上下界
problem.SetParameterLowerBound(&x, 0, 0.0); // x >= 0
problem.SetParameterUpperBound(&x, 0, 100.0); // x <= 100

```

1.5.26.5. 求解器配置

Ceres 提供了丰富的求解器选项：

```

ceres::Solver::Options options;

// 线性求解器类型
options.linear_solver_type = ceres::DENSE_QR;           // 小规模稠密问题
options.linear_solver_type = ceres::DENSE_SCHUR;         // BA 问题
options.linear_solver_type = ceres::SPARSE_SCHUR;        // 大规模 BA
options.linear_solver_type = ceres::SPARSE_NORMAL_CHOLESKY; // 大规模稀疏

```

```

// 优化算法
options.trust_region_strategy_type = ceres::LEVENBERG_MARQUARDT; // LM 算法
options.trust_region_strategy_type = ceres::DOGLEG; // Dogleg 算法

// 收敛条件
options.max_num_iterations = 100;
options.function_tolerance = 1e-6;
options.gradient_tolerance = 1e-10;
options.parameter_tolerance = 1e-8;

// 多线程
options.num_threads = 4;

// 输出
options.minimizer_progress_to_stdout = true;

// 求解
ceres::Solver::Summary summary;
ceres::Solve(options, &problem, &summary);

// 检查结果
if (summary.termination_type == ceres::CONVERGENCE) {
    std::cout << "优化收敛" << std::endl;
}

std::cout << summary.FullReport() << std::endl;

```

1.5.26.6. RoboMaster 应用：相机标定

相机标定是确定相机内参（焦距、主点、畸变系数）的过程。使用棋盘格标定板，通过最小化重投影误差来估计参数：

```

#include <ceres/ceres.h>
#include <ceres/rotation.h>
#include <Eigen/Dense>

// 重投影误差
struct ReprojectionError {
    ReprojectionError(double observed_x, double observed_y,
                      double point_x, double point_y, double point_z)
        : observed_x_(observed_x), observed_y_(observed_y),
          point_x_(point_x), point_y_(point_y), point_z_(point_z) {}

    template <typename T>
    bool operator()(const T* const camera_intrinsics, // fx, fy, cx, cy
                    const T* const distortion,           // k1, k2, p1, p2
                    const T* const camera_pose,         // angle-axis (3) +
                    translation (3)
                    T* residuals) const {
        // 3D 点
        T point[3] = {T(point_x_), T(point_y_), T(point_z_)};

        // 旋转
        T rotated_point[3];
        ceres::AngleAxisRotatePoint(camera_pose, point, rotated_point);

```

```

// 平移
T transformed_point[3];
transformed_point[0] = rotated_point[0] + camera_pose[3];
transformed_point[1] = rotated_point[1] + camera_pose[4];
transformed_point[2] = rotated_point[2] + camera_pose[5];

// 归一化相机坐标
T xn = transformed_point[0] / transformed_point[2];
T yn = transformed_point[1] / transformed_point[2];

// 畸变
T r2 = xn * xn + yn * yn;
T radial = T(1.0) + distortion[0] * r2 + distortion[1] * r2 * r2;
T xd = xn * radial + T(2.0) * distortion[2] * xn * yn +
    distortion[3] * (r2 + T(2.0) * xn * xn);
T yd = yn * radial + distortion[2] * (r2 + T(2.0) * yn * yn) +
    T(2.0) * distortion[3] * xn * yn;

// 像素坐标
T predicted_x = camera_intrinsics[0] * xd + camera_intrinsics[2];
T predicted_y = camera_intrinsics[1] * yd + camera_intrinsics[3];

// 残差
residuals[0] = predicted_x - T(observed_x_);
residuals[1] = predicted_y - T(observed_y_);

return true;
}

static ceres::CostFunction* Create(double observed_x, double observed_y,
                                   double point_x, double point_y, double
point_z) {
    return new ceres::AutoDiffCostFunction<ReprojectionError, 2, 4, 4, 6>(
        new ReprojectionError(observed_x, observed_y, point_x, point_y,
point_z));
}

private:
    double observed_x_, observed_y_;
    double point_x_, point_y_, point_z_;
};

class CameraCalibrator {
public:
    void addObservation(int image_id,
                         const Eigen::Vector2d& pixel,
                         const Eigen::Vector3d& point_3d) {
        observations_.push_back({image_id, pixel, point_3d});
    }

    bool calibrate() {
        // 初始化内参猜测
        intrinsics_[0] = 500; // fx
        intrinsics_[1] = 500; // fy
        intrinsics_[2] = 320; // cx

```

```

intrinsics_[3] = 240; // cy

// 初始化畸变系数
std::fill(distortion_, distortion_ + 4, 0.0);

// 初始化每张图像的位姿
int num_images = 0;
for (const auto& obs : observations_) {
    num_images = std::max(num_images, obs.image_id + 1);
}
poses_.resize(num_images * 6, 0.0);

// 构建问题
ceres::Problem problem;

for (const auto& obs : observations_) {
    ceres::CostFunction* cost = ReprojectionError::Create(
        obs.pixel.x(), obs.pixel.y(),
        obs.point_3d.x(), obs.point_3d.y(), obs.point_3d.z());

    problem.AddResidualBlock(
        cost,
        new ceres::HuberLoss(1.0),
        intrinsics_,
        distortion_,
        poses_.data() + obs.image_id * 6);
}

// 求解
ceres::Solver::Options options;
options.linear_solver_type = ceres::DENSE_SCHUR;
options.max_num_iterations = 100;
options.minimizer_progress_to_stdout = true;

ceres::Solver::Summary summary;
ceres::Solve(options, &problem, &summary);

std::cout << summary.BriefReport() << std::endl;
return summary.termination_type == ceres::CONVERGENCE;
}

Eigen::Matrix3d getCameraMatrix() const {
    Eigen::Matrix3d K;
    K << intrinsics_[0], 0, intrinsics_[2],
       0, intrinsics_[1], intrinsics_[3],
       0, 0, 1;
    return K;
}

Eigen::Vector4d getDistortion() const {
    return Eigen::Vector4d(distortion_[0], distortion_[1],
                           distortion_[2], distortion_[3]);
}

private:

```

```

    struct Observation {
        int image_id;
        Eigen::Vector2d pixel;
        Eigen::Vector3d point_3d;
    };

    std::vector<Observation> observations_;
    double intrinsics_[4];
    double distortion_[4];
    std::vector<double> poses_;
};


```

1.5.26.7. RoboMaster 应用：PnP 问题

PnP (Perspective-n-Point) 问题是已知相机内参和 3D-2D 对应点，求解相机位姿：

```

struct PnPError {
    PnPError(const Eigen::Vector2d& observed,
             const Eigen::Vector3d& point_3d,
             const Eigen::Matrix3d& K)
        : observed_(observed), point_3d_(point_3d), K_(K) {}

    template <typename T>
    bool operator()(const T* const pose, T* residuals) const {
        // pose: [qw, qx, qy, qz, tx, ty, tz]
        Eigen::Quaternion<T> q(pose[0], pose[1], pose[2], pose[3]);
        Eigen::Matrix<T, 3, 1> t(pose[4], pose[5], pose[6]);

        // 3D 点转换
        Eigen::Matrix<T, 3, 1> point;
        point << T(point_3d_.x()), T(point_3d_.y()), T(point_3d_.z());

        Eigen::Matrix<T, 3, 1> p_cam = q * point + t;

        // 投影
        T fx = T(K_(0, 0));
        T fy = T(K_(1, 1));
        T cx = T(K_(0, 2));
        T cy = T(K_(1, 2));

        T u = fx * p_cam(0) / p_cam(2) + cx;
        T v = fy * p_cam(1) / p_cam(2) + cy;

        // 残差
        residuals[0] = u - T(observed_.x());
        residuals[1] = v - T(observed_.y());

        return true;
    }

private:
    Eigen::Vector2d observed_;
    Eigen::Vector3d point_3d_;
    Eigen::Matrix3d K_;
};


```

```

class PnP Solver {
public:
    PnP Solver(const Eigen::Matrix3d& K) : K_(K) {}

    bool solve(const std::vector<Eigen::Vector3d>& points_3d,
               const std::vector<Eigen::Vector2d>& points_2d,
               Eigen::Quaterniond& q_out,
               Eigen::Vector3d& t_out) {
        // 初始位姿（可以用 EPnP 等方法初始化）
        double pose[7] = {1, 0, 0, 0, 0, 0, 1}; // qw, qx, qy, qz, tx, ty, tz

        ceres::Problem problem;

        for (size_t i = 0; i < points_3d.size(); i++) {
            ceres::CostFunction* cost =
                new ceres::AutoDiffCostFunction<PnPError, 2, 7>(
                    new PnPError(points_2d[i], points_3d[i], K_));
            problem.AddResidualBlock(cost, new ceres::HuberLoss(1.0), pose);
        }

        // 四元数参数化
        problem.SetManifold(pose, new ceres::ProductManifold<
            ceres::QuaternionManifold,
            ceres::EuclideanManifold<3>>());

        ceres::Solver::Options options;
        options.linear_solver_type = ceres::DENSE_QR;
        options.max_num_iterations = 50;

        ceres::Solver::Summary summary;
        ceres::Solve(options, &problem, &summary);

        if (summary.termination_type == ceres::CONVERGENCE) {
            q_out = Eigen::Quaterniond(pose[0], pose[1], pose[2], pose[3]);
            t_out = Eigen::Vector3d(pose[4], pose[5], pose[6]);
            return true;
        }
        return false;
    }

private:
    Eigen::Matrix3d K_;
};

```

1.5.26.8. RoboMaster 应用：弹道模型参数辨识

通过实际发射数据拟合弹道模型参数：

```

struct BallisticResidual {
    BallisticResidual(double launch_angle, double measured_distance,
                      double measured_drop, double initial_speed)
        : launch_angle_(launch_angle), measured_distance_(measured_distance),
          measured_drop_(measured_drop), v0_(initial_speed) {}

    template <typename T>

```

```

bool operator()(const T* const params, T* residuals) const {
    // params: [drag_coefficient, lift_coefficient]
    T drag = params[0];
    T lift = params[1];

    // 数值积分模拟弹道
    T x = T(0), z = T(0);
    T vx = T(vθ_) * cos(T(launch_angle_));
    T vz = T(vθ_) * sin(T(launch_angle_));

    T dt = T(0.0001);
    T g = T(9.8);

    for (int i = 0; i < 100000 && x < T(measured_distance_); i++) {
        T v = sqrt(vx * vx + vz * vz);
        T ax = -drag * v * vx;
        T az = -g - drag * v * vz + lift * vx;

        vx += ax * dt;
        vz += az * dt;
        x += vx * dt;
        z += vz * dt;
    }

    // 残差: 实际落点与模拟落点的差异
    residuals[0] = z - T(measured_drop_);

    return true;
}

private:
    double launch_angle_;
    double measured_distance_;
    double measured_drop_;
    double vθ_;
};

class BallisticCalibrator {
public:
    void addSample(double launch_angle, double distance,
                  double drop, double speed) {
        samples_.push_back({launch_angle, distance, drop, speed});
    }

    bool calibrate(double& drag_out, double& lift_out) {
        double params[2] = {0.001, 0.0}; // 初始猜测

        ceres::Problem problem;

        for (const auto& s : samples_) {
            problem.AddResidualBlock(
                new ceres::AutoDiffCostFunction<BallisticResidual, 1, 2>(
                    new BallisticResidual(s.angle, s.distance, s.drop,
s.speed)),
                nullptr, params);
        }
    }
}

```

```

    }

    // 参数边界
    problem.SetParameterLowerBound(params, 0, 0.0);      // drag >= 0
    problem.SetParameterUpperBound(params, 0, 0.1);      // drag <= 0.1
    problem.SetParameterLowerBound(params, 1, -0.01);    // lift >= -0.01
    problem.SetParameterUpperBound(params, 1, 0.01);     // lift <= 0.01

    ceres::Solver::Options options;
    options.linear_solver_type = ceres::DENSE_QR;
    options.max_num_iterations = 200;

    ceres::Solver::Summary summary;
    ceres::Solve(options, &problem, &summary);

    drag_out = params[0];
    lift_out = params[1];

    return summary.termination_type == ceres::CONVERGENCE;
}

private:
    struct Sample {
        double angle, distance, drop, speed;
    };
    std::vector<Sample> samples_;
};

```

1.5.26.9. RoboMaster 应用：IMU 内参标定

IMU 存在零偏 (bias)、刻度因子 (scale) 和轴间不正交等误差：

```

struct IMUCalibrationResidual {
    IMUCalibrationResidual(const Eigen::Vector3d& measured_acc,
                           const Eigen::Vector3d& true_gravity)
        : measured_(measured_acc), gravity_(true_gravity) {}

    template <typename T>
    bool operator()(const T* const bias,           // 3 个零偏
                    const T* const scale,         // 3 个刻度因子
                    const T* const misalign,     // 3 个轴间不正交角
                    T* residuals) const {
        // 构建校正矩阵 (下三角)
        Eigen::Matrix<T, 3, 3> M;
        M << scale[0], T(0), T(0),
          misalign[0], scale[1], T(0),
          misalign[1], misalign[2], scale[2];

        // 校正后的测量值
        Eigen::Matrix<T, 3, 1> measured;
        measured << T(measured_.x()), T(measured_.y()), T(measured_.z());

        Eigen::Matrix<T, 3, 1> b;
        b << bias[0], bias[1], bias[2];

        Eigen::Matrix<T, 3, 1> corrected = M * (measured - b);
    }
}

```

```

        // 静止时应该只测量到重力
        Eigen::Matrix<T, 3, 1> g;
        g << T(gravity_.x()), T(gravity_.y()), T(gravity_.z()));

        Eigen::Matrix<T, 3, 1> error = corrected - g;

        residuals[0] = error(0);
        residuals[1] = error(1);
        residuals[2] = error(2);

        return true;
    }

private:
    Eigen::Vector3d measured_;
    Eigen::Vector3d gravity_;
};

class IMUCalibrator {
public:
    // 添加静止状态下不同姿态的测量
    void addStaticMeasurement(const Eigen::Vector3d& acc,
                              const Eigen::Vector3d& expected_gravity) {
        measurements_.push_back({acc, expected_gravity});
    }

    bool calibrate() {
        // 初始参数
        std::fill(bias_, bias_ + 3, 0.0);
        scale_[0] = scale_[1] = scale_[2] = 1.0;
        std::fill(misalign_, misalign_ + 3, 0.0);

        ceres::Problem problem;

        for (const auto& m : measurements_) {
            problem.AddResidualBlock(
                new ceres::AutoDiffCostFunction<IMUCalibrationResidual, 3, 3,
3, 3>(
                    new IMUCalibrationResidual(m.first, m.second),
                    nullptr, bias_, scale_, misalign_);
        }

        // 刻度因子应该接近 1
        for (int i = 0; i < 3; i++) {
            problem.SetParameterLowerBound(scale_, i, 0.9);
            problem.SetParameterUpperBound(scale_, i, 1.1);
        }

        ceres::Solver::Options options;
        options.linear_solver_type = ceres::DENSE_QR;
        options.max_num_iterations = 100;

        ceres::Solver::Summary summary;
        ceres::Solve(options, &problem, &summary);
    }
}

```

```

        std::cout << "IMU 标定结果:" << std::endl;
        std::cout << "零偏: " << bias_[0] << ", " << bias_[1] << ", " <<
bias_[2] << std::endl;
        std::cout << "刻度: " << scale_[0] << ", " << scale_[1] << ", " <<
scale_[2] << std::endl;

    return summary.termination_type == ceres::CONVERGENCE;
}

Eigen::Vector3d correct(const Eigen::Vector3d& raw) const {
    Eigen::Matrix3d M;
    M << scale_[0], 0, 0,
        misalign_[0], scale_[1], 0,
        misalign_[1], misalign_[2], scale_[2];

    Eigen::Vector3d b(bias_[0], bias_[1], bias_[2]);
    return M * (raw - b);
}

private:
    std::vector<std::pair<Eigen::Vector3d, Eigen::Vector3d>> measurements_;
    double bias_[3];
    double scale_[3];
    double misalign_[3];
};

```

1.5.26.10. RoboMaster 应用：手眼标定

手眼标定确定相机与机械臂末端（或云台）之间的固定变换：

```

// AX = XB 问题
// A: 相机运动, B: 云台运动, X: 相机到云台的变换
struct HandEyeResidual {
    HandEyeResidual(const Eigen::Isometry3d& A, const Eigen::Isometry3d& B)
        : A_(A), B_(B) {}

    template <typename T>
    bool operator()(const T* const x_quat, // 4: 四元数
                    const T* const x_trans, // 3: 平移
                    T* residuals) const {
        // X 的旋转和平移
        Eigen::Quaternion<T> q_x(x_quat[0], x_quat[1], x_quat[2], x_quat[3]);
        Eigen::Matrix<T, 3, 1> t_x(x_trans[0], x_trans[1], x_trans[2]);

        // A 的旋转和平移
        Eigen::Quaternion<T> q_a = A_.rotation().cast<T>();
        Eigen::Matrix<T, 3, 1> t_a = A_.translation().cast<T>();

        // B 的旋转和平移
        Eigen::Quaternion<T> q_b = B_.rotation().cast<T>();
        Eigen::Matrix<T, 3, 1> t_b = B_.translation().cast<T>();

        // AX 的旋转和平移
        Eigen::Quaternion<T> q_ax = q_a * q_x;
        Eigen::Matrix<T, 3, 1> t_ax = q_a * t_x + t_a;

```

```

// XB 的旋转和平移
Eigen::Quaternion<T> q_xb = q_x * q_b;
Eigen::Matrix<T, 3, 1> t_xb = q_x * t_b + t_x;

// 旋转误差 (四元数差)
Eigen::Quaternion<T> q_err = q_ax * q_xb.inverse();
residuals[0] = T(2.0) * q_err.x();
residuals[1] = T(2.0) * q_err.y();
residuals[2] = T(2.0) * q_err.z();

// 平移误差
Eigen::Matrix<T, 3, 1> t_err = t_ax - t_xb;
residuals[3] = t_err(0);
residuals[4] = t_err(1);
residuals[5] = t_err(2);

return true;
}

private:
    Eigen::Isometry3d A_, B_;
};

class HandEyeCalibrator {
public:
    void addMotionPair(const Eigen::Isometry3d& camera_motion,
                        const Eigen::Isometry3d& gimbal_motion) {
        motion_pairs_.push_back({camera_motion, gimbal_motion});
    }

    bool calibrate(Eigen::Isometry3d& X_out) {
        // 初始化
        double x_quat[4] = {1, 0, 0, 0};
        double x_trans[3] = {0, 0, 0};

        ceres::Problem problem;

        for (const auto& [A, B] : motion_pairs_) {
            problem.AddResidualBlock(
                new ceres::AutoDiffCostFunction<HandEyeResidual, 6, 4, 3>(
                    new HandEyeResidual(A, B)),
                nullptr, x_quat, x_trans);
        }

        problem.SetManifold(x_quat, new ceres::QuaternionManifold());
    }

    ceres::Solver::Options options;
    options.linear_solver_type = ceres::DENSE_QR;
    options.max_num_iterations = 100;

    ceres::Solver::Summary summary;
    ceres::Solve(options, &problem, &summary);

    if (summary.termination_type == ceres::CONVERGENCE) {

```

```

        Eigen::Quaterniond q(x_quat[0], x_quat[1], x_quat[2], x_quat[3]);
        X_out = Eigen::Isometry3d::Identity();
        X_out.rotate(q);
        X_out.pretranslate(Eigen::Vector3d(x_trans[0], x_trans[1],
x_trans[2]));
        return true;
    }
    return false;
}

private:
std::vector<std::pair<Eigen::Isometry3d, Eigen::Isometry3d>> motion_pairs_;
};
```

1.5.26.11. 调试与常见问题

检查雅可比矩阵:

```

// 使用数值方法验证自动求导的正确性
ceres::Problem problem;
// ... 添加残差块 ...

ceres::NumericDiffOptions numeric_diff_options;
std::vector<const ceres::Manifold*> manifolds;

// 获取参数块
std::vector<double*> parameter_blocks;
problem.GetParameterBlocks(&parameter_blocks);

// 检查每个残差块的雅可比
// 如果自动求导实现有误，这里会报错
```

处理优化不收敛:

```

// 1. 检查初值是否合理
// 2. 检查残差函数是否正确
// 3. 尝试不同的求解器
options.linear_solver_type = ceres::DENSE_QR; // 或 DENSE_SCHUR

// 4. 调整收敛条件
options.function_tolerance = 1e-4; // 放宽

// 5. 增加迭代次数
options.max_num_iterations = 500;

// 6. 检查是否有数值问题
options.check_gradients = true; // 调试时启用
options.gradient_check_relative_precision = 1e-4;

// 7. 使用损失函数处理异常值
problem.AddResidualBlock(cost, new ceres::HuberLoss(1.0), params);
```

性能优化:

```

// 1. 优先使用自动求导，它已经很快
// 2. 对于大规模问题，使用稀疏求解器
options.linear_solver_type = ceres::SPARSE_NORMAL_CHOLESKY;
```

```

// 3. 启用多线程
options.num_threads = std::thread::hardware_concurrency();

// 4. 对于 BA 问题, 使用 Schur 消元
options.linear_solver_type = ceres::SPARSE_SCHUR;

// 5. 如果某些参数不需要优化, 将其设为常量
problem.SetParameterBlockConstant(fixed_params);

// 6. 利用问题结构
ceres::Problem::Options problem_options;
problem_options.cost_function_ownership = ceres::DO_NOT_TAKE_OWNERSHIP;
problem_options.loss_function_ownership = ceres::DO_NOT_TAKE_OWNERSHIP;

```

Ceres 是机器人开发中处理优化问题的利器。它强大的自动求导功能让你可以快速实现复杂的代价函数, 而不必担心求导错误。在 RoboMaster 开发中, 从相机标定到状态估计, 从参数辨识到 SLAM 后端, Ceres 都能发挥重要作用。熟练掌握 Ceres, 将大大提升你解决实际工程问题的能力。

1.5.27. OpenCV 基础

1.5.28. 串口通信

至此, 我们已经完成了 C++ 核心特性的学习。从基础的变量和函数, 到面向对象编程, 再到模板、智能指针和移动语义, 这些知识构成了编写现代 C++ 代码的基础。在实际的 RoboMaster 项目中, 你将综合运用这些技术来构建高效、可靠的机器人控制系统。

1.6. CMake 与构建系统

1.7. 软件工程基础

1.8. 软件工程基础

1.8.1. 为什么需要软件工程

1.8.2. 代码规范与风格

打开一个陌生的代码库, 你最先注意到的是什么? 不是算法的精妙, 也不是架构的优雅, 而是代码的“样子”——变量是怎么命名的、缩进用的是空格还是制表符、大括号放在行尾还是另起一行、注释写了什么。这些看似琐碎的细节, 构成了代码的第一印象, 也在很大程度上决定了你理解这段代码的难易程度。代码规范就是关于这些细节的约定, 它让代码成为团队的共同语言。

1.8.2.1. 为什么需要统一的代码规范

每个程序员都有自己的编码习惯。有人喜欢用 camelCase 命名变量, 有人偏爱 snake_case; 有人习惯在运算符两边加空格, 有人觉得紧凑一些更好; 有人写详尽的注释, 有人信奉“好代码不需要注释”。当一个人独自开发时, 这些差异无关紧要。但当多人协作时, 如果每个人都按照自己的风格编写代码, 代码库很快就会变成一锅大杂烩。

想象一下这样的场景: 你打开一个文件, 前半部分变量名是 `imageWidth`, 后半部分变成了 `image_height`; 有的函数用四个空格缩进, 有的用制表符; 有的大括号跟在函数声明后面, 有的

另起一行。这种风格的不一致会给阅读者带来额外的认知负担——你的大脑需要不断地适应不同的风格，而不能专注于理解代码的逻辑。更糟糕的是，当你需要修改这样的代码时，你不知道应该遵循哪种风格，于是又增加了一种新的风格，混乱进一步加剧。

统一的代码规范解决的正是这个问题。当团队所有成员遵循相同的规范时，代码库呈现出一致的外观，仿佛出自同一人之手。新成员加入时，只需要学习一套规范，就能阅读和编写符合团队标准的代码。代码审查时，审查者可以专注于逻辑和设计，而不是争论风格问题。版本控制中的差异也会更加清晰——如果没有规范，一次简单的修改可能因为格式调整而产生大量无关的改动，淹没真正的变化。

选择哪种规范并不是最重要的，重要的是团队有一个规范并且严格遵守。业界已经有许多成熟的代码规范可供参考，其中 Google C++ Style Guide 是最广泛使用的 C++ 规范之一。它不仅规定了格式和命名，还包含了许多关于 C++ 语言使用的建议，是学习现代 C++ 最佳实践的好材料。接下来，我们以 Google C++ Style Guide 为基础，介绍 C++ 代码规范的核心要点。

1.8.2.2. 命名规范

命名是编程中最重要也最困难的事情之一。好的命名能让代码自解释，读者一眼就能明白变量的含义、函数的功能；糟糕的命名则让代码变成谜语，需要仔细阅读实现才能理解意图。

Google C++ Style Guide 对不同类型的标识符规定了不同的命名风格，这种差异化的命名让读者能够从名字本身判断标识符的类型。

文件名使用小写字母，单词之间用下划线连接。头文件使用 .h 扩展名，源文件使用 .cpp 扩展名。文件名应该反映其内容，通常与其中定义的主要类同名。例如，定义 ImageProcessor 类的文件应该命名为 image_processor.h 和 image_processor.cpp。

```
// 文件命名示例
image_processor.h      // 头文件
image_processor.cpp    // 源文件
robot_controller.h
camera_driver.cpp
```

类型名称使用大驼峰命名法（PascalCase），即每个单词首字母大写，不使用下划线。这包括类、结构体、类型别名、枚举类型和模板参数。这种命名风格让类型名称一眼就能与变量名区分开来。

```
// 类型命名示例
class ImageProcessor;
struct SensorData;
enum class RobotState;
using TargetList = std::vector<Target>;
```



```
template <typename DataType> // 模板参数也用大驼峰
class CircularBuffer;
```

变量名使用小写字母加下划线（snake_case）。这包括局部变量、函数参数和公有成员变量。对于类的私有和保护成员变量，在名称末尾加一个下划线，以便与局部变量区分。这个小小的约定非常有用——当你看到 image_width_ 时，立即知道它是成员变量而非局部变量。

```
// 变量命名示例
int image_width;          // 局部变量
double detection_threshold; // 函数参数
```

```
class Camera {
public:
    int frame_count;           // 公有成员（如果有的话）

private:
    int image_width_;          // 私有成员，末尾加下划线
    double exposure_time_;
    std::string device_path_;
};
```

函数名使用大驼峰命名法，与类型名相同。函数名应该是动词或动词短语，描述函数执行的动作。访问器（getter）和修改器（setter）可以使用与变量类似的命名，如 `image_width()` 和 `set_image_width()`。

```
// 函数命名示例
void ProcessImage();
bool DetectTarget();
int CalculateDistance();

// 访问器和修改器
int image_width() const { return image_width_; }
void set_image_width(int width) { image_width_ = width; }
```

常量和枚举值使用 `k` 前缀加大驼峰命名。这种命名让常量一眼就能被识别出来，避免与变量混淆。宏定义则使用全大写字母加下划线，但现代 C++ 中应尽量避免使用宏。

```
// 常量命名示例
const int kMaxBufferSize = 1024;
constexpr double kPi = 3.14159265358979;

enum class Color {
    kRed,
    kGreen,
    kBlue
};

// 宏命名（尽量避免使用宏）
#define DEPRECATED_FUNCTION __attribute__((deprecated))
```

命名空间使用小写字母，通常是项目名或模块名的缩写。命名空间用于避免全局命名冲突，尤其在大型项目中非常重要。

```
// 命名空间示例
namespace rm {           // RoboMaster 项目
    namespace vision {    // 视觉模块
        class Detector { /* ... */ };
    } // namespace vision
} // namespace rm

// 使用
rm::vision::Detector detector;
```

除了遵循格式规则，命名还有更本质的要求：名称应该准确、完整地描述其代表的含义。避免使用过于简短或含糊的名称，如 `temp`、`data`、`info`、`process()`；也避免使用过于冗长的名称，

如 `the_current_image_frame_from_camera`。好的名称应该在准确和简洁之间取得平衡。在循环中使用 `i`、`j` 作为索引变量是可以接受的约定，但在其他场景下，应该使用更有意义的名称。

```
// 不好的命名
int n;                                // 什么的数量?
void Process();                         // 处理什么? 怎么处理?
std::vector<int> data;                 // 什么数据?

// 好的命名
int target_count;
void ProcessImage();
std::vector<int> detection_scores;
```

1.8.2.3. 格式规范

格式规范涉及代码的视觉呈现：缩进、空格、换行、大括号位置等。良好的格式让代码结构一目了然，就像排版精美的书籍比手写稿更易阅读。

缩进使用空格而非制表符，每级缩进 2 个或 4 个空格（Google 风格是 2 个，许多团队选择 4 个）。制表符在不同编辑器中可能显示为不同宽度，导致代码在你的屏幕上对齐完美，在别人的屏幕上却参差不齐。使用空格可以避免这个问题。

```
// 缩进示例 (2 空格)
class Robot {
    void Move() {
        if (is_enabled_) {
            for (int i = 0; i < 10; ++i) {
                Step();
            }
        }
    }
};
```

每行代码的长度应该限制在一定范围内，通常是 80 或 100 个字符。过长的行需要水平滚动才能看完，降低了可读性。当一行太长时，应该合理地换行。函数调用参数过多时，可以每个参数一行；长表达式可以在运算符处断开。

```
// 长行换行示例
void ProcessTarget(const Target& target,
                    const CameraParams& camera_params,
                    const GimbalState& gimbal_state,
                    std::vector<Result>* results);
```

```
// 长表达式换行
double distance = std::sqrt(
    (target.x - origin.x) * (target.x - origin.x) +
    (target.y - origin.y) * (target.y - origin.y) +
    (target.z - origin.z) * (target.z - origin.z));
```

大括号的位置是一个经典的“圣战”话题。Google C++ Style Guide 规定：函数的左大括号另起一行，其他情况（类定义、控制语句等）左大括号放在行尾。但实际上，许多团队选择统一将左大括号放在行尾（K&R 风格），或者统一另起一行（Allman 风格）。无论选择哪种，团队内部保持一致即可。

```
// Google 风格：函数大括号另起一行，其他在行尾
class Robot {
```

```

void Move() {
    if (is_enabled_) {
        // ...
    }
}

// K&R 风格: 所有大括号都在行尾
class Robot {
    void Move() {
        if (is_enabled_) {
            // ...
        }
    }
};

// Allman 风格: 所有大括号都另起一行
class Robot
{
    void Move()
    {
        if (is_enabled_)
        {
            // ...
        }
    }
};

```

空格的使用应该一致且有助于可读性。二元运算符两边加空格，一元运算符不加；逗号和分号后面加空格，前面不加；控制语句的关键字与括号之间加空格，函数名与括号之间不加。

```

// 空格使用示例
int sum = a + b * c;           // 二元运算符两边加空格
int neg = -value;              // 一元运算符不加
Call(arg1, arg2, arg3);        // 逗号后加空格
for (int i = 0; i < n; ++i)    // for 后加空格, 分号后加空格
if (condition) {               // if 后加空格
    DoSomething();             // 函数名与括号之间不加
}

```

空行用于分隔逻辑上独立的代码块。函数之间用一个空行分隔；函数内部，不同逻辑步骤之间可以用空行分隔；但不要过度使用空行，导致代码过于稀疏。头文件的 #include 部分，通常按组分隔：系统头文件、第三方库头文件、项目头文件，每组之间用空行分隔。

```

#include <vector>
#include <string>

#include <opencv2/opencv.hpp>
#include <Eigen/Dense>

#include "robot/vision/detector.h"
#include "robot/common/types.h"

```

1.8.2.4. 注释规范

注释是代码与人之间的对话。好的注释解释“为什么”而不是“是什么”——代码本身已经说明了它在做什么，注释应该补充代码无法表达的信息：为什么选择这种方案、这里有什么需要注意的陷阱、这个魔法数字的来源是什么。

文件头注释出现在每个文件的开头，说明文件的用途、作者、创建日期、版权信息等。不同团队对文件头的要求不同，有的要求详尽，有的比较简略。

```
// Copyright 2024 RoboMaster Team. All rights reserved.  
//  
// Licensed under the MIT License.  
//  
// Author: Zhang San <zhangsan@example.com>  
// Date: 2024-01-15  
//  
// This file implements the armor detector for RoboMaster robots.  
// The detector uses color and shape features to identify enemy armor plates.
```

类注释放在类定义之前，说明类的用途、使用方法和注意事项。对于复杂的类，还应该说明其线程安全性、生命周期管理等。

```
// ArmorDetector detects enemy armor plates from camera images.  
//  
// Usage:  
//   ArmorDetector detector(config);  
//   detector.Init();  
//   auto armors = detector.Detect(image);  
  
// Thread safety: This class is NOT thread-safe. Each thread should  
// create its own instance.  
class ArmorDetector {  
    // ...  
};
```

函数注释放在函数声明之前，说明函数的功能、参数含义、返回值和可能抛出的异常。对于简单的函数，如果函数名已经足够清晰，可以省略注释。

```
// Detects armor plates in the given image.  
//  
// Args:  
//   image: The input BGR image from camera.  
//   timestamp: The timestamp when the image was captured.  
//  
// Returns:  
//   A vector of detected armor plates, sorted by confidence.  
//   Returns an empty vector if no armor is detected.  
//  
// Throws:  
//   std::invalid_argument if image is empty.  
std::vector<Armor> Detect(const cv::Mat& image, double timestamp);
```

行内注释用于解释单行或几行代码。它们应该放在代码的上方或右侧，解释代码的意图或需要注意的地方。避免写没有信息量的注释，如 `i++; // increment i`。

```
// Apply bilateral filter to reduce noise while preserving edges.  
// Bilateral filter is slower than Gaussian but better for armor detection.  
cv::bilateralFilter(image, filtered, 9, 75, 75);
```

```
int retry_count = 0;
const int kMaxRetries = 3; // Determined by network latency tests
while (retry_count < kMaxRetries) {
    // ...
}
```

TODO 注释用于标记需要后续处理的地方。它们应该包含具体的任务描述，最好还有负责人和预计完成时间。TODO 不应该长期存在——如果一个 TODO 超过一个月还没有处理，要么应该立即处理，要么应该删除或转为正式的任务跟踪。

```
// TODO(zhangsan): Optimize this loop using SIMD. Current implementation
// is too slow for 60fps processing. Expected completion: 2024-02.
for (int i = 0; i < n; ++i) {
    // ...
}
```

1.8.2.5. 头文件规范

头文件的组织对大型项目的编译效率和模块化至关重要。良好的头文件实践可以减少编译依赖、加快编译速度、避免重复包含等问题。

防止头文件重复包含有两种方式：`#pragma once` 和传统的 `#ifndef` 守卫。`#pragma once` 更简洁，被所有主流编译器支持，是现代 C++ 推荐的方式。

```
// 推荐：使用 #pragma once
#pragma once

class MyClass {
    // ...
};

// 传统方式：#ifndef 守卫
#ifndef PROJECT_MODULE_MY_CLASS_H_
#define PROJECT_MODULE_MY_CLASS_H_

class MyClass {
    // ...
};

#endif // PROJECT_MODULE_MY_CLASS_H_
```

头文件中应该只包含必要的声明，实现放在源文件中。尽量使用前向声明（forward declaration）代替包含头文件，减少编译依赖。如果只需要使用指针或引用，就不需要包含完整的类定义。

```
// my_class.h
#pragma once

class OtherClass; // 前向声明，不需要 #include "other_class.h"

class MyClass {
public:
    void Process(OtherClass* obj); // 只用指针，不需要完整定义

private:
    OtherClass* other_; // 只用指针，不需要完整定义
```

```

};

// my_class.cpp
#include "my_class.h"
#include "other_class.h" // 实现时才需要完整定义

void MyClass::Process(OtherClass* obj) {
    obj->DoSomething(); // 调用成员函数需要完整定义
}

```

`#include` 的顺序有助于发现遗漏的依赖。推荐的顺序是：首先包含当前文件对应的头文件（如 `foo.cpp` 首先包含 `foo.h`），然后是系统头文件，接着是第三方库头文件，最后是项目内部头文件。每组之间用空行分隔，每组内部按字母顺序排列。

```

// image_processor.cpp
#include "vision/image_processor.h" // 对应的头文件放第一个

#include <algorithm>
#include <vector>

#include <opencv2/opencv.hpp>
#include <Eigen/Dense>

#include "common/config.h"
#include "vision/detector.h"

```

将对应的头文件放在第一个是一个聪明的技巧：如果这个头文件缺少某些必要的 `#include`，编译这个源文件时会立即报错，而不是等到其他文件包含它时才发现问题。

1.8.2.6. 现代 C++ 的额外建议

Google C++ Style Guide 最初制定时，C++11 还未发布。随着现代 C++ 的发展，一些额外的建议值得补充。

优先使用 `auto` 进行类型推导，特别是当类型很长或很明显时。但不要滥用——当类型对理解代码很重要时，显式写出类型更好。

```

// 好：类型很长或很明显
auto iter = container.begin();
auto result = std::make_unique<MyClass>();
auto lambda = [](int x) { return x * 2; };

// 好：显式类型让意图更清晰
double ratio = GetRatio(); // 而不是 auto ratio = GetRatio();

```

使用初始化列表语法 (`{}`) 初始化变量，它可以防止窄化转换，更加安全。但要注意 `std::vector` 等容器的特殊情况。

```

int value{42};
std::string name{"robot"};
std::vector<int> sizes{1, 2, 3}; // 包含三个元素

// 注意区分
std::vector<int> v1(5);      // 5 个元素，值为 0
std::vector<int> v2{5};       // 1 个元素，值为 5
std::vector<int> v3(5, 1);    // 5 个元素，值为 1

```

使用 `nullptr` 而不是 `NULL` 或 `0` 表示空指针。`nullptr` 有明确的类型，可以避免函数重载时的歧义。

```
void Process(int value);
void Process(int* ptr);

Process(NULL);      // 歧义：可能调用 Process(int)
Process(nullptr);   // 明确：调用 Process(int*)
```

使用范围 `for` 循环遍历容器，更简洁也更容易出错。

```
std::vector<Target> targets = GetTargets();

// 传统方式
for (size_t i = 0; i < targets.size(); ++i) {
    Process(targets[i]);
}

// 现代方式
for (const auto& target : targets) {
    Process(target);
}
```

使用 `enum class` 而不是普通 `enum`，避免枚举值污染外层命名空间，也更类型安全。

```
// 不好：枚举值泄漏到外层
enum Color { Red, Green, Blue };
int Red = 5; // 错误：重定义

// 好：枚举值限定在枚举类内
enum class Color { kRed, kGreen, kBlue };
int Red = 5; // OK
Color c = Color::kRed;
```

使用智能指针管理动态内存，避免手动 `new` 和 `delete`。优先使用 `std::unique_ptr`，只在需要共享所有权时使用 `std::shared_ptr`。

```
// 不好：手动管理内存
MyClass* obj = new MyClass();
// ... 如果中间抛出异常，内存泄漏
delete obj;

// 好：使用智能指针
auto obj = std::make_unique<MyClass>();
// 离开作用域自动释放，异常安全
```

1.8.2.7. 工具辅助

手动维护代码格式既繁琐又容易遗漏。幸运的是，有成熟的工具可以自动化这项工作。

`clang-format` 是 LLVM 项目提供的代码格式化工具，支持多种预设风格（Google、LLVM、Chromium 等），也可以通过配置文件自定义。它可以集成到编辑器中，在保存文件时自动格式化，或者作为 CI 检查的一部分。

```
# 使用 Google 风格格式化文件
clang-format -style=Google -i my_file.cpp

# 使用配置文件格式化
clang-format -style=file -i my_file.cpp
```

```
# 检查是否符合格式（用于 CI）
clang-format -style=file --dry-run --Werror my_file.cpp
```

clang-format 的配置文件名为 .clang-format，放在项目根目录下。以下是一个基于 Google 风格的配置示例，适合 RoboMaster 项目使用：

```
# .clang-format
BasedOnStyle: Google

# 缩进设置
IndentWidth: 4
TabWidth: 4
UseTab: Never
AccessModifierOffset: -4

# 行宽限制
ColumnLimit: 100

# 大括号风格
BreakBeforeBraces: Attach

# 指针和引用的对齐
PointerAlignment: Left
ReferenceAlignment: Left

# 头文件排序
SortIncludes: true
IncludeBlocks: Regroup
IncludeCategories:
- Regex: '^<.*>'
  Priority: 1
- Regex: '^".*"'
  Priority: 2

# 其他
AllowShortFunctionsOnASingleLine: Empty
AllowShortIfStatementsOnASingleLine: Never
AllowShortLoopsOnASingleLine: false
```

clang-tidy 是另一个 LLVM 工具，用于静态代码分析。它不仅检查格式，还能发现潜在的 bug、性能问题和不符合最佳实践的代码。clang-tidy 可以配置启用哪些检查，并能自动修复某些问题。

```
# 运行 clang-tidy 检查
clang-tidy my_file.cpp -- -std=c++17
```

```
# 自动修复可修复的问题
clang-tidy -fix my_file.cpp -- -std=c++17
```

在 VS Code 中，可以安装 C/C++ 扩展和 Clang-Format 扩展，配置保存时自动格式化。在 CLion 中，clang-format 支持是内置的。将格式化工具集成到开发流程中，可以让团队成员无需刻意关注格式问题，工具会自动处理。

1.8.2.8. 代码审查

代码审查（Code Review）是指在代码合并到主分支之前，由其他团队成员检查代码的过程。它是保障代码质量、传播知识、统一风格的重要实践。

代码审查的价值远不止于发现 bug。通过审查，团队成员可以相互学习——资深成员可以指导新成员，新成员的新鲜视角有时也能发现资深成员忽视的问题。审查过程中的讨论可以统一团队对技术问题的认识，形成共同的最佳实践。知道代码会被审查，编写者也会更加用心，不会轻易提交敷衍的代码。

代码审查应该关注什么？首先是正确性——代码是否实现了预期的功能，是否有逻辑错误，边界条件是否处理正确。其次是可维护性——代码是否清晰易懂，命名是否恰当，结构是否合理，是否有足够的注释。然后是一致性——代码是否符合团队规范，是否与代码库中其他部分风格一致。最后是性能和安全——是否有明显的性能问题，是否存在安全隐患。

作为审查者，应该提供建设性的反馈，而不是简单地批评。指出问题的同时，最好给出改进建议。对于风格问题，如果有工具可以自动检查，就不需要在审查中花费过多精力。尊重被审查者的工作，认可做得好的地方。

作为被审查者，应该以开放的心态接受反馈。审查的目的是改进代码，而不是评价人。对于不同意的反馈，可以讨论，但要用事实和理由说服对方，而不是固执己见。将审查意见视为学习机会，而不是对自己的否定。

代码审查清单：

- 代码是否实现了需求描述的功能？
- 逻辑是否正确？边界条件是否处理？
- 命名是否清晰、一致？
- 代码结构是否清晰？函数是否过长？
- 注释是否充分？是否解释了“为什么”？
- 是否有重复代码可以提取？
- 错误处理是否完善？
- 是否有明显的性能问题？
- 是否符合团队代码规范？
- 测试是否充分？

1.8.2.9. RoboMaster 团队代码规范建议

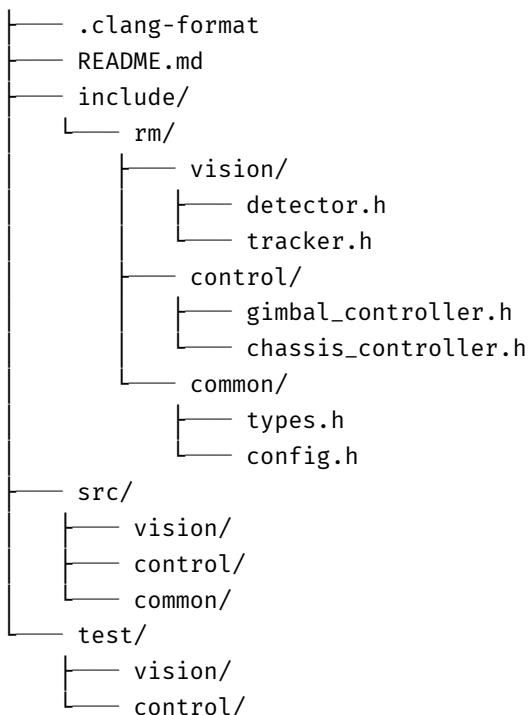
基于以上讨论，这里为 RoboMaster 团队提供一些具体的代码规范建议。这些建议可以作为起点，团队可以根据自己的情况调整。

关于命名，建议使用 Google 风格：类型用 `PascalCase`，变量和函数参数用 `snake_case`，成员变量末尾加下划线，常量用 `kPascalCase`。对于 RoboMaster 项目中的特定概念，建议统一术语：敌方装甲板用 `Armor`，云台用 `Gimbal`，底盘用 `Chassis`，自瞄用 `AutoAim` 等。

关于格式，建议使用 4 空格缩进、100 字符行宽、K&R 大括号风格。这与 ROS 社区的习惯相近，也是许多团队的偏好。使用 `clang-format` 自动化格式检查，将配置文件纳入版本控制。

关于项目结构，建议按功能模块组织代码：`vision/`（视觉）、`control/`（控制）、`communication/`（通信）、`common/`（公共）等。每个模块有自己的头文件目录和源文件目录。使用命名空间与目录结构对应，如 `rm::vision`、`rm::control`。

```
robomaster_project/
└── CMakeLists.txt
```



关于注释，建议所有公开的类和函数都有文档注释，说明功能、参数和返回值。对于复杂的算法，添加解释原理的注释或引用相关论文。对于临时的解决方案或已知的限制，使用 TODO 或 FIXME 标记。

关于代码审查，建议所有代码在合并前至少经过一人审查。对于核心模块的修改，建议有两人审查。审查时使用检查清单，确保不遗漏重要方面。将代码审查作为团队文化的一部分，而不是可选的流程。

代码规范不是一成不变的教条，而是团队协作的工具。最重要的是团队达成共识，并且一致地执行。工具可以帮助自动化格式检查，但对命名、设计、文档的关注需要每个成员的自觉。当遵循规范成为习惯，代码质量自然会提升，团队协作也会更加顺畅。

1.8.3. 设计模式的起源

设计模式(Design pattern)是软件开发者在长期实践中提炼出的、可复用的代码设计经验，它们经过大量项目验证，被系统化分类，并形成了完整的知识体系。设计模式是软件工程的基石脉络，如同大厦的结构一样。

设计模式使代码编制真正工程化，是软件工程的基石。在项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，且都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

1994 年，Erich Gamma、Richard Helm、Ralph Johnson 与 John Vlissides 四人共同出版了划时代的著作 ***Design Patterns: Elements of Reusable Object-Oriented Software*** (中文译名：《设计模式——可复用面向对象软件的基础》)。该书首次系统化地阐述了软件开发中“设计模式”这一核心概念，为面向对象软件设计奠定了理论基石。

在这本书中，GoF 将设计模式系统地归纳为三大类，共提出了“**23 种经典设计模式**”，这些模式奠定了面向对象软件设计的基础框架。

然而需要指出的是，书中讨论的许多模式主要基于 Java 等当时主流的面向对象语言。在 C++ —— 尤其是 现代 C++ (Modern C++) 的发展背景下，由于语言本身拥有更强大的表达能力（如 RAII、模板、智能指针、`constexpr` 等特性），其中部分设计模式已不再必要，甚至被语言机制自然替代。

尽管如此，现代软件工程仍将设计模式分为三大类：

1. 创建型模式 (Creational Patterns)

关注对象的创建方式，以提升灵活性与可扩展性。

2. 结构型模式 (Structural Patterns)

关注类与对象的组织结构，使系统模块更清晰、更易复用。

3. 行为型模式 (Behavioral Patterns)

关注对象间的协作方式与职责划分，提升系统行为的稳定性与扩展性。

这些模式依然构成理解现代架构思想的重要基础，只是在 C++ 语境下，其实现方式随着语言进化而更为简洁与自然。

1.8.4. 设计模式的基本原则

设计模式的核心目标是实现高内聚、低耦合的软件架构。所谓“高内聚”，是指一个模块内部的各个元素紧密相关、职责明确；而“低耦合”则强调模块之间的依赖关系应尽可能松散，以便于系统的维护、扩展与升级。

为了达成这一目标，软件工程界总结出了以下七大设计原则，它们共同构成了设计模式的理论基石：

1. 开闭原则 (Open-Closed Principle, OCP)

对扩展开放，对修改关闭。

软件实体（类、模块、函数等）应当在不修改现有代码的前提下进行功能扩展。这一原则是实现系统良好扩展性与可维护性的关键。想要达到这样的效果，需要充分利用接口、抽象类等机制，将易变部分与稳定部分分离。

开闭原则是设计模式的总纲，其他原则都是实现开闭原则的具体手段。

2. 单一职责原则 (Single Responsibility Principle, SRP)

一个类应该只有一个引起它变化的原因。

类的职责应当单一明确，对外只提供一种功能。当一个类承担的职责过多时，这些职责之间可能会相互耦合，一个职责的变化可能会削弱或抑制这个类完成其他职责的能力。遵循单一职责原则可以提高类的可读性、可维护性，降低变更引起的风险。

3. 里氏替换原则 (Liskov Substitution Principle, LSP)

任何基类可以出现的地方，子类一定可以出现。

里氏替换原则是面向对象设计的基本原则之一，是继承复用的基石。它要求子类在继承父类时，除了添加新的方法完成新增功能外，尽量不要重写父类的方法。只有当派生类可以替换掉基类，且软件单位的功能不受影响时，基类才能真正被复用。

里氏替换原则是对开闭原则的补充，它通过规范继承关系来实现抽象化，从而支撑开闭原则的实现。这一原则在 C++ 中通过虚函数机制在语言层面得到了实现。

4. 依赖倒置原则 (Dependence Inversion Principle, DIP)

依赖于抽象，不要依赖于具体实现。

高层模块不应该依赖低层模块，两者都应该依赖其抽象（接口或抽象类）。抽象不应该依赖细节，细节应该依赖抽象。这一原则是开闭原则的基础，强调“面向接口编程”而非“面向实现编程”，从而降低模块间的耦合度。

5. 接口隔离原则 (Interface Segregation Principle, ISP)

使用多个专门的接口，而不使用单一的总接口。

客户端不应该被迫依赖它不使用的接口方法。一个接口应该只提供一种对外功能，不应该把所有操作都封装到一个接口中。接口隔离原则要求将臃肿的接口拆分为多个细粒度的接口，降低类之间的耦合度，提高系统的灵活性。

6. 合成复用原则 (Composite Reuse Principle, CRP)

优先使用对象组合，而不是继承来达到复用的目的。

在复用代码时，应优先选择对象组合（一个对象包含另一个对象）的方式，而不是使用类继承。继承会导致父类的任何变化都可能影响到子类的行为，增加了系统的脆弱性；而对象组合则降低了这种依赖关系，使系统更加灵活。这一原则是对里氏替换原则的补充，提示我们思考功能组合实现的正确方式。

7. 迪米特法则 (Law of Demeter, LoD)

一个对象应当对其他对象有尽可能少的了解。

也称为**最少知识原则**，要求一个实体应当尽量少地与其他实体发生相互作用，使得系统功能模块相对独立，从而降低各个对象之间的耦合，提高系统的可维护性。

例如在程序设计中，各个模块之间相互调用时，通常会提供一个统一的接口来实现功能。这样其他模块不需要了解模块内部的实现细节（黑盒原理），当一个模块内部的实现发生改变时，不会影响其他模块的使用。

1.8.5. 常用设计模式详解

1.8.6. 单元测试

1.8.7. 调试技巧

1.8.8. 性能分析与优化

1.8.9. 文档编写

代码是程序员与机器的对话，而文档是程序员与人的对话——可能是与队友、与未来的维护者，也可能是与几个月后已经忘记实现细节的自己。许多程序员不喜欢写文档，觉得这是编码之外的“杂事”，浪费时间。但当你在凌晨三点调试一个陌生的模块，翻遍代码却找不到任何解释时；当新队员接手项目，花了一周时间还没搞清楚系统架构时；当你试图使用去年写的库，却想不起那些参数是什么含义时——你会深刻体会到文档的价值。好的文档是项目的第二生命，它让知识得以传承，让协作成为可能。

1.8.9.1. 文档的价值

文档最直接的价值是降低沟通成本。在一个团队中，如果每个问题都需要口头解释，那么回答问题的人会被频繁打断，提问的人也要等待对方有空。而一份好的文档可以回答大多数常见问

题，让团队成员能够自助式地获取信息。当有人问“这个函数怎么用”或“这个模块是做什么的”时，你可以说“看文档”，而不是每次都从头解释一遍。

文档的另一个重要价值是保存知识。人的记忆是不可靠的，今天清清楚楚的设计决策，三个月后可能就想不起来了。更何况团队成员会毕业、会换项目，如果知识只存在于人的脑子里，人走了知识也就丢了。文档将知识外化，使其独立于任何个人而存在。一份记录了设计思路、架构决策、踩过的坑的文档，是团队最宝贵的资产之一。

文档还能帮助作者理清思路。写文档的过程迫使你用清晰的语言解释自己的设计，这个过程往往会展露出之前没有意识到的问题。如果你发现某个部分很难解释清楚，很可能是因为设计本身就不够清晰。有经验的工程师常常在写代码之前先写设计文档，通过写作来思考和验证方案。

对于 RoboMaster 团队来说，文档还有一层特殊的意义：赛季传承。每年都有老队员毕业、新队员加入，如果没有文档，新人只能从零开始摸索，前人的经验无法积累。而有了好的文档，新队员可以快速了解系统全貌，在前人的基础上继续前进，而不是每年都在重复发明轮子。

1.8.9.2. 代码即文档

在讨论如何写文档之前，我们首先要认识到：最好的文档是代码本身。如果代码写得足够清晰，很多时候不需要额外的解释。这就是“代码即文档”（Code as Documentation）的理念。

清晰的命名是代码自解释的基础。一个名为 `CalculateProjectileDropCompensation` 的函数，不需要注释也能让人明白它是计算弹道下坠补偿的。而一个名为 `Calc` 的函数，即使有注释也需要读者花费额外的精力去理解。变量名也是如此：`remainingAmmo` 比 `n` 有意义得多。好的命名就像好的路标，让读者不需要地图也能找到方向。

合理的代码结构也是一种文档。当一个函数只做一件事、一个类只有一个职责时，代码的意图就很明显。当相关的功能被组织在一起、模块之间有清晰的边界时，系统的架构就呼之欲出。相反，如果一个函数做了十件不同的事，即使每一行都有注释，也很难理解它的整体逻辑。

类型系统也是文档的一部分。在 C++ 中，使用强类型而不是原始类型可以传达更多信息。例如，`Angle` 类型比 `double` 更能说明参数的含义；`std::optional<Target>` 比返回空指针更能表达“可能没有结果”的语义；`enum class RobotState` 比一堆整数常量更能说明状态的含义和取值范围。

```
// 不好：类型没有传达信息
double Calculate(double a, double b, int mode);

// 好：类型本身就是文档
Velocity CalculateVelocity(Distance distance, Duration time);
std::optional<Target> DetectTarget(const Image& image);
```

常量和配置也应该自解释。不要在代码中使用魔法数字，而是定义有意义的常量。当读者看到 `kMaxDetectionDistance` 时，他立刻知道这是最大检测距离；而看到 `5.0` 时，他只能猜测这个数字的含义。

```
// 不好：魔法数字
if (distance < 5.0 && confidence > 0.8) {
    Fire();
}

// 好：有意义的常量
```

```

constexpr double kMaxFiringDistance = 5.0; // meters
constexpr double kMinConfidenceThreshold = 0.8;

if (distance < kMaxFiringDistance && confidence > kMinConfidenceThreshold) {
    Fire();
}

```

然而，代码即文档有其局限性。代码能够说明“是什么”(what)和“怎么做”(how)，但很难表达“为什么”(why)。为什么选择这个算法而不是那个？为什么这个参数是5.0而不是3.0？为什么要加这个看起来多余的检查？这些问题需要注释和文档来回答。

1.8.9.3. 注释的艺术

注释是代码中嵌入的文档，用于解释代码本身无法表达的信息。好的注释能够显著提高代码的可读性，而糟糕的注释则可能误导读者、浪费时间。掌握何时写注释、写什么注释，是一门需要练习的艺术。

首先要明确的是，注释不是用来解释代码“做了什么”的。如果代码需要注释来解释它在做什么，这通常意味着代码本身不够清晰，应该重构而不是添加注释。那些像`i++; // 将 i 加 1`这样的注释毫无价值，只会增加阅读负担。

注释真正的价值在于解释“为什么”。为什么选择这种实现方式？有什么约束或权衡？这里有什么需要注意的陷阱？这些是代码本身无法表达的信息，是注释的用武之地。

```

// 不好：解释"是什么"
// 遍历所有目标
for (const auto& target : targets) {
    // 如果目标在范围内
    if (target.distance < max_distance) {
        // 添加到结果中
        results.push_back(target);
    }
}

// 好：解释"为什么"
// 优先处理近距离目标，因为它们对比赛结果影响更大。
// 远距离目标的检测置信度较低，容易产生误判。
for (const auto& target : targets) {
    if (target.distance < max_distance) {
        results.push_back(target);
    }
}

```

注释还应该解释非显而易见的代码。有时候，为了性能、兼容性或规避某个bug，代码不得不写得比较晦涩。这时候，注释可以解释这样写的原因，避免后人“好心”地重构掉这些看起来奇怪的代码。

```

// 使用位运算代替除法，在 ARM 平台上快约 10 倍。
// 仅当 divisor 是 2 的幂时有效。
int FastDivide(int value, int divisor) {
    int shift = __builtin_ctz(divisor); // 计算尾随零的个数
    return value >> shift;
}

// OpenCV 4.2 之前的版本在这里有内存泄漏，需要手动释放。

```

```
// 参见: https://github.com/opencv/opencv/issues/12345
cv::Mat temp;
// ... 使用 temp ...
temp.release(); // 必要的手动释放
```

注释应该解释假设和约束。函数对输入有什么要求？调用时需要满足什么前置条件？有什么副作用？这些信息可以帮助调用者正确使用代码。

```
// 计算两点之间的角度。
// 假设：两点不重合 (distance > 0)。
// 返回值：弧度，范围 [-π, π]。
// 注意：此函数不是线程安全的，因为它使用了全局缓存。
double CalculateAngle(const Point& from, const Point& to);
```

注释要保持与代码同步。过时的注释比没有注释更糟糕，因为它会误导读者。当修改代码时，一定要检查相关注释是否需要更新。如果一段注释描述的行为与代码不符，读者会困惑：是代码错了还是注释错了？为了降低注释过时的风险，注释应该描述意图和原因，而不是复述代码的实现细节——意图通常比实现稳定。

TODO 和 FIXME 注释用于标记待办事项和已知问题。它们是对未来的承诺，提醒自己或他人这里还有工作要做。好的 TODO 注释应该包含具体的任务描述、负责人和预期时间。定期清理 TODO 是良好的习惯——如果一个 TODO 存在超过一个月还没处理，要么立即处理，要么承认它不重要并删除。

```
// TODO(zhangsan): 实现多目标跟踪。当前只支持单目标。
// 预计下周完成。

// FIXME: 在光线变化剧烈时会产生误检。
// 需要添加时域滤波来改善稳定性。

// HACK: 临时解决方案，等待上游库修复后移除。
// 跟踪 issue: https://github.com/xxx/xxx/issues/123
```

1.8.9.4. README：项目的封面

README 是项目的封面，通常是用户接触项目的第一份文档。一份好的 README 能让人在几分钟内了解项目是什么、能做什么、怎么开始使用。它应该简洁但完整，回答读者最关心的问题。

一份标准的 README 通常包含以下部分：

项目标题和简介放在最开头，用一两句话说明项目是什么、解决什么问题。如果有项目 logo 或徽章（构建状态、版本号等），可以放在这里。

RoboMaster Vision System

基于深度学习的 RoboMaster 自瞄视觉系统，支持装甲板检测、跟踪和预测。

```
![Build Status](https://img.shields.io/badge/build-passing-green)
![Version](https://img.shields.io/badge/version-2.0.0-blue)
```

功能特性列出项目的主要功能，让读者快速了解项目能做什么。可以用列表形式简洁地呈现。

功能特性

- 📺 实时装甲板检测 (60+ FPS @ 1080p)

- 多目标跟踪与 ID 关联
- 基于卡尔曼滤波的运动预测
- 支持 ROS 2 Humble 集成
- CUDA 加速推理

快速开始是最重要的部分之一，它应该让读者能够在最短时间内运行项目。包括环境要求、安装步骤和基本使用示例。命令应该可以直接复制执行，不要让读者猜测。

快速开始

环境要求

- Ubuntu 22.04
- ROS 2 Humble
- CUDA 11.8+
- OpenCV 4.5+

安装

```
```bash
克隆仓库
git clone https://github.com/your-team/rm_vision.git
cd rm_vision

安装依赖
./scripts/install_dependencies.sh

编译
colcon build --symlink-install
```

```

运行

```
```bash
启动检测节点
ros2 launch rm_vision detector.launch.py

使用录制的数据测试
ros2 launch rm_vision detector.launch.py use_bag:=true bag_path:=/path/to/bag
```

```

配置说明解释主要的配置选项，让用户知道如何根据自己的需求调整系统。可以提供配置文件的示例和各选项的含义。

配置

配置文件位于 `config/detector_params.yaml`：

```
```yaml
detector:
 model_path: "models/armor_yolov8.onnx" # 模型路径
 confidence_threshold: 0.7 # 置信度阈值
 nms_threshold: 0.4 # NMS 阈值
 target_color: "red" # 目标颜色
```

```

项目结构帮助读者理解代码的组织方式，特别是对于想要深入了解或贡献代码的人。

项目结构

```
```
rm_vision/
├── rm_vision/ # 主功能包
│ ├── detector/ # 装甲板检测
│ ├── tracker/ # 目标跟踪
│ └── predictor/ # 运动预测
├── rm_interfaces/ # 消息和服务定义
├── rm_bringup/ # 启动文件
├── config/ # 配置文件
├── models/ # 预训练模型
└── docs/ # 详细文档
```

```

其他可选部分包括：详细文档的链接、贡献指南、许可证信息、致谢、联系方式等。对于开源项目，这些信息尤其重要。

文档

详细文档请参阅 [Wiki](https://github.com/your-team/rm_vision/wiki)。

贡献

欢迎贡献！请阅读 [贡献指南](CONTRIBUTING.md) 了解如何参与。

许可证

本项目采用 MIT 许可证。详见 LICENSE。

致谢

- 感谢 [YOLO](<https://github.com/ultralytics/yolov5>) 提供的目标检测框架
- 感谢历届队员的贡献

联系我们

- 邮箱: robomaster@example.com
- QQ 群: 123456789

1.8.9.5. API 文档与 Doxygen

对于库或框架，API 文档是不可或缺的。它详细描述每个类、函数、参数的用法，是开发者使用库时的参考手册。手写 API 文档工作量大且容易过时，因此通常使用工具从代码中的注释自动生成。Doxygen 是 C++ 社区最流行的文档生成工具。

Doxygen 通过解析代码中特定格式的注释来生成文档。最常用的注释风格是 Javadoc 风格（以 `/**` 开头）和 Qt 风格（以 `/*!` 开头）。以下是一些常用的 Doxygen 命令：

```
/**
 * @file armor_detector.h
 * @brief 装甲板检测器的头文件
 * @author Zhang San
 * @date 2024-01-15
 */
```

```

/**
 * @brief 装甲板检测器类
 *
 * ArmorDetector 使用深度学习模型检测图像中的装甲板。
 * 它支持红色和蓝色装甲板的检测，并能处理不同光照条件。
 *
 * @note 此类不是线程安全的。每个线程应创建独立的实例。
 *
 * 使用示例：
 * @code
 * ArmorDetector detector(config);
 * detector.Init();
 * auto armors = detector.Detect(image);
 * for (const auto& armor : armors) {
 *     std::cout << "检测到装甲板，置信度：" << armor.confidence << std::endl;
 * }
 * @endcode
 *
 * @see Tracker 用于目标跟踪
 * @see Predictor 用于运动预测
 */
class ArmorDetector {
public:
    /**
     * @brief 构造函数
     * @param config 检测器配置参数
     * @throws std::invalid_argument 如果配置无效
     */
    explicit ArmorDetector(const DetectorConfig& config);

    /**
     * @brief 初始化检测器
     *
     * 加载模型并准备推理引擎。此方法必须在 Detect() 之前调用。
     *
     * @return true 如果初始化成功
     * @return false 如果初始化失败（如模型文件不存在）
     */
    bool Init();

    /**
     * @brief 检测图像中的装甲板
     *
     * @param image 输入图像，必须是 BGR 格式
     * @param timestamp 图像时间戳，用于时间同步
     * @return 检测到的装甲板列表，按置信度降序排列
     *
     * @pre Init() 已成功调用
     * @pre image 不为空
     *
     * @warning 此方法会修改内部状态，不要在多线程中共享实例
     */
    std::vector<Armor> Detect(const cv::Mat& image, double timestamp);

}

```

```

    * @brief 设置目标颜色
    * @param color 目标颜色
    * @see TargetColor
    */
void SetTargetColor(TargetColor color);

/**
 * @brief 获取当前检测统计信息
 * @return 包含检测帧数、平均耗时等的统计信息
 */
DetectorStats GetStats() const;

private:
    DetectorConfig config_; //;< 检测器配置
    bool initialized_; //;< 是否已初始化
    // ...
};

/***
 * @brief 目标颜色枚举
 */
enum class TargetColor {
    kRed, //;< 红色方
    kBlue //;< 蓝色方
};

/***
 * @struct Armor
 * @brief 装甲板检测结果
 */
struct Armor {
    cv::Point2f center; //;< 装甲板中心点（像素坐标）
    cv::Size2f size; //;< 装甲板尺寸（像素）
    double confidence; //;< 检测置信度，范围 [0, 1]
    int id; //;< 装甲板编号（1-5 对应英雄到哨兵）
    TargetColor color; //;< 装甲板颜色
};

```

配置和运行 Doxygen 非常简单。首先安装 Doxygen：

```
sudo apt install doxygen graphviz
```

然后在项目根目录生成配置文件：

```
doxygen -g Doxyfile
```

编辑 Doxyfile 配置主要选项：

```

# 项目信息
PROJECT_NAME          = "RoboMaster Vision"
PROJECT_NUMBER        = 2.0.0
PROJECT_BRIEF         = "自瞄视觉系统"

# 输入设置
INPUT                 = include src
RECURSIVE             = YES
FILE_PATTERNS         = *.h *.hpp *.cpp

# 输出设置

```

```

OUTPUT_DIRECTORY      = docs/api
GENERATE_HTML        = YES
GENERATE_LATEX       = NO

# 提取设置
EXTRACT_ALL          = NO
EXTRACT_PRIVATE      = NO
EXTRACT_STATIC        = YES

# 图表
HAVE_DOT              = YES
CALL_GRAPH             = YES
CALLER_GRAPH           = YES

```

运行 Doxygen 生成文档：

`doxygen Doxyfile`

生成的 HTML 文档在 `docs/api/html/` 目录下，用浏览器打开 `index.html` 即可查看。

1.8.9.6. 项目文档类型

除了代码中的注释和 API 文档，一个完整的项目还需要其他类型的文档。不同文档面向不同的读者、服务于不同的目的。

需求文档描述系统应该做什么。它定义功能需求（系统应该具备什么功能）和非功能需求（性能、可靠性、可维护性等要求）。需求文档是开发的起点，所有后续工作都是为了满足需求。在 RoboMaster 中，需求可能包括：检测精度要达到多少、延迟不能超过多少毫秒、要支持哪些目标类型等。

自瞄系统需求文档

功能需求

FR-001: 装甲板检测

- 系统应能检测红色和蓝色装甲板
- 支持识别装甲板编号（1-5）
- 检测距离范围：1-8 米

FR-002: 目标跟踪

- 系统应能跟踪多个目标
- 支持目标遮挡后重新识别
- 跟踪 ID 应保持稳定

非功能需求

NFR-001: 性能

- 端到端延迟 < 20ms
- 检测帧率 >= 60 FPS

NFR-002: 可靠性

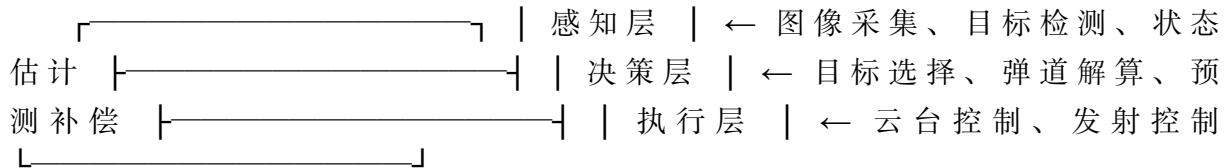
- 误检率 < 1%
- 连续运行 8 小时无崩溃

设计文档描述系统如何实现需求。它包括系统架构、模块划分、接口定义、数据流、关键算法等。设计文档是开发者的蓝图，帮助团队成员理解系统的整体结构和设计决策。好的设计文档不仅说明“怎么做”，还要解释“为什么这么做”。

自瞄系统设计文档

系统架构

系统采用三层架构：感知层、决策层、执行层。



模块设计

检测模块

采用 YOLOv8 进行装甲板检测，原因：

1. 速度快，满足实时性要求
2. 准确率高，尤其对小目标
3. 社区支持好，便于优化

跟踪模块

采用 EKF 进行状态估计，状态向量：

- 位置 (x, y, z)
- 速度 (v_x, v_y, v_z)
- 装甲板朝向 θ

接口定义

...

用户手册面向最终用户，说明如何安装、配置和使用系统。它应该假设读者不了解内部实现，用清晰的语言和步骤指导用户完成任务。对于 RoboMaster 项目，用户可能是操作手或调试人员。

变更日志 (CHANGELOG) 记录项目的版本历史和每个版本的变化。它帮助用户了解版本之间的差异，决定是否升级，以及升级时需要注意什么。一个常见的格式是 Keep a Changelog：

变更日志

本项目遵循 [语义化版本](<https://semver.org/>)。

[2.1.0] - 2024-03-15

新增

- 支持能量机关检测
- 添加录像回放功能

变更

- 升级 YOLOv8 模型，检测速度提升 20%
- 调整默认参数，适配新赛季规则

修复

- 修复在强光下的误检问题 (#42)
- 修复内存泄漏 (#45)

废弃

- `DetectArmor()` 已废弃, 请使用 `Detect()`

[2.0.0] - 2024-01-10

破坏性变更

- 重构检测接口, 不兼容 1.x 版本
- 配置文件格式改为 YAML

...

1.8.9.7. Markdown 写作技巧

Markdown 是最流行的文档格式之一, 几乎所有的代码托管平台都支持它。掌握 Markdown 的写作技巧可以让你的文档更清晰、更易读。

结构化是好文档的基础。使用标题建立层次结构, 让读者能够快速把握文档的脉络。但不要过度嵌套——三级标题通常就足够了, 更深的层次可能意味着文档需要拆分。

一级标题 (文档标题)

二级标题 (主要章节)

三级标题 (子章节)

列表用于并列的内容。有序列表用于有顺序的步骤, 无序列表用于没有顺序的项目。列表项应该结构一致——要么都是完整的句子, 要么都是短语。

安装步骤

1. 克隆仓库
2. 安装依赖
3. 编译项目
4. 运行测试

支持的功能

- 装甲板检测
- 能量机关检测
- 目标跟踪

代码块是技术文档的重要组成部分。始终指定语言以获得语法高亮, 使用行内代码标记命令、函数名、文件名等。

运行 `ros2 launch` 启动节点:

```
bash ros2 launch rm_vision detector.launch.py
```

函数 `Detect()` 返回检测结果。

表格适合展示结构化的对比信息。保持表格简洁, 复杂的内容应该用其他方式呈现。

| 参数 | 类型 | 默认值 | 说明 |
|----------------------|--------|-------|--------|
| confidence_threshold | float | 0.7 | 置信度阈值 |
| nms_threshold | float | 0.4 | NMS 阈值 |
| target_color | string | "red" | 目标颜色 |

图片和图表可以大大提高文档的可理解性。系统架构图、流程图、效果截图都是有价值的数据。使用 Mermaid 可以在 Markdown 中直接编写图表：

```
```mermaid
graph LR
 A[相机] --> B[检测器]
 B --> C[跟踪器]
 C --> D[预测器]
 D --> E[云台控制]
````
```

链接让文档形成网络。使用相对链接引用项目内的其他文档，使用外部链接引用参考资料。但不要过度链接——每个链接都是一个潜在的离开当前上下文的机会。

详细配置说明请参阅 [配置文档]([./config.md](#))。

算法原理基于这篇 [论文](<https://arxiv.org/abs/xxx>)。

1.8.9.8. 文档即代码

“文档即代码”（Docs as Code）是一种现代的文档管理理念：将文档视为代码一样对待，使用相同的工具和流程来管理。

首先，文档应该纳入版本控制。将文档与代码放在同一个仓库中，使用 Git 追踪变更历史。这样可以保证文档与代码的版本对应，也可以回溯历史版本。代码审查流程也应该包括文档——修改代码时，检查相关文档是否需要更新。

其次，文档应该使用纯文本格式。Markdown、reStructuredText、AsciiDoc 等格式易于编辑、易于比较差异、易于自动处理。避免使用 Word 等二进制格式，它们难以进行版本控制和协作编辑。

第三，可以利用 CI/CD 自动化文档流程。代码推送时自动构建文档、检查链接、部署到网站。这样可以保证文档始终是最新的，也可以在问题出现时及早发现。

```
# .github/workflows/docs.yml
name: Build Documentation

on:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build Doxygen
        run: doxygen Doxyfile

      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./docs/api/html
```

最后，鼓励团队成员贡献文档。降低贡献门槛——使用简单的格式、提供模板、及时审查 PR。将文档贡献视为与代码贡献同等重要的工作。

1.8.9.9. RoboMaster 文档实践

结合 RoboMaster 的实际情况，以下是一些具体的文档实践建议。

赛季交接文档是 RoboMaster 团队最重要的文档之一。每年赛季结束后，核心队员应该编写详细的交接文档，包括：系统整体架构和设计思路、各模块的功能和接口、调试经验和已知问题、本赛季的改进和遗留问题、对下赛季的建议。这份文档将成为新队员的入门指南，也是团队知识积累的载体。

2024 赛季自瞄系统交接文档

系统概述

本赛季自瞄系统采用 YOLOv8 + EKF 架构...

取得的成果

- 检测帧率从 30 FPS 提升到 60 FPS
- 命中率从 60% 提升到 80%
- 支持了能量机关检测

遗留问题

1. 强光下仍有误检（建议添加时域滤波）
2. 远距离检测精度不足（建议尝试更大的模型）
3. 代码耦合度高（建议重构跟踪模块）

下赛季建议

- 考虑使用 Transformer 架构
- 完善单元测试覆盖
- 整理代码规范

硬件接口文档对于软硬件协作至关重要。它应该详细描述通信协议、数据格式、引脚定义、时序要求等。当软件组和电控组需要对接时，这份文档可以避免大量的沟通成本。

视觉-电控通信协议

物理接口

- 接口类型：UART
- 波特率：115200
- 数据位：8
- 停止位：1
- 校验：无

数据帧格式

| 字节 | 内容 | 说明 |
|---------|-------|------|
| 0 | 0xA5 | 帧头 |
| 1 | len | 数据长度 |
| 2 | seq | 帧序号 |
| 3 | crc8 | 头部校验 |
| 4-N | data | 数据内容 |
| N+1,N+2 | crc16 | 整帧校验 |

```
## 命令定义

### 0x01: 自瞄数据

```c
struct AimData {
 float yaw; // 目标 yaw 角度, 单位: 度
 float pitch; // 目标 pitch 角度, 单位: 度
 uint8_t fire; // 是否开火: 0-否, 1-是
};

```

```

调试手册记录常见问题的诊断和解决方法。比赛现场时间紧迫, 一份好的调试手册可以帮助快速定位和解决问题。

自瞄系统调试手册

问题: 检测不到目标

可能原因

1. 相机未正确连接
2. 曝光参数不合适
3. 目标颜色设置错误

诊断步骤

1. 检查相机是否被识别: `ls /dev/video*`
2. 查看原始图像: `ros2 run rqt_image_view rqt_image_view`
3. 检查配置文件中的 `target_color` 设置

解决方案

1. 重新插拔相机, 检查 USB 连接
2. 使用 `scripts/auto_exposure.py` 自动调整曝光
3. 修改 `config/detector.yaml` 中的颜色设置

问题: 云台抖动

...

文档不是写完就结束的一次性工作, 而是需要持续维护的活文档。每次代码修改、问题解决、经验总结, 都应该反映到文档中。将文档更新作为开发流程的一部分, 而不是事后的补充工作。团队可以建立这样的规范: 每个 PR 如果涉及接口变更, 必须同时更新相关文档; 每次调试解决的问题, 都要记录到调试手册中; 每个赛季结束, 必须完成交接文档。

文档是软件工程中经常被忽视但极其重要的一环。它是代码与人之间的桥梁, 是知识传承的载体, 是团队协作的基础。花在文档上的时间不是浪费, 而是对未来的投资。当你写文档时, 想象一下几个月后的自己、刚加入团队的新人、或者在比赛现场焦急调试的队友——你写下的每一个字, 都可能在某个时刻帮助到他们。

1.8.10. 版本控制与协作（可选/扩展）

1.9. Git 版本控制

1.10. ROS/ROS2 入门

Euler's identity is $e^{\pi i} + 1 = 0$. Do you really believe that they charged an armed enemy, or treated their children, their own flesh and blood, so cruelly, without a thought for their own interest or advantage? Such is Schrödinger's equation in

$$i\hbar \frac{\partial}{\partial t} \Psi(x, t) = \left[-\frac{\hbar^2}{2m} \frac{\partial^2}{\partial x^2} + V(x, t) \right] \Psi(x, t)$$

But who has any right to find fault with a man who chooses to enjoy a pleasure that has no annoying consequences, or one who avoids a pain that produces no resultant pleasure?

But I must explain to you how all this mistaken idea of reprobating pleasure and extolling pain arose. Increase ease-of-use to where variable and print() shall be of use.

```
if a != b:  
    print("Hello world!")  
else if a == b:  
    print("Goodbye world!")  
else:  
    print("This is a long sentence where I ramble until I get 80 characters here.")
```

代码 1 Example python code printing text.

2. 数学理论篇

2.1. 计算机视觉基础

2.2. 传统视觉算法

2.3. 深度学习和神经网络

2.4. 相机模型

2.5. 坐标变换

2.6. 卡尔曼滤波

Definition 2.6.1

本章使用的符号规定：

- **粗体小写字母** 表示向量，例如 \mathbf{x}
- **粗体大写字母** 表示矩阵，例如 \mathbf{A}
- **普通小写字母** 标量或向量的分量
- **普通大写字母** 表示矩阵元素

2.6.1. 什么是卡尔曼滤波？

2.6.2. 如何准确地测量体重

考虑一个存在随机测量误差的体重秤。单次测量无法得到准确结果，但通过多次测量取平均，可以获得更稳定的估计值。

对同一对象进行 n 次测量，取平均值作为估计：

$$\hat{x}_n = \frac{1}{n} \sum_{i=1}^n z_i$$

然而，直接计算平均值需要存储全部历史数据 z_1, z_2, \dots, z_n ，且每次更新都需遍历所有数据，时间复杂度为 $O(n)$ 。对于资源受限的嵌入式系统，这一方法难以实现。

更理想的方案是递推（Recursive）形式：仅利用上一次估计 \hat{x}_{n-1} 与当前测量 z_n 完成更新，无需保存历史数据。

通过代数变换，平均值公式可改写为：

$$\hat{x}_n = \hat{x}_{n-1} + \frac{1}{n}(z_n - \hat{x}_{n-1})$$

上式表明，第 n 次估计仅依赖 \hat{x}_{n-1} 与 z_n ，时间和空间复杂度均为 $O(1)$ 。

$$\begin{aligned}
\hat{x}_n &= \frac{1}{n} \sum_{i=1}^n z_i = \frac{1}{n} \left(\sum_{i=1}^{n-1} z_i + z_n \right) \\
&= \frac{1}{n} \sum_{i=1}^{n-1} z_i + \frac{1}{n} z_n = \frac{n-1}{n(n-1)} \sum_{i=1}^{n-1} z_i + \frac{1}{n} z_n \\
&= \frac{n-1}{n} \cdot \frac{1}{n-1} \sum_{i=1}^{n-1} z_i + \frac{1}{n} z_n = \frac{n-1}{n} \hat{x}_{n-1} + \frac{1}{n} z_n \\
&= \hat{x}_{n-1} - \frac{1}{n} \hat{x}_{n-1} + \frac{1}{n} z_n \\
&= \hat{x}_{n-1} + \frac{1}{n} (z_n - \hat{x}_{n-1})
\end{aligned}$$

| 符号 | 含义 |
|-----------------------|-------------------|
| x | 体重真值 |
| z_n | 第 n 次测量值 |
| \hat{x}_n | 基于前 n 次测量的估计值 |
| \hat{x}_{n-1} | 基于前 $n-1$ 次测量的估计值 |
| $z_n - \hat{x}_{n-1}$ | 测量残差 |

note 2 详细推导

上述递推公式已具备状态估计滤波器的基本形式。其中增益系数 $K_n = \frac{1}{n}$ 决定了测量值与历史估计的融合权重。随着 n 增加， K_n 递减——可用数据越多，单次新测量的影响越小。

这一结构与卡尔曼滤波的状态更新公式形式一致：

$$\hat{x}_k = \hat{x}_{k|k-1} + K_k (z_k - h(\hat{x}_{k|k-1}))$$

其中：

- $\hat{x}_{k|k-1}$: 基于前 $k-1$ 次观测的先验估计
- z_k : 第 k 次测量值
- $h(\cdot)$: 观测函数
- K_k : 卡尔曼增益

在体重秤场景中，状态不随时间变化 ($\hat{x}_{k|k-1} = \hat{x}_{k-1}$)，观测函数为恒等映射 ($h(x) = x$)，因此递推平均可视为卡尔曼滤波在静态系统下的特例。

将递推公式标注为：

$$\hat{x}_n = \underbrace{\hat{x}_{n-1}}_{\text{先验估计}} + \underbrace{\frac{1}{n} (z_n - \hat{x}_{n-1})}_{\substack{\text{增益} \\ \text{残差}}}$$

三个组成部分的含义：先验估计 \hat{x}_{n-1} 为获得新测量前的当前认知；测量残差 $r_n = z_n - \hat{x}_{n-1}$ 为观测值与先验的偏差；增益系数 K_n 控制残差对估计的修正幅度。这一“先验 + 增益 × 残差”的结构是递推估计的通用框架。

2.6.3. 从体重秤到赛车：静态估计的局限性

递推平均能够有效估计静态量，是因为体重在测量过程中保持不变，所有偏差均可归因于传感器噪声。但对于动态系统，情况有所不同。

考虑追踪赛道上的赛车：起始位置 0 米，以 20 m/s 匀速行驶。GPS 每秒测量一次位置，误差 ± 5 米。

真实轨迹为 $p_t = 20t$ ，GPS 测量值 $z_t = p_t + \text{噪声}$ 。

| 时刻 (s) | 真实位置 (m) | GPS 测量 (m) | 递推平均 (m) | 误差 (m) |
|--------|----------|------------|----------|--------|
| 0 | 0 | 1.2 | 1.2 | +1.2 |
| 1 | 20 | 18.3 | 9.8 | -10.2 |
| 2 | 40 | 43.1 | 20.9 | -19.1 |
| 3 | 60 | 56.8 | 29.9 | -30.1 |
| 4 | 80 | 82.4 | 40.4 | -39.6 |
| 5 | 100 | 97.6 | 49.9 | -50.1 |

表 1 递推平均追踪动态目标的失效

估计误差随时间单调增大。问题在于递推平均的假设：被估计量保持不变。该公式将测量值的所有变化都视为噪声，无法区分噪声与真实的位置变化。

本质问题是：**递推平均只有修正，没有预测**。正确的思路应在获得新测量前，先根据运动规律预测当前位置，再用测量值修正。这正是 g-h 滤波器的核心思想。

2.6.3.1. g-h 滤波器

对于运动物体，估计应包含预测步骤：

$$\text{估计} = \text{预测 (运动模型)} + \text{修正 (测量)}$$

设在第 $k-1$ 时刻已有位置估计 \hat{p}_{k-1} 与速度估计 \hat{v}_{k-1} 。根据匀速运动模型，第 k 时刻的预测为：

$$p_{\text{pred},k} = \hat{p}_{k-1} + \hat{v}_{k-1} \cdot \Delta t$$

$$v_{\text{pred},k} = \hat{v}_{k-1}$$

其中 Δt 为采样间隔。

若 $\hat{p}_0 = 0$ m, $\hat{v}_0 = 15$ m/s, $\Delta t = 1$ s，则 $p_{\text{pred},1} = 15$ m。

当测量值 z_k 到来时，计算残差：

$$r_k = z_k - p_{\text{pred},k}$$

位置修正

$$\hat{p}_k = p_{\text{pred},k} + g \cdot r_k$$

其中 $g \in (0, 1)$ 为位置增益。 $g = 0$ 表示完全信任预测， $g = 1$ 表示完全信任测量。

速度修正

$$\hat{v}_k = v_{\text{pred},k} + \frac{h}{\Delta t} \cdot r_k$$

其中 $h \in (0, 1)$ 为速度增益。除以 Δt 是为了将位置残差（单位：m）转换为速度修正量（单位：m/s）。

以 $g = 0.3$, $h = 0.1$, $z_1 = 18.3$ m 为例: $r_1 = 3.3$ m, $\hat{p}_1 = 15.99$ m, $\hat{v}_1 = 15.33$ m/s。

g-h 滤波器算法

输入: \hat{p}_{k-1} , \hat{v}_{k-1} , z_k , Δt , g , h

预测:

$$p_{\text{pred},k} = \hat{p}_{k-1} + \hat{v}_{k-1} \Delta t, \quad v_{\text{pred},k} = \hat{v}_{k-1}$$

修正:

$$r_k = z_k - p_{\text{pred},k}$$

$$\hat{p}_k = p_{\text{pred},k} + gr_k, \quad \hat{v}_k = v_{\text{pred},k} + \frac{h}{\Delta t} r_k$$

输出: \hat{p}_k , \hat{v}_k

数值示例

参数: 真实速度 20 m/s, $\Delta t = 1$ s, $g = 0.3$, $h = 0.1$, 初始估计 $\hat{p}_0 = 0$, $\hat{v}_0 = 15$ (故意设为错误值)。

| T (s) | 真实位置 | GPS | G-H 估计 | 真实速度 | 速度估计 |
|-------|------|------|--------|------|-------|
| 0 | 0 | — | 0.0 | 20 | 15.00 |
| 1 | 20 | 18.3 | 16.0 | 20 | 15.33 |
| 2 | 40 | 43.1 | 34.9 | 20 | 16.51 |
| 3 | 60 | 56.8 | 54.3 | 20 | 17.68 |
| 4 | 80 | 82.4 | 74.8 | 20 | 18.73 |
| 5 | 100 | 97.6 | 96.0 | 20 | 19.60 |

表 2 g-h 滤波器追踪结果

位置误差逐步收敛，速度估计自动逼近真实值，初始误差通过迭代修正被消除。

递推平均与 g-h 滤波器对比

| T (s) | 真实位置 (m) | 递推平均 (m) | G-H (m) |
|-------|----------|----------|---------|
| 0 | 0 | 1.2 | 0.0 |
| 1 | 20 | 9.8 | 16.0 |
| 2 | 40 | 20.9 | 34.9 |
| 3 | 60 | 29.9 | 54.3 |
| 4 | 80 | 40.4 | 74.8 |

| | | | |
|---|-----|------|------|
| 5 | 100 | 49.9 | 96.0 |
|---|-----|------|------|

表 3 同一数据下两种方法的对比

| | 递推平均 | G-H 滤波器 |
|---------|--------------------|-------------|
| 适用场景 | 静态量 | 动态系统 |
| 预测机制 | 无 | 有（运动模型） |
| 状态数 | 1 | 2（位置+速度） |
| 增益 | $\frac{1}{n}$ （递减） | g, h （固定） |
| 第 5 秒误差 | 50.1 m | 4.0 m |

2.6.3.2. 从标量到矩阵：为高维系统做准备

标量公式在一维场景下足够清晰，但对于高维系统会变得繁琐：

- 一维运动： (p, v) – 2 个状态
- 二维平面： (x, y, v_x, v_y) – 4 个状态
- 三维空间 + 加速度：9 个状态

标量形式需要为每个状态单独写方程，而这些方程结构相同。矩阵形式可将其统一：无论状态维数如何，公式形式保持不变。

状态向量

定义状态向量 $\mathbf{x}_k = \begin{pmatrix} p_k \\ v_k \end{pmatrix}$ 与增益向量 $\mathbf{K} = \begin{pmatrix} g \\ h/\Delta t \end{pmatrix}$ 。两个标量修正公式可合并为：

$$\hat{\mathbf{x}}_k = \mathbf{x}_{\text{pred}} + \mathbf{K} \cdot (z_k - p_{\text{pred}})$$

增益向量的每一行对应一个状态的修正规则。

观测矩阵

GPS 仅测量位置，无法测量速度。需要从状态向量中提取可观测分量，这由观测矩阵 \mathbf{H} 完成：

$$\mathbf{H} = (1 \ 0)$$

其作用为：

$$\mathbf{H}\mathbf{x}_{\text{pred}} = (1 \ 0) \begin{pmatrix} p_{\text{pred}} \\ v_{\text{pred}} \end{pmatrix} = p_{\text{pred}}$$

对于不同传感器配置， \mathbf{H} 的形式不同：

- 仅测位置： $\mathbf{H} = (1 \ 0)$
- 仅测加速度： $\mathbf{H} = (0 \ 0 \ 1)$
- 同时测位置和速度： $\mathbf{H} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

完整的矩阵形式

$$\hat{\mathbf{x}}_k = \mathbf{x}_{\text{pred}} + \mathbf{K}(z_k - \mathbf{H}\mathbf{x}_{\text{pred}})$$

各项含义： \mathbf{x}_{pred} 为模型预测； $\mathbf{H}\mathbf{x}_{\text{pred}}$ 为预测对应的测量值； $z_k - \mathbf{H}\mathbf{x}_{\text{pred}}$ 为测量残差； \mathbf{K} 控制修正幅度。

验证

展开矩阵公式：

$$\begin{aligned}\begin{pmatrix} \hat{p}_k \\ \hat{v}_k \end{pmatrix} &= \begin{pmatrix} p_{\text{pred}} \\ v_{\text{pred}} \end{pmatrix} + \begin{pmatrix} g \\ h/\Delta t \end{pmatrix} \left(z_k - (1 \ 0) \begin{pmatrix} p_{\text{pred}} \\ v_{\text{pred}} \end{pmatrix} \right) \\ &= \begin{pmatrix} p_{\text{pred}} + g(z_k - p_{\text{pred}}) \\ v_{\text{pred}} + (h/\Delta t)(z_k - p_{\text{pred}}) \end{pmatrix}\end{aligned}$$

与标量形式一致。

预测步骤的矩阵形式

定义状态转移矩阵：

$$F = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix}$$

预测步骤为 $x_{\text{pred}} = F \hat{x}_{k-1}$, 展开后：

$$\begin{pmatrix} p_{\text{pred}} \\ v_{\text{pred}} \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{p}_{k-1} \\ \hat{v}_{k-1} \end{pmatrix} = \begin{pmatrix} \hat{p}_{k-1} + \Delta t \hat{v}_{k-1} \\ \hat{v}_{k-1} \end{pmatrix}$$

2.6.3.3. 完整算法（矩阵形式）

定义：

$$x_k = \begin{pmatrix} p_k \\ v_k \end{pmatrix}, \quad F = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix}, \quad H = (1 \ 0), \quad K = \begin{pmatrix} g \\ h/\Delta t \end{pmatrix}$$

预测： $x_{\text{pred}} = F \hat{x}_{k-1}$

修正： $\hat{x}_k = x_{\text{pred}} + K(z_k - H x_{\text{pred}})$

2.6.3.4. g-h 滤波器的局限性

g-h 滤波器具备“预测 + 修正”的基本结构，但增益 g 、 h 需手动设定。卡尔曼滤波通过以下机制解决这一问题：

1. 引入误差协方差矩阵 P_k 量化估计的不确定性
2. 根据最小化估计误差的准则，动态计算最优增益 K_k

下一章将推导这一机制的数学原理。

2.6.4. 最优增益的推导

上一章建立了 g-h 滤波器的基本框架，其核心公式为：

$$\hat{x}_k = x_{\text{pred}} + K(z_k - x_{\text{pred}})$$

该公式的结构清晰：预测值与测量值的加权融合，增益 K 控制两者的权重分配。然而， K 的取值从何而来？在 g-h 滤波器中， g 和 h 由工程师根据经验手动设定，这显然不是最优解。本章将从概率的角度重新审视状态估计问题，推导出增益的最优取值，这正是卡尔曼滤波的核心贡献。

2.6.4.1. 不确定性的量化

在讨论“最优”之前，需要明确优化的目标。回到赛车追踪的场景：预测位置为 35 米，GPS 测量值为 38 米，最终估计应该取多少？

g-h 滤波器的回答是：取决于你设定的 g 值。若 $g = 0.3$ ，则估计为 $35 + 0.3 \times (38 - 35) = 35.9$ 米。但这个答案回避了一个关键问题——凭什么选 $g = 0.3$ ？

更合理的思路是考虑预测和测量各自的可信程度。如果运动模型非常精确（比如赛车在笔直赛道上匀速行驶），预测值应当更可信；如果 GPS 精度很高而模型存在较大不确定性，测量值应当更可信。这种“可信程度”需要用数学语言精确描述，而方差正是刻画不确定性的自然工具。

设预测值 x_{pred} 的误差服从均值为零、方差为 σ_{pred}^2 的分布，测量值 z 的误差服从均值为零、方差为 σ_z^2 的分布。方差越大，对应的值越不可信。现在，优化目标可以明确表述为：选择增益 K ，使得融合后的估计 \hat{x} 具有最小的误差方差。

2.6.4.2. 两个估计的最优融合

为简化推导，先考虑一个更基本的问题：有两个对同一量 x 的独立估计 x_1 和 x_2 ，误差方差分别为 σ_1^2 和 σ_2^2 ，如何将它们融合为一个更精确的估计？

自然的想法是加权平均：

$$\hat{x} = w_1 x_1 + w_2 x_2, \quad w_1 + w_2 = 1$$

设 $w_1 = 1 - K$, $w_2 = K$, 则：

$$\hat{x} = (1 - K)x_1 + Kx_2 = x_1 + K(x_2 - x_1)$$

这与滤波器的修正公式形式完全一致。接下来推导使 \hat{x} 方差最小的 K 值。

设 $x_1 = x + \varepsilon_1$, $x_2 = x + \varepsilon_2$, 其中 ε_1 、 ε_2 为零均值、相互独立的误差项，方差分别为 σ_1^2 、 σ_2^2 。融合估计的误差为：

$$\hat{x} - x = (1 - K)(x + \varepsilon_1) + K(x + \varepsilon_2) - x = (1 - K)\varepsilon_1 + K\varepsilon_2$$

由于 ε_1 与 ε_2 独立，误差方差为：

$$\sigma_{\hat{x}}^2 = (1 - K)^2 \sigma_1^2 + K^2 \sigma_2^2$$

对 K 求导并令其为零：

$$\frac{d\sigma_{\hat{x}}^2}{dK} = -2(1 - K)\sigma_1^2 + 2K\sigma_2^2 = 0$$

解得：

$$K^* = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_2^2}$$

这一结果具有直观的解释：最优增益与 x_1 的方差成正比，与 x_2 的方差成反比。换言之，越不可信的估计，其权重越低。当 $\sigma_1^2 \gg \sigma_2^2$ 时， $K^* \approx 1$ ，融合结果接近 x_2 ；当 $\sigma_1^2 \ll \sigma_2^2$ 时， $K^* \approx 0$ ，融合结果接近 x_1 。

将最优增益代回方差公式，可得融合后的方差：

$$\sigma_{\hat{x}}^2 = (1 - K^*)\sigma_1^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

注意到 $\sigma_{\hat{x}}^2 < \min(\sigma_1^2, \sigma_2^2)$, 即融合后的估计总是比任一单独估计更精确。这正是信息融合的价值所在。

2.6.4.3. 卡尔曼增益的形式

回到滤波器的语境, x_1 对应预测值 x_{pred} , x_2 对应测量值 z 。将符号替换后, 最优增益为:

$$K = \frac{\sigma_{\text{pred}}^2}{\sigma_{\text{pred}}^2 + \sigma_z^2}$$

这就是一维卡尔曼增益的表达式。其行为与直觉一致:

- 若预测不确定性大 (σ_{pred}^2 大), K 接近 1, 更信任测量
- 若测量噪声大 (σ_z^2 大), K 接近 0, 更信任预测
- 两者相当时, $K \approx 0.5$, 各取一半

与 g-h 滤波器的固定增益不同, 卡尔曼增益随不确定性的变化而动态调整。在滤波器运行初期, 状态估计的不确定性通常较大, 此时 K 较大, 测量值对估计的影响显著; 随着观测数据的积累, 估计逐渐收敛, σ_{pred}^2 减小, K 随之下降, 新测量的影响减弱。这种自适应特性是卡尔曼滤波的核心优势。

2.6.4.4. 不确定性的传播

卡尔曼增益的计算依赖于 σ_{pred}^2 , 而 σ_{pred}^2 又从何而来? 这需要追踪不确定性在预测和修正两个阶段的演变。

预测阶段的不确定性增长

考虑匀速运动模型 $x_k = x_{k-1} + v\Delta t$ 。即使上一时刻的位置估计是准确的, 模型本身也存在误差——赛车可能轻微加速或减速, 这种模型不确定性称为过程噪声, 记其方差为 σ_w^2 。

设第 $k-1$ 时刻的估计方差为 σ_{k-1}^2 。由于位置预测直接继承了上一时刻的估计 ($x_{\text{pred}} = \hat{x}_{k-1} + \hat{v}\Delta t$), 预测误差包含两部分: 上一时刻的估计误差和过程噪声。假设两者独立, 预测方差为:

$$\sigma_{\text{pred}}^2 = \sigma_{k-1}^2 + \sigma_w^2$$

这一公式说明, 预测阶段的不确定性总是增加——即使此前的估计很精确, 模型误差也会使不确定性累积。

修正阶段的不确定性减小

当新测量到来后, 融合公式降低了不确定性。由前面的推导, 修正后的方差为:

$$\sigma_k^2 = (1 - K)\sigma_{\text{pred}}^2 = \frac{\sigma_{\text{pred}}^2 \sigma_z^2}{\sigma_{\text{pred}}^2 + \sigma_z^2}$$

由于 $0 < K < 1$, 修正后的方差总是小于预测方差。每次测量都带来新信息, 使估计更加确定。

将增益公式代入, 修正后的方差也可写为:

$$\sigma_k^2 = (1 - K)\sigma_{\text{pred}}^2$$

这一形式在计算上更为简洁, 将在后续的多维推广中使用。

2.6.4.5. 一维卡尔曼滤波算法

综合以上推导，一维卡尔曼滤波的完整算法如下。设系统状态为位置 x ，状态转移方程为 $x_k = x_{k-1} + u_k + w_k$ ，其中 u_k 为已知控制量（如速度乘以时间）， w_k 为过程噪声。测量方程为 $z_k = x_k + v_k$ ，其中 v_k 为测量噪声。

初始化

设定初始估计 \hat{x}_0 和初始方差 σ_0^2 。若对初始状态一无所知，可将 σ_0^2 设为较大值。

预测

$$x_{\text{pred}} = \hat{x}_{k-1} + u_k$$

$$\sigma_{\text{pred}}^2 = \sigma_{k-1}^2 + \sigma_w^2$$

修正

$$K_k = \frac{\sigma_{\text{pred}}^2}{\sigma_{\text{pred}}^2 + \sigma_z^2}$$

$$\hat{x}_k = x_{\text{pred}} + K_k(z_k - x_{\text{pred}})$$

$$\sigma_k^2 = (1 - K_k)\sigma_{\text{pred}}^2$$

算法在预测与修正之间交替进行，每一步都输出状态估计 \hat{x}_k 及其不确定性 σ_k^2 。与 g-h 滤波器相比，核心区别在于增益 K_k 不再是预设常数，而是根据当前不确定性动态计算。

2.6.4.6. 数值示例：重新追踪赛车

沿用上一章的赛车场景，但改用卡尔曼滤波进行估计。参数设定如下：

- 过程噪声标准差： $\sigma_w = 1 \text{ m}$ （模型不完全准确）
- 测量噪声标准差： $\sigma_z = 5 \text{ m}$ （GPS 精度有限）
- 初始估计： $\hat{x}_0 = 0 \text{ m}$
- 初始标准差： $\sigma_0 = 10 \text{ m}$ （对初始位置不确定）
- 控制量： $u_k = 20 \text{ m}$ （真实速度 20 m/s ， $\Delta t = 1 \text{ s}$ ）

| T (s) | 真实 | 预测 | σ_{pred} | 测量 | K | 估计 |
|-------|-----|-------|------------------------|------|------|-------|
| 0 | 0 | — | — | — | — | 0.0 |
| 1 | 20 | 20.0 | 10.0 | 18.3 | 0.80 | 18.6 |
| 2 | 40 | 38.6 | 4.5 | 43.1 | 0.45 | 40.6 |
| 3 | 60 | 60.6 | 2.1 | 56.8 | 0.15 | 60.0 |
| 4 | 80 | 80.0 | 1.5 | 82.4 | 0.08 | 80.2 |
| 5 | 100 | 100.2 | 1.3 | 97.6 | 0.06 | 100.0 |

表 4 一维卡尔曼滤波追踪赛车

预测: $x_{\text{pred}} = 0 + 20 = 20 \text{ m}$

预测方差: $\sigma_{\text{pred}}^2 = 10^2 + 1^2 = 101, \sigma_{\text{pred}} \approx 10.0 \text{ m}$

卡尔曼增益: $K = \frac{101}{101+25} = 0.80$

修正: $\hat{x}_1 = 20 + 0.80 \times (18.3 - 20) = 18.6 \text{ m}$

修正方差: $\sigma_1^2 = (1 - 0.80) \times 101 = 20.2, \sigma_1 \approx 4.5 \text{ m}$

note 3 第1步计算过程

从表中可以观察到几个重要现象。首先，卡尔曼增益 K 从初始的 0.80 逐步下降到 0.06。这是因为随着观测数据的积累，估计的不确定性 σ_{pred} 不断减小，滤波器对新测量的依赖程度降低。其次，预测标准差从 10 m 迅速收敛到约 1.3 m，说明滤波器在几步之内就建立了对状态的可靠估计。最后，第 5 秒的估计误差仅为 0.0 m，远优于 g-h 滤波器的 4.0 m 和递推平均的 50.1 m。

这一改进的根源在于：卡尔曼滤波根据实际的不确定性水平自动调整增益，而非使用固定权重。当估计不确定时（初始阶段），大量采纳测量信息；当估计已经可靠时，对异常测量保持适度怀疑。

2.6.4.7. 从 g-h 滤波器到卡尔曼滤波

至此，可以清晰地看出 g-h 滤波器与卡尔曼滤波的关系。两者共享相同的修正公式：

$$\hat{x}_k = x_{\text{pred}} + K(z_k - x_{\text{pred}})$$

差异在于增益的来源：

| | G-H 滤波器 | 卡尔曼滤波 |
|------|------------|---|
| 增益 | 手动设定常数 g | 动态计算 $K_k = \frac{\sigma_{\text{pred}}^2}{\sigma_{\text{pred}}^2 + \sigma_z^2}$ |
| 依据 | 工程经验 | 最小化估计误差方差 |
| 自适应 | 无 | 有（增益随不确定性变化） |
| 额外输出 | 无 | 估计的不确定性 σ_k^2 |

卡尔曼滤波可视为 g-h 滤波器的最优化版本：在相同的“预测-修正”框架下，通过概率推理自动确定最优增益。此外，卡尔曼滤波还输出估计的不确定性，这在实际应用中极为重要——它告诉我们估计结果有多可信。

2.6.4.8. 本章小结

本章从优化的角度重新审视了状态估计问题，得到以下核心结论：

状态估计的目标是最小化估计误差的方差。当预测和测量各自携带不确定性时，最优融合策略是根据方差进行加权：方差越大的估计，权重越低。由此导出的卡尔曼增益 $K = \frac{\sigma_{\text{pred}}^2}{\sigma_{\text{pred}}^2 + \sigma_z^2}$ 具有清晰的物理意义——它平衡了模型预测与传感器测量之间的信任程度。

不确定性在预测阶段增长（因为模型不完美），在修正阶段减小（因为测量提供新信息）。卡尔曼滤波的递推结构正是在这两个过程之间交替，持续追踪状态估计及其不确定性。

目前的推导限于一维情形。下一章将把这些概念推广到多维系统，届时方差将被协方差矩阵取代，增益将成为矩阵，但核心思想——基于不确定性的最优融合——保持不变。

2.6.5. 多维卡尔曼滤波

上一章在一维情形下完成了卡尔曼滤波的推导，核心结论是：最优增益由预测方差与测量方差的比值决定，融合后的估计具有最小的误差方差。然而，实际系统往往涉及多个相互关联的状态量。以赛车追踪为例，完整描述其运动状态至少需要位置和速度两个变量；若在二维平面上追踪，则需要 (x, y, v_x, v_y) 四个状态。本章将一维结论推广到多维情形，建立完整的卡尔曼滤波方程组。

2.6.5.1. 为什么需要协方差矩阵

在一维情形下，估计的不确定性用单个方差 σ^2 描述。推广到多维时，一个自然的想法是为每个状态分量分别记录方差。例如，对于状态向量 $\mathbf{x} = \begin{pmatrix} p \\ v \end{pmatrix}$ ，分别记录位置方差 σ_p^2 和速度方差 σ_v^2 。

这种做法忽略了一个重要现象：状态分量之间往往存在相关性。考虑以下场景：若当前位置被高估（真实位置比估计值低），根据 $p = p_0 + vt$ 的关系，很可能速度也被高估了。换言之，位置误差和速度误差并非独立，而是倾向于同向偏离。

协方差正是刻画这种相关性的工具。对于两个随机变量 X 和 Y ，协方差定义为：

$$\text{Cov}(X, Y) = E[(X - \mu_X)(Y - \mu_Y)]$$

若 X 和 Y 倾向于同向偏离各自的均值，协方差为正；若倾向于反向偏离，协方差为负；若两者独立，协方差为零。方差是协方差的特例： $\text{Var}(X) = \text{Cov}(X, X)$ 。

对于 n 维状态向量 $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ ，所有方差和协方差可组织成一个 $n \times n$ 的对称矩阵：

$$\mathbf{P} = \begin{pmatrix} \text{Var}(x_1) & \text{Cov}(x_1, x_2) & \cdots & \text{Cov}(x_1, x_n) \\ \text{Cov}(x_2, x_1) & \text{Var}(x_2) & \cdots & \text{Cov}(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Cov}(x_n, x_1) & \text{Cov}(x_n, x_2) & \cdots & \text{Var}(x_n) \end{pmatrix}$$

这就是协方差矩阵。对角线元素是各状态的方差，非对角线元素是状态之间的协方差。以位置-速度系统为例：

$$\mathbf{P} = \begin{pmatrix} \sigma_p^2 & \sigma_{pv} \\ \sigma_{pv} & \sigma_v^2 \end{pmatrix}$$

其中 $\sigma_{pv} = \text{Cov}(p, v)$ 描述位置误差与速度误差的相关程度。若 $\sigma_{pv} > 0$ ，表示位置被高估时速度也倾向于被高估。

协方差矩阵在卡尔曼滤波中扮演核心角色：它不仅记录各状态的不确定性大小，还记录状态之间的误差关联。后者对于提高估计精度至关重要——当测量修正了某个状态时，与之相关的其他状态也应当相应调整。

2.6.5.2. 多维系统的数学描述

在建立滤波方程之前，需要明确多维系统的数学模型。

状态转移方程

系统状态从第 $k - 1$ 时刻演变到第 k 时刻的规律由状态转移方程描述：

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B}\mathbf{u}_k + \mathbf{w}_k$$

其中 \mathbf{F} 是状态转移矩阵，描述状态的自然演化； \mathbf{B} 是控制矩阵， \mathbf{u}_k 是控制输入； \mathbf{w}_k 是过程噪声，代表模型的不确定性。

以匀速运动模型为例，状态为 $\mathbf{x} = \begin{pmatrix} p \\ v \end{pmatrix}$ ，状态转移方程为：

$$\begin{pmatrix} p_k \\ v_k \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p_{k-1} \\ v_{k-1} \end{pmatrix} + \begin{pmatrix} w_p \\ w_v \end{pmatrix}$$

状态转移矩阵 $\mathbf{F} = \begin{pmatrix} 1 & \Delta t \\ 0 & 1 \end{pmatrix}$ 的含义是：新位置 = 旧位置 + 速度 \times 时间，新速度 = 旧速度（匀速假设）。

过程噪声 \mathbf{w}_k 通常假设服从零均值高斯分布，其协方差矩阵记为 \mathbf{Q} ：

$$\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$$

\mathbf{Q} 的大小反映模型的可靠程度。若系统严格遵循模型（如匀速运动）， \mathbf{Q} 较小；若模型仅为粗略近似， \mathbf{Q} 较大。

观测方程

传感器测量与系统状态的关系由观测方程描述：

$$\mathbf{z}_k = \mathbf{H}\mathbf{x}_k + \mathbf{v}_k$$

其中 \mathbf{H} 是观测矩阵，将状态空间映射到测量空间； \mathbf{v}_k 是测量噪声，协方差矩阵记为 \mathbf{R} 。

继续位置-速度的例子。若传感器仅能测量位置，观测方程为：

$$z_k = (1 \ 0) \begin{pmatrix} p_k \\ v_k \end{pmatrix} + v_k$$

观测矩阵 $\mathbf{H} = (1 \ 0)$ 表示从状态向量中提取位置分量。测量噪声方差 $\mathbf{R} = \sigma_z^2$ 反映传感器精度。

2.6.5.3. 预测阶段

卡尔曼滤波的每一步分为预测和修正两个阶段。预测阶段根据系统模型推算下一时刻的状态及其不确定性。

状态预测

状态预测直接应用状态转移方程（忽略噪声项，因为噪声均值为零）：

$$\mathbf{x}_{k|k-1} = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B}\mathbf{u}_k$$

下标 $k|k-1$ 表示“基于前 $k-1$ 时刻信息对第 k 时刻的预测”。

协方差预测

不确定性的传播稍显复杂。设第 $k-1$ 时刻的估计协方差为 \mathbf{P}_{k-1} ，需要推导预测协方差 $\mathbf{P}_{k|k-1}$ 。

状态预测误差为：

$$\mathbf{x}_k - \mathbf{x}_{k|k-1} = \mathbf{F}(\mathbf{x}_{k-1} - \hat{\mathbf{x}}_{k-1}) + \mathbf{w}_k$$

等号右边第一项是上一时刻估计误差经 \mathbf{F} 变换的结果，第二项是过程噪声。由于估计误差与过程噪声独立，预测协方差为：

$$\mathbf{P}_{k|k-1} = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^\top + \mathbf{Q}$$

这一公式的含义是：预测不确定性来自两部分——上一时刻的估计不确定性（经状态转移放大或缩小）和模型本身的不确定性（过程噪声）。即使 \mathbf{P}_{k-1} 很小， \mathbf{Q} 的存在也会使 $\mathbf{P}_{k|k-1}$ 增大，这与一维情形中“预测阶段方差增加”的结论一致。

协方差变换公式

Note 2.6.5.3.1

若随机向量 \mathbf{x} 的协方差矩阵为 \mathbf{P} ，则线性变换 $\mathbf{y} = \mathbf{Ax}$ 的协方差矩阵为 $\mathbf{A}\mathbf{P}\mathbf{A}^\top$ 。

证明：设 \mathbf{x} 的均值为 $\boldsymbol{\mu}$ ，则

$$\text{Cov}(\mathbf{y}) = E[(\mathbf{Ax} - \mathbf{A}\boldsymbol{\mu})(\mathbf{Ax} - \mathbf{A}\boldsymbol{\mu})^\top] = \mathbf{A}E[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top]\mathbf{A}^\top = \mathbf{A}\mathbf{P}\mathbf{A}^\top$$

note 4 协方差变换公式

2.6.5.4. 修正阶段

当新测量 \mathbf{z}_k 到来时，修正阶段将预测与测量融合，得到更精确的估计。

卡尔曼增益

多维卡尔曼增益的推导遵循与一维相同的原则：最小化修正后的估计误差协方差。推导过程涉及矩阵求导，此处直接给出结果：

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}^\top (\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^\top + \mathbf{R})^{-1}$$

这一公式的结构与一维情形对应。分子 $\mathbf{P}_{k|k-1}\mathbf{H}^\top$ 反映预测不确定性在测量空间的投影；分母 $\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^\top + \mathbf{R}$ 是预测测量的不确定性（来自状态预测）与测量本身的不确定性（来自传感器）之和。当预测不确定性大时， \mathbf{K} 较大，更信任测量；当测量不确定性大时， \mathbf{K} 较小，更信任预测。

状态修正

修正公式与 g-h 滤波器形式相同：

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k|k-1} + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}\mathbf{x}_{k|k-1})$$

括号内的 $\mathbf{z}_k - \mathbf{H}\mathbf{x}_{k|k-1}$ 称为测量残差或新息（Innovation），表示实际测量与预测测量之间的偏差。卡尔曼增益决定了这一偏差如何分配到各个状态分量上。

协方差修正

修正后的协方差为：

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_{k|k-1}$$

其中 \mathbf{I} 为单位矩阵。这一公式表明，测量总是减小不确定性—— $\mathbf{I} - \mathbf{K}_k\mathbf{H}$ 的特征值均小于 1，因此 \mathbf{P}_k 在各方向上都不大于 $\mathbf{P}_{k|k-1}$ 。

2.6.5.5. 完整算法

综合预测和修正阶段，多维卡尔曼滤波的完整算法如下。

系统模型

状态转移： $\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B}\mathbf{u}_k + \mathbf{w}_k$, $\mathbf{w}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$

观测方程: $\mathbf{z}_k = \mathbf{H}\mathbf{x}_k + \mathbf{v}_k$, $\mathbf{v}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$

初始化

初始状态估计 $\hat{\mathbf{x}}_0$, 初始协方差 \mathbf{P}_0

预测

$$\mathbf{x}_{k|k-1} = \mathbf{F}\hat{\mathbf{x}}_{k-1} + \mathbf{B}\mathbf{u}_k$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^\top + \mathbf{Q}$$

修正

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}^\top (\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^\top + \mathbf{R})^{-1}$$

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k|k-1} + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}\mathbf{x}_{k|k-1})$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_{k|k-1}$$

算法在预测与修正之间交替进行, 每一步输出状态估计 $\hat{\mathbf{x}}_k$ 及其协方差 \mathbf{P}_k 。协方差矩阵不仅用于计算下一步的卡尔曼增益, 还直接提供了估计的置信区间。

2.6.5.6. 示例：一维位置-速度系统

为了具体展示多维卡尔曼滤波的运算过程, 回到赛车追踪问题。此前我们假设速度已知(作为控制输入), 现在将速度也作为待估计的状态, 考察滤波器能否从位置测量中同时恢复位置和速度。

系统设定

状态向量 $\mathbf{x} = \begin{pmatrix} p \\ v \end{pmatrix}$, 真实初始状态为 $\begin{pmatrix} 0 \\ 20 \end{pmatrix}$ (位置 0 m, 速度 20 m/s)。

状态转移矩阵 (匀速模型, $\Delta t = 1$ s):

$$\mathbf{F} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

观测矩阵 (仅测量位置):

$$\mathbf{H} = (1 \ 0)$$

过程噪声协方差 (速度可能有小幅波动):

$$\mathbf{Q} = \begin{pmatrix} 0.25 & 0.5 \\ 0.5 & 1 \end{pmatrix}$$

测量噪声方差:

$$\mathbf{R} = (25) \quad (\sigma_z = 5\text{m})$$

初始估计 (故意设定错误的速度):

$$\hat{\mathbf{x}}_0 = \begin{pmatrix} 0 \\ 15 \end{pmatrix}, \quad \mathbf{P}_0 = \begin{pmatrix} 100 & 0 \\ 0 & 100 \end{pmatrix}$$

第 1 步迭代

预测:

$$\mathbf{x}_{1|0} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 15 \end{pmatrix} = \begin{pmatrix} 15 \\ 15 \end{pmatrix}$$

$$\mathbf{P}_{1|0} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 100 & 0 \\ 0 & 100 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} + \begin{pmatrix} 0.25 & 0.5 \\ 0.5 & 1 \end{pmatrix} = \begin{pmatrix} 200.25 & 100.5 \\ 100.5 & 101 \end{pmatrix}$$

卡尔曼增益：

$$\mathbf{H}\mathbf{P}_{1|0}\mathbf{H}^\top + \mathbf{R} = 200.25 + 25 = 225.25$$

$$\mathbf{K}_1 = \begin{pmatrix} 200.25 \\ 100.5 \end{pmatrix} \times \frac{1}{225.25} = \begin{pmatrix} 0.889 \\ 0.446 \end{pmatrix}$$

测量 $z_1 = 18.3$ m, 修正：

$$\hat{\mathbf{x}}_1 = \begin{pmatrix} 15 \\ 15 \end{pmatrix} + \begin{pmatrix} 0.889 \\ 0.446 \end{pmatrix} \times (18.3 - 15) = \begin{pmatrix} 17.9 \\ 16.5 \end{pmatrix}$$

$$\mathbf{P}_1 = \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} - \begin{pmatrix} 0.889 \\ 0.446 \end{pmatrix} (1 \ 0) \right) \begin{pmatrix} 200.25 & 100.5 \\ 100.5 & 101 \end{pmatrix} = \begin{pmatrix} 22.2 & 11.2 \\ 11.2 & 56.2 \end{pmatrix}$$

完整追踪结果

| t | 真实位置 | 真实速度 | 测量 | 位置估计 | 速度估计 | σ_p |
|-----|------|------|------|------|------|------------|
| 0 | 0 | 20 | — | 0.0 | 15.0 | 10.0 |
| 1 | 20 | 20 | 18.3 | 17.9 | 16.5 | 4.7 |
| 2 | 40 | 20 | 43.1 | 39.8 | 18.7 | 3.5 |
| 3 | 60 | 20 | 56.8 | 59.2 | 19.4 | 3.0 |
| 4 | 80 | 20 | 82.4 | 80.0 | 19.8 | 2.7 |
| 5 | 100 | 20 | 97.6 | 99.7 | 19.9 | 2.5 |

表 5 多维卡尔曼滤波追踪结果

从结果中可以看到，尽管传感器只能测量位置，滤波器却成功地同时估计了位置和速度。速度估计从初始的 15 m/s 逐步收敛到接近真实值 20 m/s，这正是协方差矩阵发挥作用的结果——位置测量残差通过 σ_{pv} 的关联传递到速度估计上。

位置估计的标准差从 10 m 下降到 2.5 m，表明不确定性持续减小。若继续迭代，标准差将收敛到一个稳态值，由过程噪声 \mathbf{Q} 和测量噪声 \mathbf{R} 的相对大小决定。

2.6.5.7. 矩阵维度总结

多维卡尔曼滤波涉及多个矩阵，初学者容易混淆其维度。设状态向量维度为 n , 测量向量维度为 m , 控制向量维度为 l , 各矩阵的维度如下：

| 符号 | 维度 | 含义 |
|--------------|--------------|------|
| \mathbf{x} | $n \times 1$ | 状态向量 |
| \mathbf{z} | $m \times 1$ | 测量向量 |
| \mathbf{u} | $l \times 1$ | 控制向量 |

| | | |
|--------------|--------------|---------|
| \mathbf{F} | $n \times n$ | 状态转移矩阵 |
| \mathbf{B} | $n \times l$ | 控制矩阵 |
| \mathbf{H} | $m \times n$ | 观测矩阵 |
| \mathbf{Q} | $n \times n$ | 过程噪声协方差 |
| \mathbf{R} | $m \times m$ | 测量噪声协方差 |
| \mathbf{P} | $n \times n$ | 状态估计协方差 |
| \mathbf{K} | $n \times m$ | 卡尔曼增益 |

卡尔曼增益 \mathbf{K} 的维度 $n \times m$ 值得注意：它将 m 维的测量残差映射到 n 维的状态修正。即使测量维度低于状态维度（如本例中 $m = 1, n = 2$ ），滤波器仍能通过协方差的关联信息更新所有状态。

2.6.5.8. 本章小结

本章将一维卡尔曼滤波推广到多维情形。核心概念的对应关系如下：

| | 一维 | 多维 |
|------|--|---|
| 状态 | x | \mathbf{x} （向量） |
| 不确定性 | σ^2 （方差） | \mathbf{P} （协方差矩阵） |
| 增益 | $K = \frac{\sigma_{\text{pred}}^2}{\sigma_{\text{pred}}^2 + \sigma_z^2}$ | $\mathbf{K} = \mathbf{P}_{k k-1} \mathbf{H}^\top (\mathbf{H} \mathbf{P}_{k k-1} \mathbf{H}^\top + \mathbf{R})^{-1}$ |
| 预测方差 | $\sigma_{\text{pred}}^2 = \sigma_{k-1}^2 + \sigma_w^2$ | $\mathbf{P}_{k k-1} = \mathbf{F} \mathbf{P}_{k-1} \mathbf{F}^\top + \mathbf{Q}$ |
| 修正方差 | $\sigma_k^2 = (1 - K) \sigma_{\text{pred}}^2$ | $\mathbf{P}_k = (\mathbf{I} - \mathbf{K} \mathbf{H}) \mathbf{P}_{k k-1}$ |

多维形式的本质与一维相同：预测阶段不确定性增加，修正阶段不确定性减小，卡尔曼增益平衡预测与测量的信任程度。协方差矩阵的引入使滤波器能够利用状态之间的关联信息，即使某些状态不能直接测量（如速度），也可以通过与可测状态（如位置）的相关性间接估计。

至此，线性卡尔曼滤波的理论框架已经完整。然而，上述推导依赖一个关键假设：状态转移和观测方程都是线性的。当系统存在非线性时——例如雷达以极坐标测量目标位置——标准卡尔曼滤波不再适用。下一章将介绍扩展卡尔曼滤波（EKF），它通过局部线性化处理非线性系统。

2.6.6. 扩展卡尔曼滤波

前两章建立的卡尔曼滤波理论依赖于线性假设：状态转移方程 $\mathbf{x}_k = \mathbf{F} \mathbf{x}_{k-1}$ 和观测方程 $\mathbf{z}_k = \mathbf{H} \mathbf{x}_k$ 都是状态的线性函数。这一假设在许多场景下是合理的——匀速运动、弹簧振子、电路系统等都可以用线性方程描述。然而，现实中大量系统本质上是非线性的。本章将分析线性假设的局限性，并介绍扩展卡尔曼滤波（Extended Kalman Filter, EKF）如何通过局部线性化处理非线性系统。

2.6.6.1. 非线性系统的例子

考虑一个典型的雷达追踪问题。雷达站位于原点，追踪一架在二维平面上飞行的飞机。飞机的状态用直角坐标描述：

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}$$

其中 (x, y) 是位置, (v_x, v_y) 是速度。假设飞机做匀速直线运动, 状态转移方程为:

$$\begin{pmatrix} x_k \\ y_k \\ v_{x,k} \\ v_{y,k} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{k-1} \\ y_{k-1} \\ v_{x,k-1} \\ v_{y,k-1} \end{pmatrix}$$

这是标准的线性方程, 没有问题。困难出在观测方程上。

雷达并不直接测量飞机的直角坐标 (x, y) , 而是测量极坐标: 距离 r 和方位角 θ 。测量值与状态的关系为:

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \arctan\left(\frac{y}{x}\right)$$

或写成向量形式:

$$\mathbf{z} = \begin{pmatrix} r \\ \theta \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \arctan(y/x) \end{pmatrix}$$

这是一个非线性观测方程——测量值不是状态的线性组合, 无法写成 $\mathbf{z} = \mathbf{H}\mathbf{x}$ 的形式。平方根和反正切函数破坏了线性结构。

非线性带来的问题是什么? 回顾卡尔曼滤波的核心假设: 状态估计的不确定性用高斯分布描述, 而高斯分布经过线性变换后仍然是高斯分布。这一性质保证了预测和修正阶段的协方差计算是精确的。

但高斯分布经过非线性变换后通常不再是高斯分布。考虑一个简单的例子: 若 $x \sim \mathcal{N}(0, 1)$, 则 $y = x^2$ 的分布是卡方分布, 不是高斯分布。这意味着, 当观测方程非线性时, 直接套用卡尔曼滤波公式会产生系统性误差。

2.6.6.2. 线性化的思想

扩展卡尔曼滤波的核心思想是: 既然无法处理非线性函数, 就用线性函数来近似它。具体方法是在当前估计点对非线性函数进行泰勒展开, 保留一阶项。

设非线性函数为 $\mathbf{h}(\mathbf{x})$, 在点 \mathbf{x}_0 处展开:

$$\mathbf{h}(\mathbf{x}) \approx \mathbf{h}(\mathbf{x}_0) + \frac{\partial \mathbf{h}}{\partial \mathbf{x}}|_{\mathbf{x}_0} (\mathbf{x} - \mathbf{x}_0)$$

一阶导数 $\frac{\partial \mathbf{h}}{\partial \mathbf{x}}$ 是一个矩阵, 称为雅可比矩阵 (Jacobian), 记为 \mathbf{J} 。对于 m 维输出、 n 维输入的函数, 雅可比矩阵的维度是 $m \times n$, 第 (i, j) 元素为 $\frac{\partial h_i}{\partial x_j}$ 。

以雷达观测为例, 观测函数 $\mathbf{h}(\mathbf{x}) = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \arctan(y/x) \end{pmatrix}$ 对状态 $\mathbf{x} = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}$ 的雅可比矩阵为:

$$\mathbf{H}_k = \frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial v_x} & \frac{\partial r}{\partial v_y} \\ \frac{\partial \theta}{\partial x} & \frac{\partial \theta}{\partial y} & \frac{\partial \theta}{\partial v_x} & \frac{\partial \theta}{\partial v_y} \end{pmatrix}$$

由于 r 和 θ 都不依赖于速度 (v_x, v_y) , 后两列为零。计算非零偏导数:

$$\frac{\partial r}{\partial x} = \frac{x}{\sqrt{x^2 + y^2}} = \frac{x}{r}, \quad \frac{\partial r}{\partial y} = \frac{y}{r}$$

$$\frac{\partial \theta}{\partial x} = \frac{-y/x^2}{1 + (y/x)^2} = \frac{-y}{x^2 + y^2} = \frac{-y}{r^2}, \quad \frac{\partial \theta}{\partial y} = \frac{1/x}{1 + (y/x)^2} = \frac{x}{r^2}$$

因此, 雅可比矩阵为:

$$\mathbf{H}_k = \begin{pmatrix} x/r & y/r & 0 & 0 \\ -y/r^2 & x/r^2 & 0 & 0 \end{pmatrix}$$

这个矩阵在当前状态估计 (\hat{x}, \hat{y}) 处计算, 因此是时变的——每一步都需要重新计算。这是 EKF 与标准卡尔曼滤波的关键区别: 标准卡尔曼滤波中 \mathbf{H} 是常数, EKF 中 \mathbf{H}_k 随估计值变化。

2.6.6.3. 扩展卡尔曼滤波算法

将线性化思想应用到卡尔曼滤波的各个阶段, 得到 EKF 算法。设非线性状态转移方程为 $\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k$, 非线性观测方程为 $\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k$ 。

预测阶段

状态预测使用非线性函数:

$$\mathbf{x}_{k|k-1} = \mathbf{f}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k)$$

协方差预测需要状态转移函数的雅可比矩阵 $\mathbf{F}_k = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}|_{\hat{\mathbf{x}}_{k-1}}$:

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^\top + \mathbf{Q}$$

注意状态预测直接用非线性函数计算 (保持精度), 而协方差传播用线性化后的雅可比矩阵 (便于计算)。

修正阶段

卡尔曼增益使用观测函数的雅可比矩阵 $\mathbf{H}_k = \frac{\partial \mathbf{h}}{\partial \mathbf{x}}|_{\mathbf{x}_{k|k-1}}$:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^\top \left(\mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^\top + \mathbf{R} \right)^{-1}$$

状态修正使用非线性观测函数计算预测测量:

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k|k-1} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{h}(\mathbf{x}_{k|k-1}))$$

协方差修正:

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1}$$

整个算法的流程与标准卡尔曼滤波相同, 核心区别有两点: 第一, 状态预测和测量预测使用原始的非线性函数; 第二, 协方差传播和增益计算使用在当前估计点计算的雅可比矩阵。

EKF 算法

系统模型

状态转移: $\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k$

观测方程: $\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k$

预测

$$\mathbf{x}_{k|k-1} = \mathbf{f}(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_k)$$

$$\mathbf{F}_k = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}|_{\hat{\mathbf{x}}_{k-1}}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^\top + \mathbf{Q}$$

修正

$$\mathbf{H}_k = \frac{\partial \mathbf{h}}{\partial \mathbf{x}}|_{\mathbf{x}_{k|k-1}}$$

$$\mathbf{K}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^\top \left(\mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^\top + \mathbf{R} \right)^{-1}$$

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k|k-1} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{h}(\mathbf{x}_{k|k-1}))$$

$$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k|k-1}$$

2.6.6.4. 示例：雷达目标追踪

现在用 EKF 解决前面提出的雷达追踪问题。

系统设定

状态向量 $\mathbf{x} = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}$, 采样间隔 $\Delta t = 1$ s。

状态转移函数 (匀速运动, 线性):

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{x}$$

由于状态转移是线性的, 雅可比矩阵 \mathbf{F} 就是系数矩阵本身, 且为常数。

观测函数 (非线性):

$$\mathbf{h}(\mathbf{x}) = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \arctan(y/x) \end{pmatrix}$$

观测雅可比矩阵 (在预测状态 (x, y) 处计算):

$$\mathbf{H}_k = \begin{pmatrix} x/r & y/r & 0 & 0 \\ -y/r^2 & x/r^2 & 0 & 0 \end{pmatrix}, \quad r = \sqrt{x^2 + y^2}$$

过程噪声协方差:

$$\mathbf{Q} = \begin{pmatrix} 0.25 & 0 & 0.5 & 0 \\ 0 & 0.25 & 0 & 0.5 \\ 0.5 & 0 & 1 & 0 \\ 0 & 0.5 & 0 & 1 \end{pmatrix}$$

测量噪声协方差（距离标准差 5 m，角度标准差 0.01 rad）：

$$\mathbf{R} = \begin{pmatrix} 25 & 0 \\ 0 & 0.0001 \end{pmatrix}$$

真实轨迹与测量

设飞机从 (100, 200) m 出发，以 (10, 5) m/s 的速度飞行。真实轨迹和带噪声的雷达测量如下：

| T | 真实 x | 真实 y | 真实 r | 真实 θ | 测量 r | 测量 θ |
|---|------|------|-------|-------|-------|-------|
| 0 | 100 | 200 | 223.6 | 1.107 | — | — |
| 1 | 110 | 205 | 232.7 | 1.078 | 229.3 | 1.082 |
| 2 | 120 | 210 | 241.9 | 1.052 | 245.1 | 1.048 |
| 3 | 130 | 215 | 251.2 | 1.027 | 248.7 | 1.031 |
| 4 | 140 | 220 | 260.6 | 1.004 | 263.2 | 1.001 |
| 5 | 150 | 225 | 270.0 | 0.983 | 267.8 | 0.979 |

表 6 雷达追踪：真实轨迹与测量值（角度单位：rad）

第 1 步迭代详解

初始估计（故意设定误差）：

$$\hat{\mathbf{x}}_0 = \begin{pmatrix} 95 \\ 190 \\ 8 \\ 6 \end{pmatrix}, \quad \mathbf{P}_0 = \begin{pmatrix} 100 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 25 & 0 \\ 0 & 0 & 0 & 25 \end{pmatrix}$$

预测：

$$\mathbf{x}_{1|0} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 95 \\ 190 \\ 8 \\ 6 \end{pmatrix} = \begin{pmatrix} 103 \\ 196 \\ 8 \\ 6 \end{pmatrix}$$

预测状态对应的极坐标：

$$r_{\text{pred}} = \sqrt{103^2 + 196^2} = 221.4 \text{m}$$

$$\theta_{\text{pred}} = \arctan(196/103) = 1.087 \text{ rad}$$

计算观测雅可比矩阵（在预测状态处）：

$$\mathbf{H}_1 = \begin{pmatrix} 103/221.4 & 196/221.4 & 0 & 0 \\ -196/221.4^2 & 103/221.4^2 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0.465 & 0.885 & 0 & 0 \\ -0.00400 & 0.00210 & 0 & 0 \end{pmatrix}$$

协方差预测（ \mathbf{F} 为常数矩阵）：

$$\mathbf{P}_{1|0} = \mathbf{F}\mathbf{P}_0\mathbf{F}^\top + \mathbf{Q}$$

由于矩阵较大，此处省略具体数值，直接给出结果： $\mathbf{P}_{1|0}$ 的对角元素约为 (225, 225, 26, 26)。

卡尔曼增益：

$$\mathbf{S}_1 = \mathbf{H}_1 \mathbf{P}_{1|0} \mathbf{H}_1^\top + \mathbf{R}$$

$$\mathbf{K}_1 = \mathbf{P}_{1|0} \mathbf{H}_1^\top \mathbf{S}_1^{-1}$$

测量残差：

$$z_1 - h(x_{1|0}) = \begin{pmatrix} 229.3 \\ 1.082 \end{pmatrix} - \begin{pmatrix} 221.4 \\ 1.087 \end{pmatrix} = \begin{pmatrix} 7.9 \\ -0.005 \end{pmatrix}$$

状态修正：

$$\hat{x}_1 = \begin{pmatrix} 103 \\ 196 \\ 8 \\ 6 \end{pmatrix} + K_1 \begin{pmatrix} 7.9 \\ -0.005 \end{pmatrix}$$

完整追踪结果

| t | 真实 x | 估计 x | 真实 y | 估计 y | 真实 v _x | 估计 v _x |
|---|------|-------|------|-------|-------------------|-------------------|
| 0 | 100 | 95.0 | 200 | 190.0 | 10 | 8.0 |
| 1 | 110 | 107.2 | 205 | 202.1 | 10 | 9.1 |
| 2 | 120 | 118.5 | 210 | 209.4 | 10 | 9.8 |
| 3 | 130 | 129.1 | 215 | 214.8 | 10 | 10.0 |
| 4 | 140 | 139.8 | 220 | 219.9 | 10 | 10.1 |
| 5 | 150 | 150.1 | 225 | 224.8 | 10 | 10.0 |

表 7 EKF 追踪结果（部分状态）

尽管初始估计存在明显误差（位置误差约 12 m，速度误差 2-3 m/s），EKF 在几步之内就收敛到接近真实值。这说明滤波器能够有效处理非线性观测方程。

2.6.6.5. 线性化误差与 EKF 的局限性

EKF 通过一阶泰勒展开将非线性函数近似为线性，这种近似在函数“足够接近线性”时效果良好，但在强非线性情形下会产生显著误差。

考虑一个极端例子：观测函数 $h(x) = x^2$ 在 $x_0 = 0$ 处的线性化结果是 $h(x) \approx 0$ （因为导数为零），完全丢失了原函数的信息。虽然实际系统很少如此极端，但当非线性较强或估计误差较大时，线性化误差会累积，导致滤波器性能下降甚至发散。

EKF 的另一个实践困难是雅可比矩阵的计算。对于简单系统（如本章的雷达追踪），解析求导尚可接受；但对于复杂系统，手工推导雅可比矩阵既繁琐又容易出错。虽然可以用数值差分近似，但这会引入额外的计算误差。

此外，EKF 假设线性化后的系统仍然满足高斯分布，这在理论上并不成立。非线性变换会使高斯分布变形，而 EKF 强行用高斯分布近似，相当于丢弃了分布的高阶矩信息。

这些局限促使研究者寻找更好的非线性滤波方法。下一章将介绍无迹卡尔曼滤波 (UKF)，它采用完全不同的思路：与其线性化函数，不如用采样点直接“探测”非线性变换的效果。

2.6.6.6. 本章小结

扩展卡尔曼滤波通过局部线性化将标准卡尔曼滤波推广到非线性系统。其核心思想是在当前估计点对非线性函数进行一阶泰勒展开，用雅可比矩阵替代原来的常数矩阵。

与标准卡尔曼滤波相比，EKF 的主要变化是：

| | 标准 KF | EKF |
|-------|---------------------|--|
| 状态转移 | \mathbf{F} (常数矩阵) | $\mathbf{f}(\cdot)$ (非线性函数) + \mathbf{F}_k (雅可比) |
| 观测方程 | \mathbf{H} (常数矩阵) | $\mathbf{h}(\cdot)$ (非线性函数) + \mathbf{H}_k (雅可比) |
| 协方差传播 | 精确 | 近似 (一阶) |
| 计算复杂度 | 较低 | 较高 (需计算雅可比) |

EKF 的适用条件是非线性程度适中且估计误差较小, 使得线性化近似足够准确。当这些条件不满足时, 需要考虑更高级的方法, 如无迹卡尔曼滤波。

2.6.7. 无迹卡尔曼滤波

上一章介绍的扩展卡尔曼滤波通过线性化处理非线性系统, 其本质是用简单函数近似复杂函数。这种方法在弱非线性情形下表现良好, 但当非线性较强时, 一阶近似的误差会显著影响滤波性能。本章介绍一种完全不同的思路——无迹卡尔曼滤波 (Unscented Kalman Filter, UKF), 它不再尝试近似非线性函数本身, 而是通过精心选取的采样点直接捕捉概率分布经过非线性变换后的统计特性。

2.6.7.1. 从函数近似到分布近似

EKF 的困难根源在于: 非线性函数会扭曲概率分布的形状。若输入是高斯分布, 经过非线性变换后, 输出通常不再是高斯分布。EKF 的策略是线性化函数, 使得输出仍为高斯分布, 但这是以牺牲精度为代价的。

UKF 的创始人 Jeffrey Uhlmann 提出了一个关键洞察: 近似概率分布比近似非线性函数更容易。具体而言, 与其用线性函数近似 $\mathbf{h}(\mathbf{x})$, 不如直接用一组精心选取的点来代表输入分布, 将这些点通过真实的非线性函数变换, 再从变换后的点恢复输出分布的统计特性。

这一思想可以用一个简单的例子说明。设 $x \sim \mathcal{N}(0, 1)$, 考虑非线性变换 $y = x^2$ 。EKF 在 $x = 0$ 处线性化, 得到 $y \approx 0$ (因为导数为零), 完全丢失了信息。而 UKF 的做法是选取几个代表点, 比如 $x \in \{-1, 0, 1\}$, 计算它们的变换值 $y \in \{1, 0, 1\}$, 再从这些点估计 y 的均值和方差。即使不知道 y 的精确分布, 也能得到比 EKF 更准确的统计量估计。

这就是无迹变换 (Unscented Transform) 的基本思想: 用确定性采样点代替随机采样, 以较少的计算量捕捉非线性变换的效果。

2.6.7.2. 无迹变换

无迹变换是 UKF 的核心组件。给定一个 n 维随机变量 \mathbf{x} , 其均值为 $\boldsymbol{\mu}$, 协方差为 \mathbf{P} , 目标是计算经过非线性函数 $\mathbf{y} = \mathbf{f}(\mathbf{x})$ 变换后 \mathbf{y} 的均值和协方差。

Sigma 点的选取

无迹变换选取 $2n + 1$ 个确定性采样点, 称为 Sigma 点:

$$\chi_0 = \boldsymbol{\mu}$$

$$\chi_i = \boldsymbol{\mu} + \left(\sqrt{(n + \lambda)\mathbf{P}} \right)_i, \quad i = 1, \dots, n$$

$$\chi_{i+n} = \boldsymbol{\mu} - \left(\sqrt{(n + \lambda)\mathbf{P}} \right)_i, \quad i = 1, \dots, n$$

其中 $(\sqrt{(n + \lambda)P})_i$ 表示矩阵平方根的第 i 列, $\lambda = \alpha^2(n + \kappa) - n$ 是缩放参数。 α 控制 Sigma 点的分散程度, 通常取较小的正数 (如 10^{-3}); κ 通常取 0 或 $3 - n$ 。

矩阵平方根 \sqrt{P} 定义为满足 $\sqrt{P}(\sqrt{P})^\top = P$ 的矩阵, 通常用 Cholesky 分解计算。

Sigma 点的几何含义是: χ_0 位于分布中心, 其余 $2n$ 个点沿协方差矩阵的主轴对称分布, 距离中心的远近由 λ 控制。这组点能够精确匹配原分布的均值和协方差。

权重分配

每个 Sigma 点被赋予权重, 用于计算变换后的统计量。均值权重和协方差权重分别为:

$$W_0^m = \frac{\lambda}{n + \lambda}$$

$$W_0^c = \frac{\lambda}{n + \lambda} + (1 - \alpha^2 + \beta)$$

$$W_i^m = W_i^c = \frac{1}{2(n + \lambda)}, \quad i = 1, \dots, 2n$$

参数 β 用于融入分布的先验知识, 对于高斯分布, $\beta = 2$ 是最优选择。

变换与恢复

将所有 Sigma 点通过非线性函数变换:

$$\mathcal{Y}_i = f(\chi_i), \quad i = 0, 1, \dots, 2n$$

变换后分布的均值和协方差估计为:

$$\boldsymbol{\mu}_y = \sum_{i=0}^{2n} W_i^m \mathcal{Y}_i$$

$$\mathbf{P}_y = \sum_{i=0}^{2n} W_i^c (\mathcal{Y}_i - \boldsymbol{\mu}_y)(\mathcal{Y}_i - \boldsymbol{\mu}_y)^\top$$

这一过程完全绕过了函数的线性化, 直接从采样点的变换结果恢复统计量。由于 Sigma 点的选取能够精确匹配原分布的前两阶矩, 对于高斯输入, 无迹变换能够准确捕捉输出的均值和协方差, 精度达到二阶 (泰勒展开的二阶项), 优于 EKF 的一阶精度。

2.6.7.3. 无迹卡尔曼滤波算法

将无迹变换应用于卡尔曼滤波的预测和修正阶段, 即得到 UKF 算法。

预测阶段

首先, 根据上一时刻的状态估计 \hat{x}_{k-1} 和协方差 P_{k-1} 生成 Sigma 点:

$$\chi_{k-1}^{(i)}, \quad i = 0, 1, \dots, 2n$$

将每个 Sigma 点通过状态转移函数:

$$\chi_{k|k-1}^{(i)} = f(\chi_{k-1}^{(i)}, u_k)$$

从变换后的 Sigma 点计算预测均值和协方差:

$$\mathbf{x}_{k|k-1} = \sum_{i=0}^{2n} W_i^m \chi_{k|k-1}^{(i)}$$

$$\mathbf{P}_{k|k-1} = \sum_{i=0}^{2n} W_i^c (\chi_{k|k-1}^{(i)} - \mathbf{x}_{k|k-1}) (\chi_{k|k-1}^{(i)} - \mathbf{x}_{k|k-1})^\top + \mathbf{Q}$$

过程噪声协方差 \mathbf{Q} 在最后加入，与 EKF 的处理方式相同。

修正阶段

将预测 Sigma 点通过观测函数：

$$\mathcal{Z}_k^{(i)} = \mathbf{h}(\chi_{k|k-1}^{(i)})$$

计算预测测量的均值：

$$\mathbf{z}_{k|k-1} = \sum_{i=0}^{2n} W_i^m \mathcal{Z}_k^{(i)}$$

计算测量预测的协方差和状态-测量的互协方差：

$$\mathbf{P}_{zz} = \sum_{i=0}^{2n} W_i^c (\mathcal{Z}_k^{(i)} - \mathbf{z}_{k|k-1}) (\mathcal{Z}_k^{(i)} - \mathbf{z}_{k|k-1})^\top + \mathbf{R}$$

$$\mathbf{P}_{xz} = \sum_{i=0}^{2n} W_i^c (\chi_{k|k-1}^{(i)} - \mathbf{x}_{k|k-1}) (\mathcal{Z}_k^{(i)} - \mathbf{z}_{k|k-1})^\top$$

卡尔曼增益：

$$\mathbf{K}_k = \mathbf{P}_{xz} \mathbf{P}_{zz}^{-1}$$

状态和协方差修正：

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k|k-1} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{z}_{k|k-1})$$

$$\mathbf{P}_k = \mathbf{P}_{k|k-1} - \mathbf{K}_k \mathbf{P}_{zz} \mathbf{K}_k^\top$$

UKF 算法

生成 Sigma 点

$$\chi_0 = \hat{\mathbf{x}}_{k-1}, \quad \chi_i = \hat{\mathbf{x}}_{k-1} \pm \left(\sqrt{(n + \lambda) \mathbf{P}_{k-1}} \right)_i$$

预测

$$\chi_{k|k-1}^{(i)} = \mathbf{f}(\chi_{k-1}^{(i)}), \quad i = 0, \dots, 2n$$

$$\mathbf{x}_{k|k-1} = \sum W_i^m \chi_{k|k-1}^{(i)}$$

$$\mathbf{P}_{k|k-1} = \sum W_i^c (\chi_{k|k-1}^{(i)} - \mathbf{x}_{k|k-1})(\cdot)^\top + \mathbf{Q}$$

修正

$$\mathcal{Z}_k^{(i)} = \mathbf{h}(\chi_{k|k-1}^{(i)})$$

$$\mathbf{z}_{k|k-1} = \sum W_i^m \mathcal{Z}_k^{(i)}$$

$$\mathbf{P}_{zz} = \sum W_i^c (\mathcal{Z}_k^{(i)} - \mathbf{z}_{k|k-1})(\cdot)^\top + \mathbf{R}$$

$$\mathbf{P}_{xz} = \sum W_i^c \left(\chi_{k|k-1}^{(i)} - \mathbf{x}_{k|k-1} \right) \left(\mathbf{z}_k^{(i)} - \mathbf{z}_{k|k-1} \right)^{\top}$$

$$\mathbf{K}_k = \mathbf{P}_{xz} \mathbf{P}_{zz}^{-1}$$

$$\hat{\mathbf{x}}_k = \mathbf{x}_{k|k-1} + \mathbf{K}_k (\mathbf{z}_k - \mathbf{z}_{k|k-1})$$

$$\mathbf{P}_k = \mathbf{P}_{k|k-1} - \mathbf{K}_k \mathbf{P}_{zz} \mathbf{K}_k^{\top}$$

2.6.7.4. 示例：雷达追踪问题的 UKF 实现

沿用上一章的雷达追踪场景，用 UKF 重新求解并与 EKF 对比。

系统设定

状态向量 $\mathbf{x} = \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}$ ($n = 4$)，因此需要 $2n + 1 = 9$ 个 Sigma 点。

参数选取： $\alpha = 0.001$, $\kappa = 0$, $\beta = 2$, 则 $\lambda = \alpha^2(n + \kappa) - n \approx -4$ 。

其他参数 (\mathbf{Q} 、 \mathbf{R} 、初始条件) 与上一章相同。

第 1 步迭代

初始估计 $\hat{\mathbf{x}}_0 = \begin{pmatrix} 95 \\ 190 \\ 8 \\ 6 \end{pmatrix}$, 初始协方差 $\mathbf{P}_0 = \text{diag}(100, 100, 25, 25)$ 。

生成 Sigma 点。首先计算 $(n + \lambda)\mathbf{P}_0 \approx 0.000004 \times \mathbf{P}_0$ (由于 $\lambda \approx -4$, $n + \lambda \approx 0.000004$)。实际计算中 α 通常取较大值如 0.1 以避免数值问题，此处为说明原理采用典型参数。设 $\alpha = 0.1$, 则 $\lambda = 0.01 \times 4 - 4 = -3.96$, $n + \lambda = 0.04$ 。

矩阵平方根 $\sqrt{0.04 \times \mathbf{P}_0} = 0.2 \times \text{diag}(10, 10, 5, 5) = \text{diag}(2, 2, 1, 1)$ 。

9 个 Sigma 点为：

$$\chi_0 = \begin{pmatrix} 95 \\ 190 \\ 8 \\ 6 \end{pmatrix}$$

$$\chi_1 = \begin{pmatrix} 97 \\ 190 \\ 8 \\ 6 \end{pmatrix}, \quad \chi_5 = \begin{pmatrix} 93 \\ 190 \\ 8 \\ 6 \end{pmatrix}$$

$$\chi_2 = \begin{pmatrix} 95 \\ 192 \\ 8 \\ 6 \end{pmatrix}, \quad \chi_6 = \begin{pmatrix} 95 \\ 188 \\ 8 \\ 6 \end{pmatrix}$$

$$\chi_3 = \begin{pmatrix} 95 \\ 190 \\ 9 \\ 6 \end{pmatrix}, \quad \chi_7 = \begin{pmatrix} 95 \\ 190 \\ 7 \\ 6 \end{pmatrix}$$

$$\chi_4 = \begin{pmatrix} 95 \\ 190 \\ 8 \\ 7 \end{pmatrix}, \quad \chi_8 = \begin{pmatrix} 95 \\ 190 \\ 8 \\ 5 \end{pmatrix}$$

将每个 Sigma 点通过状态转移函数（匀速运动）：

$$\chi_{1|0}^{(i)} = \mathbf{F} \chi_0^{(i)}$$

例如 $\chi_{1|0}^{(0)} = \begin{pmatrix} 95+8 \\ 190+6 \\ 8 \\ 6 \end{pmatrix} = \begin{pmatrix} 103 \\ 196 \\ 8 \\ 6 \end{pmatrix}$ 。

计算预测均值（各 Sigma 点变换结果的加权平均）：

$$\mathbf{x}_{1|0} = \sum_{i=0}^8 W_i^m \chi_{1|0}^{(i)}$$

由于状态转移是线性的，预测均值与 EKF 相同： $\mathbf{x}_{1|0} = \begin{pmatrix} 103 \\ 196 \\ 8 \\ 6 \end{pmatrix}$ 。

将预测 Sigma 点通过观测函数：

$$\mathcal{Z}^{(i)} = \begin{pmatrix} \sqrt{x_i^2 + y_i^2} \\ \arctan(y_i/x_i) \end{pmatrix}$$

例如 $\mathcal{Z}^{(0)} = \begin{pmatrix} \sqrt{103^2 + 196^2} \\ \arctan(196/103) \end{pmatrix} = \begin{pmatrix} 221.4 \\ 1.087 \end{pmatrix}$ 。

从变换后的点计算预测测量均值 $\mathbf{z}_{1|0}$ 、协方差 \mathbf{P}_{zz} 和互协方差 \mathbf{P}_{xz} ，进而计算卡尔曼增益和状态修正。

追踪结果对比

| T | 真实 x | EKF x | UKF x | EKF v_x | UKF v_x |
|---|------|-------|-------|-----------|-----------|
| 0 | 100 | 95.0 | 95.0 | 8.0 | 8.0 |
| 1 | 110 | 107.2 | 107.4 | 9.1 | 9.2 |
| 2 | 120 | 118.5 | 118.7 | 9.8 | 9.9 |
| 3 | 130 | 129.1 | 129.2 | 10.0 | 10.0 |
| 4 | 140 | 139.8 | 139.9 | 10.1 | 10.0 |
| 5 | 150 | 150.1 | 150.0 | 10.0 | 10.0 |

表 8 EKF 与 UKF 追踪结果对比

在这个弱非线性场景下，EKF 与 UKF 的结果非常接近。这是因为雷达观测函数虽然是非线性的，但在估计点附近的非线性程度并不强，一阶线性化已经足够准确。

两种方法的差异在强非线性场景下会更加明显。例如，当目标靠近雷达站（ r 较小）时，角度测量的非线性显著增强，此时 UKF 的优势会更加突出。

2.6.7.5. EKF 与 UKF 的对比

两种方法代表了处理非线性的两种不同哲学：EKF 近似函数，UKF 近似分布。下表总结了它们的主要区别。

| | EKF | UKF |
|-------|------------|----------------------|
| 核心思想 | 线性化非线性函数 | 采样传播概率分布 |
| 近似对象 | 函数（泰勒展开） | 分布（Sigma 点） |
| 精度 | 一阶 | 二阶 |
| 雅可比矩阵 | 需要解析求导 | 不需要 |
| 计算量 | 较低 | 较高 ($2n + 1$ 次函数求值) |
| 实现难度 | 需推导雅可比 | 仅需实现系统函数 |
| 数值稳定性 | 一般 | 较好 |
| 适用场景 | 弱非线性、雅可比易求 | 强非线性、雅可比难求 |

在实际选择时，有几条经验法则。如果系统的非线性程度较弱，且雅可比矩阵容易求解，EKF 是更简单高效的选择。如果非线性较强，或者雅可比矩阵难以解析求导（例如系统模型来自仿真器或神经网络），UKF 是更稳健的选择。当状态维度很高 ($n > 100$) 时，UKF 的 $2n + 1$ 个 Sigma 点会带来显著的计算负担，此时可能需要考虑其他方法，如粒子滤波。

2.6.7.6. 参数选择指南

UKF 的性能受 α 、 β 、 κ 三个参数影响。以下是常用的选择策略。

参数 α 控制 Sigma 点的分散程度。较小的 α （如 10^{-3} ）使 Sigma 点紧密围绕均值，适合近似局部非线性；较大的 α （如 1）使 Sigma 点分散更广，能够捕捉更大范围的非线性效应，但可能引入高阶误差。实践中 $\alpha = 0.001$ 到 0.01 是常见选择。

参数 β 融入分布的先验信息。对于高斯分布， $\beta = 2$ 是理论最优值，能够精确捕捉四阶矩。若分布明显偏离高斯，可以调整 β ，但通常影响不大。

参数 κ 是次要的缩放参数。常见选择是 $\kappa = 0$ （简化计算）或 $\kappa = 3 - n$ （保证正定性）。当 $n + \lambda$ 接近零或为负时，可能导致数值问题，此时需要调整 κ 。

对于大多数应用，默认参数 $\alpha = 0.001$ ， $\beta = 2$ ， $\kappa = 0$ 是合理的起点，可根据实际效果微调。

2.6.7.7. 本章小结

无迹卡尔曼滤波提供了一种不同于 EKF 的非线性处理思路。它的核心是无迹变换：通过精心选取的 Sigma 点代表概率分布，将这些点通过真实的非线性函数变换，再从变换后的点恢复统计特性。这种方法避免了雅可比矩阵的计算，且在理论上具有更高的近似精度。

UKF 的主要优势包括：不需要解析求导，实现更简单；二阶精度优于 EKF 的一阶精度；对强非线性系统更稳健。主要代价是计算量较大，需要 $2n + 1$ 次函数求值。

至此，我们已经完成了从递推平均到 UKF 的完整旅程。下表回顾了各方法的演进脉络：

| 方法 | 解决的问题 | 核心思想 |
|---------|--------|---------------|
| 递推平均 | 静态量估计 | 历史数据等权平均 |
| g-h 滤波器 | 动态系统追踪 | 预测 + 修正（固定增益） |

| | | |
|-------|---------|---------------|
| 卡尔曼滤波 | 最优增益选取 | 基于方差最小化动态计算增益 |
| 多维 KF | 多状态关联估计 | 协方差矩阵描述状态间关联 |
| EKF | 非线性系统 | 局部线性化（雅可比矩阵） |
| UKF | 强非线性系统 | 采样传播（Sigma 点） |

每一步演进都是为了解决前一方法的局限：递推平均无法追踪动态系统，g-h 滤波器的增益需要手动设定，标准卡尔曼滤波仅适用于线性系统，EKF 在强非线性时精度下降。理解这一演进逻辑，比记住公式本身更为重要——它揭示了工程方法论的核心：从具体问题出发，逐步放松假设，构建更普适的解决方案。

3. 实战技术篇

3.1. 通信协议设定

3.2. 相机标定与手眼标定

3.3. 时间戳对齐

3.4. 弹道解算

4. RoboMaster 应用篇

4.1. 工业相机

4.2. 装甲板识别

4.3. 装甲板的跟踪

4.4. 能量机关识别

4.5. 自瞄算法设计

4.6. 串口模块

5. 进阶篇

5.1. 雷达站视觉方案

5.2. 哨兵决策视觉

5.3. 性能优化技巧

5.4. 实战经验与坑点总结

6. 项目分析

6.1. rm_vision

6.2. rm.cv.fans

6.3. 同济自瞄

Stokes' theorem

Definition 6.3.1

Let Σ be a smooth oriented surface in \mathbb{R}^3 with boundary $\partial\Sigma \equiv \Gamma$. If a vector field $\boxed{\mathbf{F}(x, y, z)} = (F_x(x, y, z), F_y(x, y, z), F_z(x, y, z))$ is defined and has continuous first order partial derivatives in a region containing Σ , then

$$\iint_{\Sigma} (\nabla \times \mathbf{F}) \cdot \Sigma = \oint_{\partial\Sigma} \mathbf{F} \cdot d\Gamma$$

Information extracted from a well-known public encyclopedia

Definition 5 Stokes' theorem

这是引用内容。可以放多行文本。

Index of Tables

| | |
|-----------------------------------|-----|
| 表 1 递推平均追踪动态目标的失效 | 323 |
| 表 2 g-h 滤波器追踪结果 | 324 |
| 表 3 同一数据下两种方法的对比 | 324 |
| 表 4 一维卡尔曼滤波追踪赛车 | 329 |
| 表 5 多维卡尔曼滤波追踪结果 | 335 |
| 表 6 雷达追踪：真实轨迹与测量值（角度单位：rad） | 340 |
| 表 7 EKF 追踪结果（部分状态） | 341 |
| 表 8 EKF 与 UKF 追踪结果对比 | 346 |

Index of Listings

| | |
|--|-----|
| 代码 1 Example python code printing text. | 320 |
|--|-----|