面向自动驾驶的C++实战
第6章 C++编程的资源使用和管理

主　讲：姜朝峰

公众号：自动驾驶之心

自动驾驶之心
全栈知识矩阵

自动驾驶之心，专注自动驾驶与AI

公众号

干货每日放送

B站

不定期直播+最新视频

知识星球

海量图文教程和Paper分享

视频号

技术视频一站式分享

主要内容

1. 计算机资源的使用
2. 文件管理：数据存储的钥匙
3. 内存管理：优化系统的血管
4. 多线程管理：梳理错综复杂的世界
5. 网络管理：连接信息高速公路
6. GPU管理：释放计算的潜能

1 计算机资源的使用

1. 计算机资源的使用

1.1 计算机资源

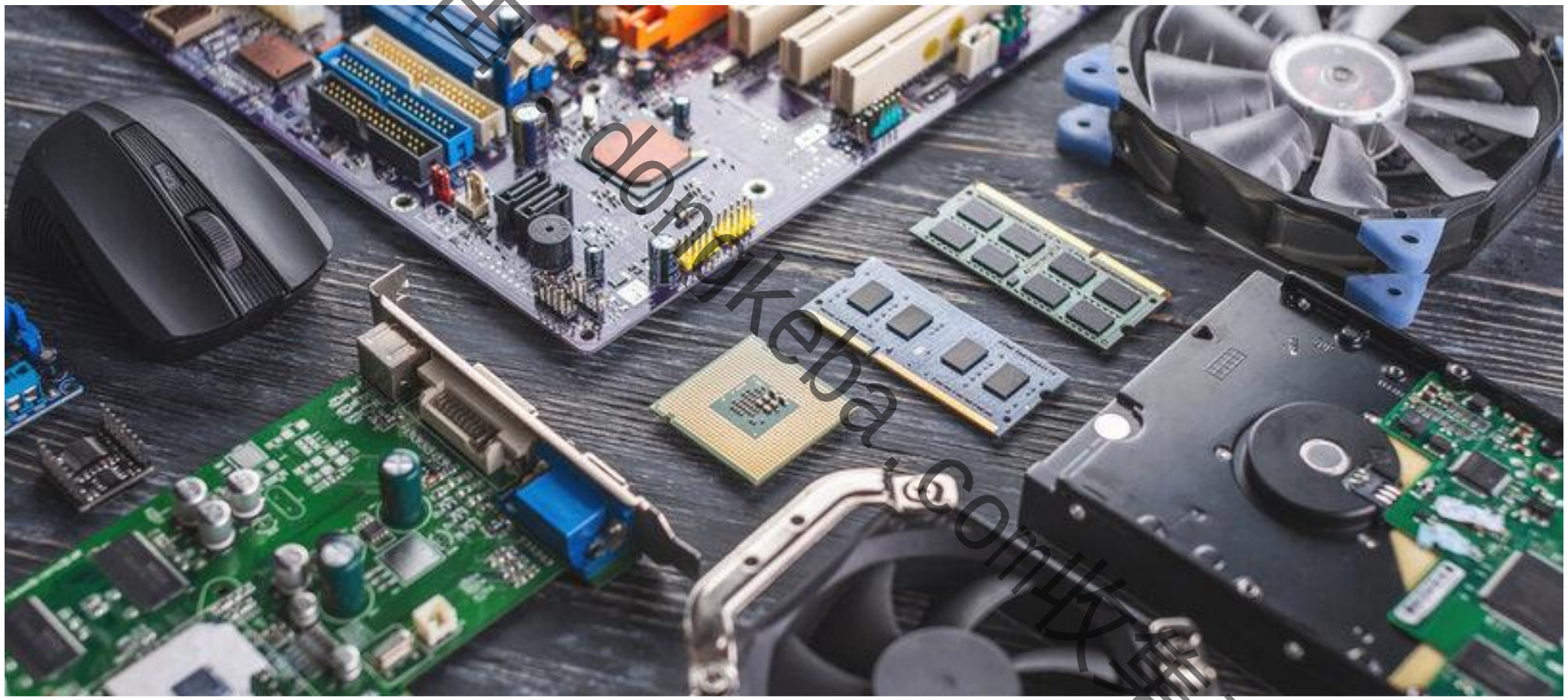
计算机的硬件组成

1. **中央处理器（CPU）：**计算机的核心处理器，负责执行指令和处理数据。
2. **内存（RAM）：**随机存取存储器，用于临时存储当前正在执行的程序和数据。
3. **存储设备（Storage Disk）：机械硬盘（HDD），**传统的机械式存储设备，容量大，速度较慢；**固态硬盘（SSD），**新型存储设备，速度快，稳定性高。
4. **网络接口卡（NIC）：**用于连接计算机网络，实现数据通信。
5. **显卡（GPU）：**负责图形处理的设备，尤其在图形密集型应用和并行计算中至关重要。
6. **主板：**连接和协调各个硬件组件的电路板，包括CPU插槽、内存插槽、扩展插槽等。
7. **电源供应器（PSU）：**为计算机各个组件提供电力的设备。
8. **RTC（Real-Time Clock）：**为计算机提供长期准确的计时器。
9. **其他输入设备：**键盘、鼠标、扫描仪、触摸屏等，用于向计算机输入数据。
10. **其他输出设备：**显示器、打印机、扬声器等，用于输出计算机处理结果。
11. **外部接口和端口：**USB接口、HDMI端口、音频接口等，用于连接外部设备和配件。
12. **……**

1. 计算机资源的使用

1.1 计算机资源

计算机的硬件组成



计算机的硬件

1. 计算机资源的使用

1.1 计算机资源

计算机资源

在计算机编程中，**资源（Resource）**是一个广泛、虚拟的概念，涉及程序运行时使用的各种系统资源和硬件资源。

对应计算机硬件，有如下计算机资源：

硬件	资源	举例
CPU	系统资源	进程、线程
内存RAM	内存资源	堆内存、栈内存
硬盘Disk	文件系统资源	文件句柄
网卡NIC	网络资源	网络连接
显卡GPU	规模并行计算资源	显存
其他设备	• 打印机资源 • 摄像头资源 • 音频设备资源	……

1. 计算机资源的使用

1.1 计算机资源

资源抽象的过程

把计算机硬件转变为可供程序使用的资源是一个多层次的过程：

1. **硬件层：**包括CPU、内存、硬盘、网卡等物理设备。
2. **操作系统内核：**
   - **设备驱动程序：**与特定硬件通信的软件。
   - **OS资源管理：**内核负责分配和调度硬件资源。
   - **抽象层：**提供统一的接口来访问不同的硬件。
3. **操作系统API和运行时库：**为应用程序提供访问系统资源的接口，如Windows文件操作、网络通信库等；或者C运行时库，提供更高级的资源管理功能。
4. **编程语言和框架：**提供更抽象的接口来使用系统资源、对系统资源进行管理，如C++的RAII机制。

举例：当在C++中使用new分配堆内存时，这个请求通过运行时库传递给操作系统的内存管理部分，后者再通过硬件抽象层控制物理内存。

这个过程使得程序员可以使用高级抽象来管理底层硬件资源，而不需要直接处理复杂的硬件细节。

1. 计算机资源的使用

1.2 资源使用

程序 = 算法 + 数据结构？

是计算机科学中的一个**经典观点**，由计算机科学家尼克劳斯·维尔特（Niklaus Wirth）提出。算法是解决问题的**方法和步骤**，而数据是算法操作的**对象**。

但这个观点太过于抽象和简化，只阐述了程序的核心部分。现代程序的复杂性和多样性，包含了更多的内容：数据持久化、网络通信、用户界面、安全性等等等等。现代程序开发更像是将多个复杂系统、将**各种资源**整合在一起，需要考虑的因素远超早期的简单程序。

硬件层、**操作系统内核**、**系统API与运行时库**和**语言框架**将资源抽象出来，提供给程序去使用各种**计算机资源**，以实现各种目的。

1. 计算机资源的使用

1.2 资源使用

如何使用资源

- **内存资源：**声明、定义和操作变量来使用栈内存；使用new和delete操作符进行动态内存分配和释放。
- **文件系统资源：**使用C风格的FILE*和相关函数；使用std::fstream类进行文件的读写操作。
- **多线程资源：**使用std::thread创建和管理线程；使用std::mutex, std::condition_variable等进行线程同步。
- **网络资源：**通常使用操作系统提供的socket API；使用网络库如Boost.Asio, Qt Network等。
- **时间资源：**使用std::chrono库进行时间和计时操作。
- **图形资源：**使用图形库如CUDA、OpenGL、DirectX。
- **系统资源：**通过操作系统API访问，如Windows API或POSIX函数。
- **其他资源：……**

1. 计算机资源的使用

1.2 资源使用

资源的使用并非一成不变

- **早期**：CPU和计算机不支持多线程，不用考虑多线程的互斥和并发；没有网卡，不需要考虑网络资源的使用和管理，也就不需要网络编程。
- **1980s**：光盘和软盘是主流媒介，程序员需要了解光驱资源的使用，掌握如何从光盘、软盘中读取和写入数据。
- **1990s-2010s**：互联网快速兴起，程序员需要学习和了解越来越多的网络知识和网络编程技能，以通过各种网络协议进行数据传输；GPU是边缘设备，只有媒体播放、游戏渲染等少数领域的程序员需要了解GPU资源的编程使用。
- **2010s至今**：随着深度学习的兴起、人工智能时代的到来，越来越多的计算被放到GPU上以加速，GPU从一个边缘硬件设备变成核心硬件设备，GPU的资源使用和管理愈发重要。

1. 计算机资源的使用

1.2 资源使用

难度和复杂性

仅仅是掌握各种资源的原理、协议、并使用API，就已经是一个很大的挑战了。以网络编程为例来具体说明：

1. **协议复杂性**。TCP/IP、UDP、HTTP、HTTPS、WebSocket等不同协议的使用。
2. **Socket编程**。创建、绑定、监听、接受连接等基本操作；处理阻塞和非阻塞I/O；设置各种socket选项。
3. **数据序列化和反序列化**。网络字节序和主机字节序的转换；网络数据的反序列化解析。
4. **错误处理**。处理网络超时、连接断开、数据包丢失等情况；实现重试机制和错误恢复。
5. **安全性**。SSL/TLS的使用、证书管理和验证。
6. **异步编程**。处理并发连接、使用回调、异步/等待模式。
7. **网络库的选择和使用**。如 Boost.Asio、libcurl、Qt Network。
8. **性能优化**。使用epoll、select、kqueue等高效I/O多路复用技术；实现高并发服务器。
9. **其他**。……

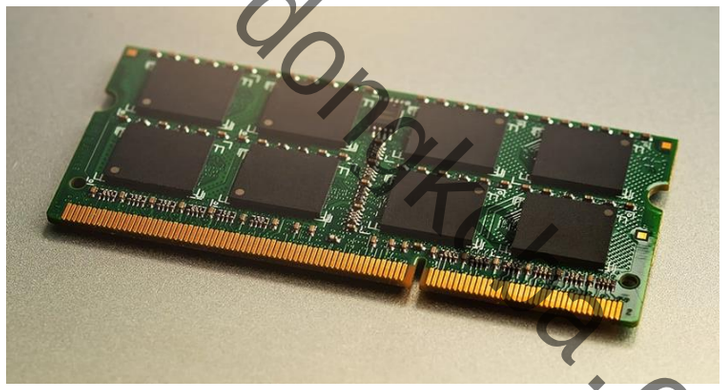
1. 计算机资源的使用

1.2 资源使用

难度和复杂性

受限于广度和复杂度，对于计算机的主要资源的使用，本章会做大概讲解，让大家有一定的认识和理解，但不会拓展很深入。

SSD



RAM

CPU

NIC

GPU

1. 计算机资源的使用

1.2 资源使用

相关书籍

如果还打算深入某个方向、领域钻研，可以阅读和学习：

- **文件系统资源/多线程资源 -> 系统编程/环境编程**
  - 《现代操作系统（Modern Operating Systems）》，by Andrew S. Tanenbaum
  - 《UNIX环境高级编程（Advanced Programming in the UNIX Environment）》，by W. Richard Stevens, Stephen A. Rago
  - 《Linux/UNIX系统编程手册（The Linux Programming Interface）》，by Michael Kerrisk
- **网络资源 -> 网络编程**
  - 《计算机网络：自顶向下方法（Computer Networking: A Top-Down Approach）》，by James F. Kurose, Keith W. Ross
  - 《UNIX网络编程（UNIX Network Programming）》，by W. Richard Stevens
  - 《TCP/IP详解（TCP/IP Illustrated）》，by W. Richard Stevens
- **GPU资源 -> GPU编程/高性能计算**
  - 《Programming Massively Parallel Processors: A Hands-on Approach》，by David B. Kirk, Wen-mei W. Hwu
  - 《CUDA by Example: An Introduction to General-Purpose GPU Programming》，by Jason Sanders, Edward Kandrot
  - NVIDIA的官方CUDA文档和教程

1. 计算机资源的使用

1.3 资源管理

对资源使用的要求复杂、多样之后，自然而然产生了“**管理**”的必要。

资源管理的难题

- **有限性**：系统资源，如内存、硬盘空间、网络连接等，通常是有限的，不能任意申请；
- **并发访问**：多个程序或线程可能同时需要访问和使用同一个系统资源，资源需要协调；
- **生命周期管理**：系统资源通常需要显式地申请和释放。如果程序在使用完资源后忘记释放，就会导致资源泄漏；
- **异常处理**：在申请、使用和释放资源的过程中，可能会发生各种异常，如内存不足、文件无法打开、网络连接中断等。如何恰当地处理这些异常，确保资源在异常情况下也能正确释放，是另一个难题。
- **性能优化**：需要在性能和资源整体使用效率之间取得平衡。
- **跨语言**和**系统的兼容性**等……

生命周期问题的主要方案是：RAII；资源优化问题需要长期的权衡经验积累。

2

文件管理：
数据存储的钥匙

2. 文件管理：数据存储的钥匙

2.1 文件系统操作

计算机的硬盘，被操作系统抽象为文件系统进行操作和管理。
(如果不了解**文件系统**，请先学习**《操作系统概念》-存储管理** 章节）
Modern C++中常用的**文件系统**操作，主要在**<filesystem>**中提供实现。

文件系统操作	使用的函数	功能描述
检查文件/目录存在	std::filesystem::exists	检查文件或目录是否存在
获取文件大小	std::filesystem::file_size	获取文件的大小（以字节为单位）
创建目录	std::filesystem::create_directory	创建一个目录
删除文件/目录	std::filesystem::remove	删除一个文件或目录
遍历目录	std::filesystem::directory_iterator	遍历目录中的文件和子目录
复制文件	std::filesystem::copy	复制文件
移动/重命名文件	std::filesystem::rename	移动或重命名文件
获取文件属性	std::filesystem::status	获取文件的各种属性，如文件类型、权限等
修改文件权限	std::filesystem::permissions	修改文件或目录的权限
查询磁盘空间	std::filesystem::space	查询磁盘的空间信息（总空间、可用空间和剩余空间）
获取最后修改时间	std::filesystem::last_write_time	获取文件的最后修改时间
修改最后修改时间	std::filesystem::last_write_time	设置文件的最后修改时间

2. 文件管理：数据存储的钥匙

2.2 文件流操作

流是一种C++中处理**输入**和**输出**的**抽象机制**，用于表示数据的**有序序列**，让程序进行**读写**操作。

C++标准库提供了丰富的流类和函数，包括控制台输入输出流**<iostream>**、字符串流操作**<stringstream>**、文件流操作**<fsteam>**。
<fstream> 提供的三个类及其多种操作，以方便地操作文件的数据输入和输出。

std::ifstream类（输入文件流）

操作	描述	示例代码
打开文件	打开文件以供读取	std::ifstream input_file("example.txt");
读取文件	从文件中读取数据	std::getline(input_file, line);
检查是否打开	检查文件是否成功打开	if (!input_file) { std::cerr << "Unable to open file"; }
读取一行	读取文件中的一行	std::getline(input_file, line);
读取数据	使用流提取运算符读取数据	input_file >> data;
关闭文件	关闭文件	input_file.close();
检查EOF	检查是否到达文件末尾	if (input_file.eof()) { ... }

2. 文件管理：数据存储的钥匙

2.2 文件流操作

std::ofstream类（输出文件流）

操作	描述	示例代码
打开文件	打开文件以供写入	`std::ofstream outputFile("example.txt");`
写入文件	向文件中写入数据	`outputFile << "Hello, world!" << std::endl;`
检查是否打开	检查文件是否成功打开	`if (!outputFile) { std::cerr << "Unable to open file"; }`
写入数据	使用流插入运算符写入数据	`outputFile << data;`
关闭文件	关闭文件	`outputFile.close();`

std::fstream类（文件流）则包含了std::ifstream和std::ofstream的能力组合。

文件操作常用方式

方式	描述
std::ios::in	以读模式打开文件
std::ios::out	以写模式打开文件
std::ios::app	以追加模式打开文件，写入的数据会被追加到文件末尾
std::ios::trunc	如果文件存在则清空文件内容
std::ios::binary	以二进制模式打开文件

2. 文件管理：数据存储的钥匙

2.2 文件流操作

文件格式

文件格式，其实就是解析文件的一种编解码方式。

特性	文本格式（Text Mode）	二进制格式（Binary Mode）
数据表示	人类可读的字符序列	原始的字节序列
换行符处理	自动转换换行符（Windows：\r\n，Unix/Linux：\n）	不进行任何转换，按原样读写
EOF处理	某些系统自动处理EOF标记	没有特殊的EOF标记处理，按原样读写
应用场景	适用于纯文本文件（源代码、配置文件、日志文件等）	适用于二进制文件（图像、音频、可执行文件等）
示例	纯文本日志、CSV、TSV、JSON、XML、YAML	图像格式、音频格式、视频格式、可执行文件格式、压缩格式
编解码	ASCII、UTF-8、Unicode.....	JPEG、MP3、MP4、ZIP.....
<fstream>读写操作	不指定std::ios::binary	指定std::ios::binary

2. 文件管理：数据存储的钥匙

2.3 XML和JSON

XML

- **全称：可扩展标记语言/eXtensible Markup Language**
- **结构：**使用标签来描述数据的层次结构，比如

  `<person><name>John</name><age>30</age></person>`
- **特点：**可读性较强，支持命名空间和复杂的文档结构
- **应用场景：**通常用于配置文件、文档存储等需要明确结构和元数据的场合

JSON

- **全称：JavaScript对象表示法/JavaScript Object Notation**
- **结构：**使用键值对的方式来组织数据，例如 `{ "name": "John", "age": 30 }`
- **特点：**紧凑且易于解析，支持数组和嵌套对象，是现代Web应用中常用的数据格式
- **应用场景：**适合于Web服务的数据交换、API通信等场景

2. 文件管理：数据存储的钥匙

2.3 XML和JSON

C++中操作XML文件

在C++中操作XML文件，通常使用第三方库，如**TinyXML2**、**RapidXML**。

TinyXML2 Github Repo: https://github.com/leethomason/tinyxml2

Example:

- https://github.com/leethomason/tinyxml2/blob/master/xmltest.cpp#L152
- https://gist.github.com/felton/5530029

C++中操作JSON文件

在 C++ 中操作JSON文件，通常使用第三方库，如**nlohmann/json**、**RapidJSON**。

nlohmann/json Github Repo: https://github.com/nlohmann/json

Example:

- https://json.nlohmann.me/api/basic_json/dump/#examples

除了XML、JSON这样的文本文件，还有图片、音频、视频、压缩包等各种格式的文件，也都有相应的C++第三方库。

2. 文件管理：数据存储的钥匙

2.4 Protocol Buffer

Protocol Buffers（protobuf）是一种语言无关、平台无关、可扩展的序列化数据结构的格式，由Google开发并开源。它能够高效地序列化结构化数据，适合用于数据存储、通信协议等场景。

优点

- **高效性：** 生成的序列化数据比XML、JSON紧凑，解析速度更快。
- **可扩展性：** 可以向已存在的消息类型添加新字段，而不会破坏旧程序的兼容性。
- **跨语言支持：** 支持多种编程语言，包括 C++、Java、Python 等。

资料

- Github仓库：https://github.com/protocolbuffers/protobuf
- Documentation： https://protobuf.dev/
- Protobuf in C++： https://protobuf.dev/getting-started/cpptutorial/
- 如何在C++工程中使用Google Protocol Buffer ：
  https://www.neohugh.art/%E5%BA%93/protobuf/protobuf-C/

2. 文件管理：数据存储的钥匙

2.4 Protocol Buffer

protobuf相对于XML和JSON更为复杂：

1. **学习曲线陡峭：** 需要理解 .proto 文件的定义、编译器的使用以及生成的代码如何与应用程序集成。
2. **工具链依赖多：** 需要依赖特定的编译器和代码生成工具（如 protoc），
3. **数据结构定义繁琐：** protobuf 的.proto文件需要明确定义数据结构，没有XML和JSON自由。

那为什么还有大量支持者使用protobuf呢：

1. **性能优势：** protobuf 序列化后的数据体积小，解析速度快，比起 XML 和 JSON 有更好的性能表现。
2. **跨语言支持：** 可以直接在多种语言之间直接使用。
3. **版本兼容性：** protobuf 支持向已存在的消息类型添加新字段而不破坏现有程序的兼容性，这使得数据结构的演进变得更为容易和安全，而XML和JSON就容易出现消息的兼容性问题。

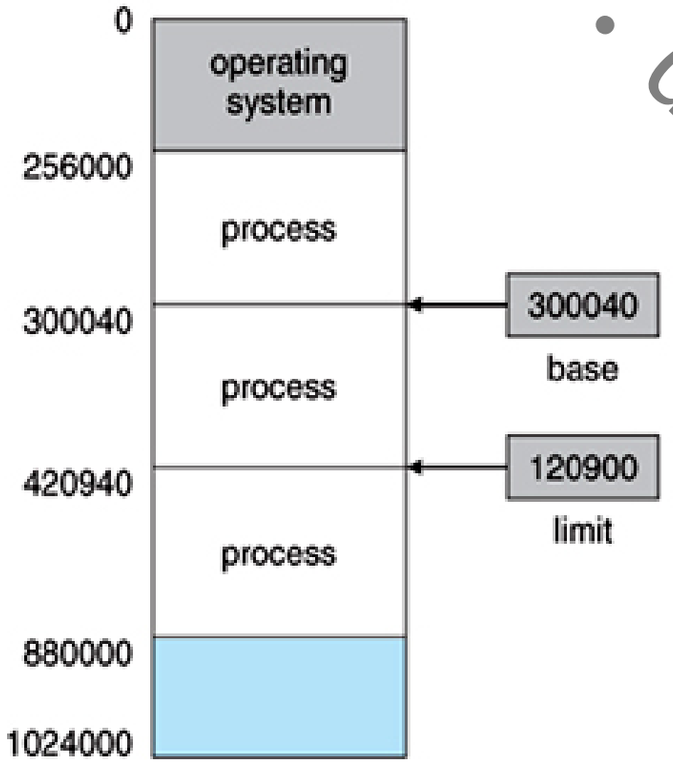
复杂的设计和使用，带来更多使用上的优势。所以，Apollo很多地方都会使用Protobuf。

内存管理：优化系统的血脉

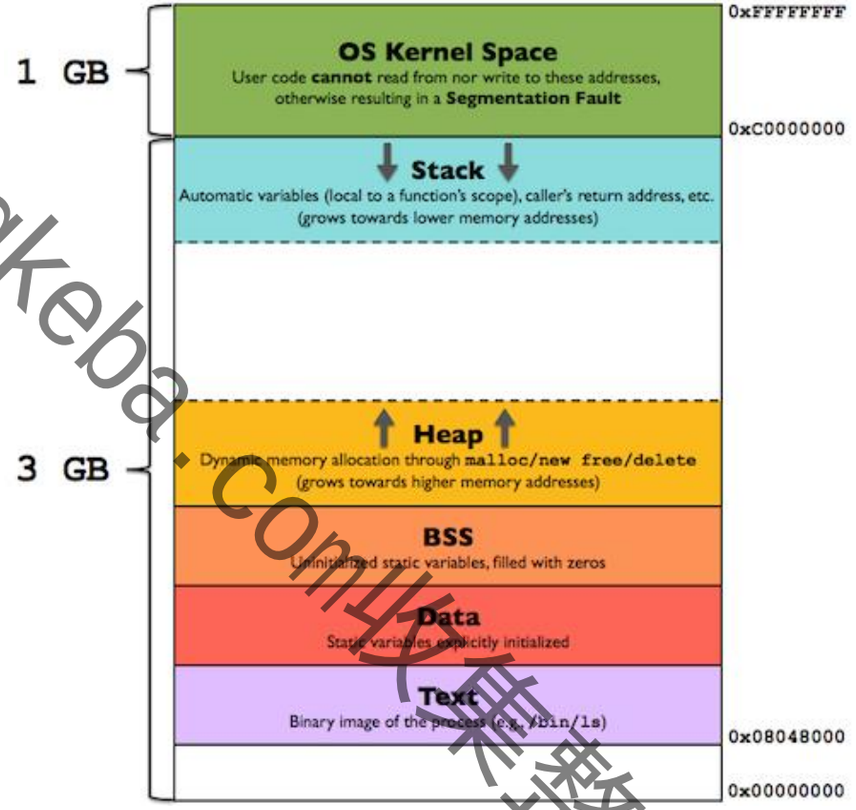
3. 内存管理：优化系统的血管

3.1 内存布局

利用虚拟内存映射的机制，操作系统为每个进程独立分配一套虚拟地址空间，让每个进程以为自己可以占据所有、连续的内存空间。



Memory Layout in Operating System, link



Memory Layout in Process, link

3. 内存管理：优化系统的血管

3.1 内存布局

进程里的内存布局

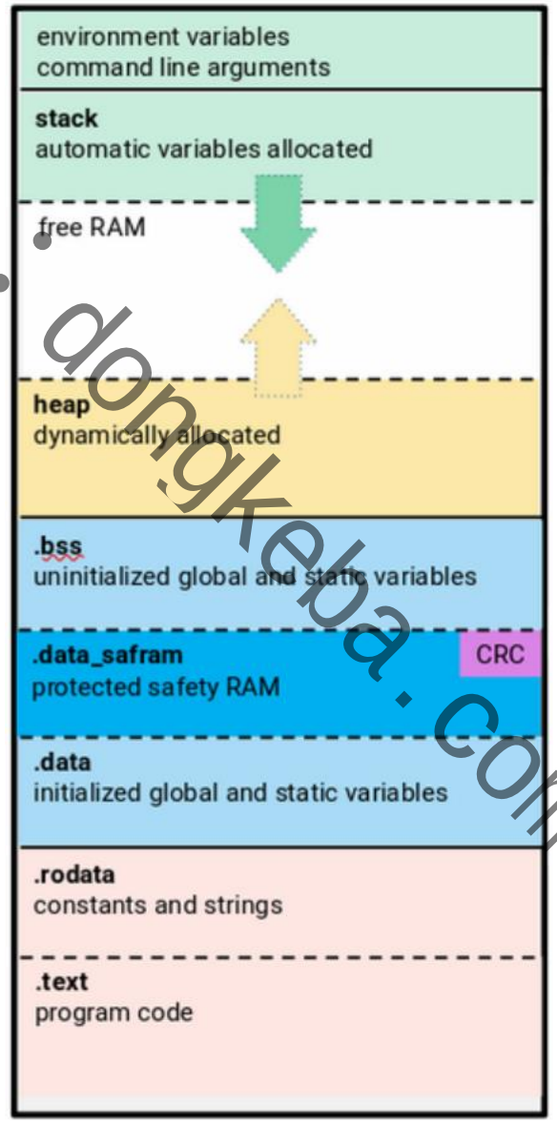
内存区域	内容	特点	保护和管理
保留区 **Reserved Area**	系统保留的内存区域	通常不供用户进程直接使用	由操作系统管理
栈 **Stack**	存储函数调用时的局部变量、参数和返回地址	向下增长，每次函数调用分配新的栈帧，返回时释放栈帧	通常有栈溢出保护机制，防止内存覆盖
内存映射段 **Memory Mapped Segment**	映射文件或设备到内存地址空间（如共享库、内存映射文件）	可读写或只读，取决于映射对象的权限	由操作系统管理，根据权限设置读写保护
堆 **Heap**	用于动态分配内存	向上增长，大小可变	由库函数和操作系统共同管理，涉及复杂的内存管理算法
.bss未初始化数据段 **Block Started by Symbol**	存放未初始化的全局变量和静态变量	程序开始时自动初始化为零	由编译器和操作系统管理
.data初始化数据段 **Initialized Data Segment**	存放已初始化的全局变量和静态变量	程序开始时已被赋予初始值	由编译器和操作系统管理
.text代码段 **Text Segment**	存放程序的可执行代码（机器指令）	只读，通常共享	只读保护，防止程序修改其自身代码

多数内存段是已经确定好、无法随意操作的内存段，比较灵活的就是栈区和堆区。

3. 内存管理：优化系统的血管

3.1 内存布局

进程里的内存布局



进程内存布局，link

3. 内存管理：优化系统的血管

3.1 内存布局

栈内存

- **分配**
  - 栈内存通常不需要显式申请，栈的增长是自动的，由编译器生成的代码来管理。
  - 在程序启动时，操作系统会为进程分配一个默认大小的栈空间。
- **工作机制**
  - 当函数被调用时，栈自动向下增长（在大多数架构中）。
  - 函数的局部变量、参数、返回地址等信息被压入栈中。
  - 函数返回时，这些信息自动从栈中弹出。
- **特点**
  - 分配和释放非常快速，因为只需要移动栈指针。
  - 大小通常是固定的，在程序启动时确定。
  - 空间较小，一般从几KB到几MB不等，具体取决于操作系统和编译器设置。
- **栈溢出**
  - 如果栈空间用尽（例如，由于深度递归），会发生栈溢出错误。
  - 一些系统允许动态增加栈大小，但这不是普遍做法。
  - 栈溢出之后，会触发异常，由操作系统接管处理。

栈内存资源的使用是**自动的**、**快速的**、**隐式的**，不需要程序员显式操作，也不涉及向操作系统主动请求的过程。

3. 内存管理：优化系统的血管

3.1 内存布局

堆内存

C和C++允许开发者手动分配和释放**堆内存**，C使用malloc()和free()函数，C++使用new和delete操作符。这种灵活性使得程序可以精确地控制内存使用。

应用程序向操作系统申请堆内存（动态内存）的过程通常包括以下几个主要步骤：

1. **应用程序发起请求：** 程序通过API调用（如C语言中的malloc()或C++中的new）请求内存。
2. **系统调用转换：** 这个请求被转换为对应的系统调用（如Linux中的brk()或mmap()）。
3. **内核接管：** 控制权转移到操作系统内核。
4. **内存管理器检查：** 内核的内存管理器检查可用的物理内存。
5. **页表更新：** 如果有足够的内存，内核更新进程的页表，建立虚拟地址到物理地址的映射。
6. **物理内存分配：** 内核可能会立即分配物理内存，或者使用“延迟分配”策略。
7. **返回结果：** 内核将控制权返回给应用程序，并提供分配的内存地址。
8. **应用程序使用：** 应用程序获得内存地址，可以开始使用这块内存。

这个过程涉及到虚拟内存、页表、物理内存管理等复杂的操作系统概念，实际的过程可能因操作系统和硬件架构的不同而有差异。

所以，堆内存资源的使用是**手动的、耗时的、显式的、需要OS介入**，所以堆内存最需要程序员主动关注的，也是本节讨论**内存管理**的主要对象。

3. 内存管理：优化系统的血管

3.2 内存使用常见问题

常见问题

问题	描述	金融中的对应现象	描述
内存泄漏 **Memory Leak**	程序分配了内存但没有正确释放，导致内存使用量不断增加	**贷款诈骗** **Loan Fraud**	借款人在借钱、完成项目、清算之后，没有将贷款归还银行或金融机构，而是卷款跑路
非法内存访问-悬空指针 **Dangling Pointer**	指针指向的内存已经被释放，但指针没有被重置，继续被使用	**资产虚增** **Asset Overstatement**	资产已经被冻结或清算，但仍然宣称拥有大量资产
非法内存访问-野指针 **Wild Pointer**	未初始化的指针或指向不合法内存区域的指针	**投资欺诈** **Investment Fraud**	声称要大笔投资，实际上并未出资
非法内存访问-缓冲区溢出 **Buffer Overflow**	访问数组或内存块时超出了其边界，导致未定义行为或内存破坏	**违法占用** **Illegal Occupation**	占用本不属于自己的资金
双重释放 **Double Free**	对同一块内存进行多次释放，可能导致程序崩溃或未定义行为	**重复还款** **Repeated Repayment**	已经偿还的一笔债务或投资，错误地再次尝试偿还
内存对齐问题 **Memory Alignment**	对齐不正确的内存访问可能导致性能下降或未定义行为	**会计对账不一致** **Accounting Discrepancies**	财务报表中的数据未正确对齐或记录，导致财务状况误报和潜在审计问题
栈溢出 **Stack Overflow**	函数调用层次过深或分配过大的局部变量导致栈空间耗尽，导致程序崩溃	**市场泡沫** **Market Bubble**	市场参与者反复加杠杆推高资产价格，资产价格过度上涨，等到再也没有新资金和杠杆空间，市场突然崩溃

3. 内存管理：优化系统的血管

3.2 内存使用常见问题

解决办法

问题	原因	代码示例	直接解决方案
内存泄漏 **Memory Leak**	忘记释放动态分配的内存	int* p = new int[10]; // 忘记 delete[] p; 造成内存泄漏	在适当位置添加 delete

非法内存访问-悬空指针
Dangling Pointer
	释放动态内存后继续使用该指针；返回指向局部变量的指针	int* p = new int;
		delete p;
		*p = 10; // p指向已释放的内存
			在释放内存后将指针设为nullptr，避免返回指向局部变量的指针，并最好不再使用

非法内存访问-野指针 **Wild Pointer**	声明指针但未初始化；访问已经释放的内存	int* p; *p = 10; // 野指针，p未初始化	在声明指针时初始化，使用 nullptr或有效内存地址
非法内存访问-缓冲区溢出 **Buffer Overflow**	数组索引超出范围；字符串操作（如 strcpy）时未检查目标缓冲区大小	int arr[10]; arr[10] = 5; // 缓冲区溢出，访问超出数组范围	严格检查数组边界； 使用包含长度检查的安全函数如 strncpy

双重释放
Double Free
	逻辑错误导致同一指针被释放	int* p = new int;
		delete p;
		delete p; // 双重释放，同一块内存被释放两次
			在释放内存后将指针设为 nullptr

内存对齐问题 **Memory Alignment**	手动分配内存时不正确地使用结构体，导致结构体成员未对齐，访问性能较低	struct A { char a; int b; }; A* p = new A;	使用编译器提供的对齐指令或标准库函数，如 posix_memalign
栈溢出 **Stack Overflow**	递归函数未正确终止；局部变量分配过多内存	void func() { func(); } int main() { func(); } // 递归过深导致栈溢出	为递归函数设置递归边界，限制递归深度，避免在栈上分配过大内存

3. 内存管理：优化系统的血管

3.2 内存使用常见问题

类比

- 其他内存区域，是政府和家庭提供的生活最低保障；
- 动态（堆）内存区域，是金融市场提供的融资手段。

特性	金融市场	内存管理
严格监管/管理	防止非法交易和欺诈，保障市场稳定；缺乏灵活性，降低创新和效率	防止内存错误和安全漏洞，保障程序稳定性；增加复杂性和性能开销
自由市场/宽松管理	提高灵活性和创新性，快速响应变化；增加金融泡沫和欺诈风险	提高开发效率和系统灵活性；增加内存错误和安全漏洞风险

适当的内存使用自由度是大型程序所必需的，但也由此带来前述的各种问题。

程序员可以靠自己的经验和精力去尽量避免或解决这些问题，但这会花费大量的时间和精力，导致程序开发的低效。而C++提供了更有效的方案。

3. 内存管理：优化系统的血管

3.3 再探RAII

C++编译器在背后做的工作

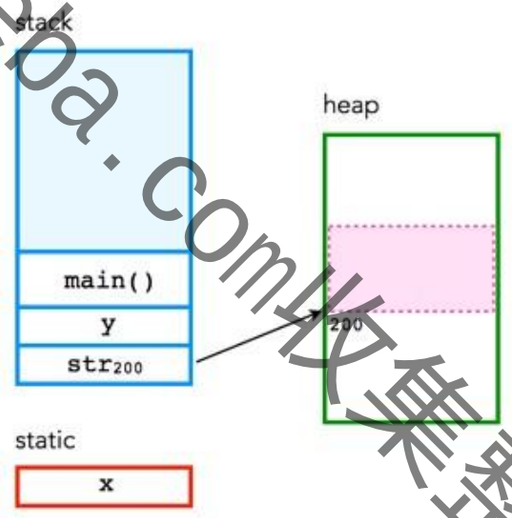
C++编译器生成的代码会自动为对象添加如下操作：

- 自动分配对象所需的（栈）内存空间。
- 调用构造函数，进行对象的初始化。构造可以选择默认构造、带参数构造、复制构造等多种方式。
- 当其作用域结束时，编译器会在对象的作用域末尾自动调用析构函数。
- 自动释放对象所用的（栈）内存空间。

另外，堆内存无法直接使用，而是通过栈内存上的指针间接去使用：

```
A* a = new A;
char[] s = new char[200];
```

a和s指针本身是栈上的对象，但指向堆上的数据的首地址。



Stack memory pointer to heap data, link

3. 内存管理：优化系统的血管

3.3 再探RAII

回顾RAII

RAII（Resource Acquisition Is Initialization）翻译为资源获取即初始化，由c++之父Bjarne Stroustrup提出，是C++中一种重要的编程思想，它的本质是利用对象的生命周期来管理资源的获取和释放。

RAII的核心思想是：在对象的构造函数中获取资源，在对象的析构函数中释放资源，利用对象的生命周期来确保资源的正确管理，从而避免资源泄漏，提高程序的健壮性。

- **用类对象来封装资源**：设计一个类封装资源和对资源的各种操作，在使用时定义一个该类的对象。
- **资源获取即初始化**：RAII 的核心概念是将资源的获取操作绑定到对象的构造过程中。当对象被创建时，它的构造函数会负责获取所需的资源，如内存、文件句柄、互斥锁等。
- **资源释放即析构**：对象的析构函数负责释放在其生命周期内所获取的资源。无论对象因何种原因（正常结束、异常、早期返回等）被销毁，都会确保资源的正确释放，从而避免资源泄漏。
- **依赖对象生命周期**：RAII 依赖于对象的生命周期管理资源。只要对象存在，资源就被持有；一旦对象销毁，资源就被释放。这种依赖生命周期的方式使得资源管理变得自动化和安全。

3. 内存管理：优化系统的血管

3.3 再探RAII

C++ STL广泛使用RAII

利用C++的OOP编程范式和RAII思想，将C语言库或系统库封装，得到STL的<filestream>、<memory>等库，可以更方便地进行C++编程。

1. **智能指针。**参考3.6.3节。
2. **容器。std::vector, std::list, std::map 等：**这些容器类在内部管理动态分配的内存，使用 RAII 来确保内存的安全管理。当容器对象超出作用域时，它们会自动调用析构函数，释放容器内元素所占用的内存。
3. **文件流。std::ifstream, std::ofstream, std::fstream 等：**这些类用于文件的读写操作，它们的构造函数负责打开文件，析构函数负责关闭文件。这种设计确保在文件流对象超出作用域时，文件资源会被正确关闭。
4. **互斥量和锁。std::mutex, std::lock_guard, std::unique_lock 等：**用于多线程编程的类利用 RAII 来管理互斥量的锁定和解锁。std::lock_guard 和 std::unique_lock 对象在构造时锁定互斥量，在析构时自动解锁，确保了在作用域结束时互斥量被正确释放。
5. **……**

把对资源的合理释放操作封装到类中，利用RAII自动完成，就达到了对内存的合理使用。
另外，STL容器通过迭代器（iterator）避免了非法访问，STL的算法也尽量避免其他安全问题。

3. 内存管理：优化系统的血管

3.3 再探RAII

代码示例

```
class fstream {
 private:
        std::string filename_;

 public:
    fstream(const std::string& filename) : filename_(filename) {
        if (!fopen(filename_), 'w') {
            throw std::runtime_error("Failed to open file");
        }
        std::cout << "File opened successfully\n";
    }

    ~fstream() {
        if (is_open(filename_)) {
        fclose(filename_);
        std::cout << "File closed\n";
        }
    }

    void write(const std::string& data) {
    fwrite(filename_, &data, data.size());
    }
};
```

```
void WriteMore(const fstream& file_stream) {
    file_stream.write("Write in the middle.");
}

int main() {
    try {
        fstream fs("test.txt");
        fs.write("Hello, RAII!");
        WriteMore(fs);
    } catch (const std::exception& e) {
    std::cerr << "Exception: " << e.what() << std::endl;
        return -1;
    }

    return 0;
}
```

3. 内存管理：优化系统的血管

3.4 内存池

内存管理和银行系统

把计算机中的内存和银行系统中的资金做类似：

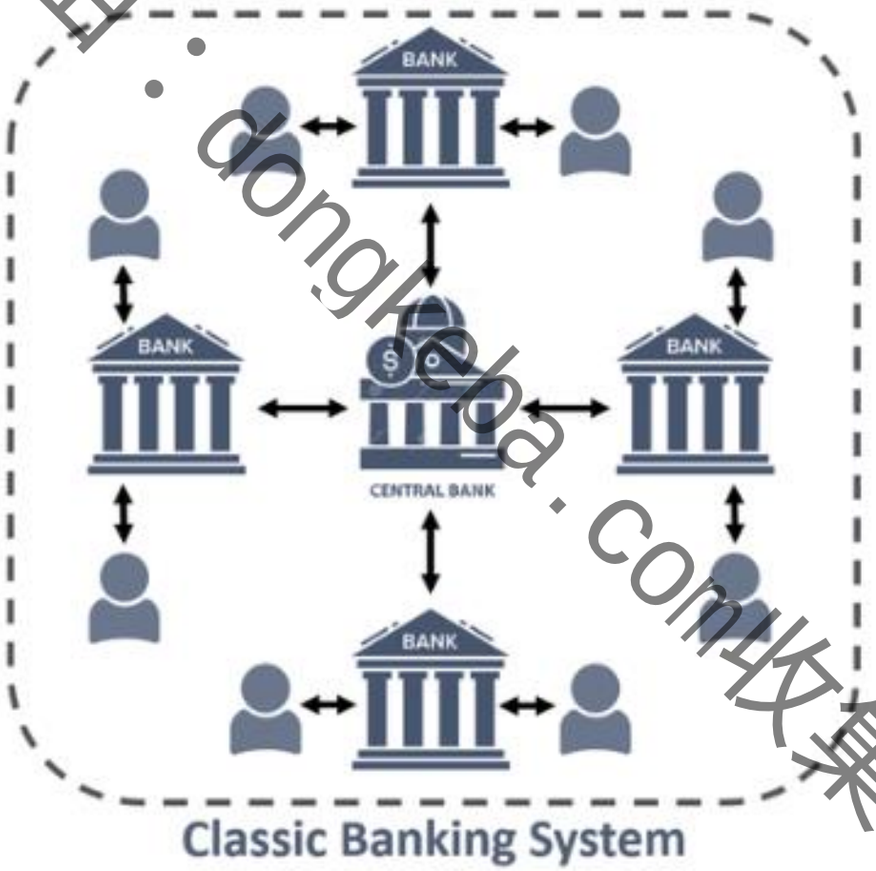
- 应用启动，需要申请内存，相当于银行系统中企业和个人申请贷款；
- 应用退出，需要释放内存，相当于银行系统中企业和个人归还贷款。

内存管理概念	类比银行系统概念	特点和功能
内存	货币	
应用程序	企业和个人	• 资源的最终使用者 • 有多样化的需求 • 效率和成本敏感
内存条硬件	国家和中央银行	• 控制总量 • 制定基本政策 • 决定系统资源的基础
操作系统的内存管理	各大商业银行	• 规范化管理 • 面向广泛用户 • 标准化分配机制 • 相对稳定和安全
内存池	民间借贷、影子银行	• 更高的灵活性，资源使用效率高 • 针对特定需求 • 效率更高 • 潜在风险也更高 • 在标准体系之外运作

3. 内存管理：优化系统的血管

3.4 内存池

内存管理和银行系统



3. 内存管理：优化系统的血管

3.4 内存池

概念

内存池（Memory Pool）是一种用于内存管理的技术，旨在提高内存分配和释放的效率。

1. **预分配**：在程序开始时或在需要时，一次性向操作系统申请并分配得到一大块连续的内存。这块内存被称为内存池。
2. **分配策略**：内存池内部实现一个分配策略，用于从内存池中分配小块内存。这种策略通常比系统的内存分配器（如 malloc 或 new）更高效。
3. **回收策略**：内存块被释放时，不是直接交还给系统，而是返回到内存池中，以备将来再次使用。
4. **内存碎片**：由于内存池管理的是一大块连续内存，内存碎片的产生几率较低，有助于提高内存利用率。

适用场景示例

1. **游戏开发**：在游戏中，频繁创建和销毁对象（如子弹、敌人等），需要快速且高效的内存管理。
2. **网络服务器**：处理大量客户端请求时，需要快速创建和释放响应对象，需要快速分配和释放内存，以应对高并发。
3. **嵌入式系统**：资源有限且需要高效利用内存，同时要求严格的实时性能。

内存池在需要高效、可预测的内存管理的场景中具有显著优势。

3. 内存管理：优化系统的血管

3.4 内存池

优势

与直接向操作系统申请内存相比，内存池的优势包括：

1. **提高分配和释放速度。**操作系统的内存分配器（如malloc和free）通常会进行复杂的操作逻辑来管理内存，包括寻找合适大小的空闲块、维护内存块的元数据等。这些操作可能涉及多次系统调用，导致性能开销较大；而内存池通过预先分配一大块内存，并在其内部进行简单的分配和释放操作，可以显著加快内存分配和释放的速度。内存池的分配和释放操作通常只需简单的指针操作，减少了时间开销。
2. **减少内存碎片。**操作系统的内存管理器需要处理不同大小的内存块，长时间运行后可能会导致内存碎片化，利用率较低；内存池通过管理一大块连续的内存，可以更有效地减少内存碎片，提高内存利用率。
3. **简化内存管理逻辑。**在某些特定应用场景下，内存池可以简化内存管理的逻辑，使开发者更容易维护代码，减少内存泄漏和其他内存管理错误的风险。

如果申请和释放内存的频率不高、需要内存不多，就没必要强行使用内存池，否则反而增加程序的复杂性和维护难度，降低可移植性。

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

引用、复制和移动的区别

概念	定义	特点	直观理解
引用 Reference	对象或数据的别名，不创建副本	共享原始对象的内存空间，修改影响原始对象	多个人共同享有同一个苹果
复制 Copy	创建对象或数据的完整副本	新对象独立于原始对象，修改不影响原始对象	一个人把一个苹果复制成两个苹果，把一个给另一个人
移动 Move	转移对象或数据的所有权	避免不必要的复制操作，提高效率	一个人把苹果直接转移给另一个人

之前讲过浅拷贝、深拷贝。本质上，**浅拷贝就是引用，深拷贝就是复制**。

引用、复制和移动，都是直接、浅显的概念，也是程序员在资源管理中很可能遇到的情况，C++语言应该给程序员提供这3个选择。

- 但是C++11之前，C++只提供了**引用**和**复制**的概念和语义，并没有明确提供**移动语义（Move Semantics）**。只能将就使用**引用**或**复制**。
- 到了C++11，C++终于提供了移动的语义，程序员从而可以使用移动来提高性能。

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

移动语义的缺失

C++11 之前，只有 copy 语意，这对于极度关注性能的语言而言是一个重大的缺失。那时候程序员为了避免性能损失，只好采取规避的方式。比如：

```
std::string str = s1;
s += s2;
```

这种写法就可以规避不必要的拷贝。

而更加直观的写法：

```
std::string str = s1 + s2;
// or std::string str = s1.append(s2);

// 对于基础类型，移动语义的意义很小
// 假设一个picture有100M，则时间消耗更明显
Picture mix_picture = picture1.Mix(picture2);
```

则必须忍受一个 s1 + s2 所导致的中间 **临时对象**temp_str到str的拷贝开销。即便那个中间临时对象随着表达式的结束，会被销毁（更糟的是，销毁所伴随的资源释放，也是一种性能开销，或者说时间消耗）。

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

移动语义的缺失

对于 move 语义的急迫需求，到了 C++11 终于被引入。其直接的驱动力很简单：在构造或者赋值时，如果等号右侧是一个中间临时对象，应直接将其占用的资源直接 move 过来。

但问题是：怎么让一个类的构造函数或者赋值运算符识别到右边是一个临时变量，或者它主动想把资源所有权转移出来？

```
Picture picture(temp_picture);
Picture picture = temp_picture;
// Copy or Move? How to recognize it automatically?
```

答案是：1. 为变量引入右值引用，2. 为类增加移动构造函数和移动赋值运算符。

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

右值引用

左值和右值的区别

- **左值（Lvalue）：**具有持久命名的对象，可以取地址。例如变量、数组元素、对象成员等。
- **右值（Rvalue）：**临时对象或字面量，通常是不能取地址的。例如字面量、表达式的返回值、临时对象等。

在C++11之前，C++中只有左值引用（T&），用于引用左值。

在C++11中引入了新的引用类型——**右值引用（Rvalue Reference）**，其语法形式为 T&&，其中 T 是类型。右值引用主要用于实现移动语义，其特点包括：

- 可以绑定到临时对象（右值），例如临时创建的对象或表达式的结果。
- 允许区分左值（Lvalue）和右值（Rvalue），以便在编译器级别上优化资源的转移而非复制。

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

移动语义

移动语义是通过**移动构造函数（Move Constructor）**和**移动赋值运算符（Move Assignment Operator）**来实现的。它们的作用是使得对象在资源所有权转移时，可以高效地将已分配的资源（如动态分配的内存、文件句柄等）从一个对象转移到另一个对象，而不是进行深度复制。

移动构造函数（Move Constructor）允许从一个临时对象（右值引用）中"窃取"资源，避免额外的资源复制。通常的形式为：

```
class Picture {
public:
    // 复制构造函数
    Picture(const Picture& other) { // 左值引用
        // 资源复制逻辑，将 other 的资源复制到this对象中
    }

    // 移动构造函数
    Picture(Picture&& other) { // 右值引用
        // 资源转移逻辑，将other的资源转移到 this 对象中
        // 由于把other的资源拿走了，所以要把other内部资源的指针置为空，也就是修改other，所以other不为const
    }
};
```

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

移动语义

移动赋值运算符（Move Assignment Operator）类似于移动构造函数，但是用于对象已经存在时的资源转移操作。形式为：

```
class Picture {
public:
    // 赋值运算符
    Picture& operator=(const Picture& other) { // 左值引用
        ...
    }

    // 移动赋值运算符
    Picture& operator=(Picture&& other) { // 右值引用
        if (this != &other) {
            // 资源转移逻辑，将 other 的资源转移到 this 对象中
            // 同理，other不是const
        }
        return *this;
    }
};
```

所以前面的问题就有了答案：

- 如果temp_picture是**左值（有地址的持久对象）**，编译器和CPU就会帮我们自动匹配到**复制构造函数**和**赋值运算符函数**，按照**复制**的逻辑去操作；
- 如果temp_picture是**右值（临时对象或字面量）**，编译器和CPU就会帮我们自动匹配到**移动构造函数**和**移动赋值运算符函数**，按照**移动**的逻辑去操作。

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

std::move()函数

在实际使用中，有时想**明确地**、**主动地**让左值对象发生**移动**操作，可以使用 std::move() 函数将对象强制转换为右值引用。例如：

```
MyClass obj1;
MyClass obj2 = std::move(obj1);  // 使用移动构造函数将 obj1 的资源转移到obj2中
```

std::move() 将 obj1 转换为右值引用，从而可以调用移动构造函数来实现资源的转移。

Std::move()的实现

```
template <typename T>
typename remove_reference<T>::type&& move(T&& t) noexcept {
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

std::move 是一个模板函数，接受一个通用引用 T&& 作为参数，将参数 t 转换为一个右值引用类型。
其实，std::move()并没有真正去*move*，而是*make it moveable*；真正的move还是发生在移动构造函数和移动赋值运算符函数里。
Example： simple-vector, st_boundary_mapper

3. 内存管理：优化系统的血管

3.5 移动语义和右值引用

注意事项

- 移动不是什么都不复制，而是只复制资源的指针，不复制资源主体。
- **什么时候使用移动？**
  - **大数据结构**：当你的类包含大数据结构（如大型数组、字符串、容器等）时，使用移动语义可以避免昂贵的深拷贝。对于基础数据类型（int, double等），没有必要使用移动语义，因为数据量小，也没有移动构造函数和移动赋值运算符。
  - **资源管理**：当你的类管理资源（如动态分配的内存、文件句柄、网络连接等）时，移动语义可以更高效地转移这些资源的所有权。
- 原来的对象被 **移动** 操作之后，需要**避免再使用它**。因此虽然对象本身还在，但是它内部的资源已经被转移了，再去使用这个对象会发生未定义行为，可能引发程序错误。
- C++ STL的各个容器（如std::vector, std::string）已经广泛地实现、支持了移动语义，所以我们可以高效地进行移动操作。

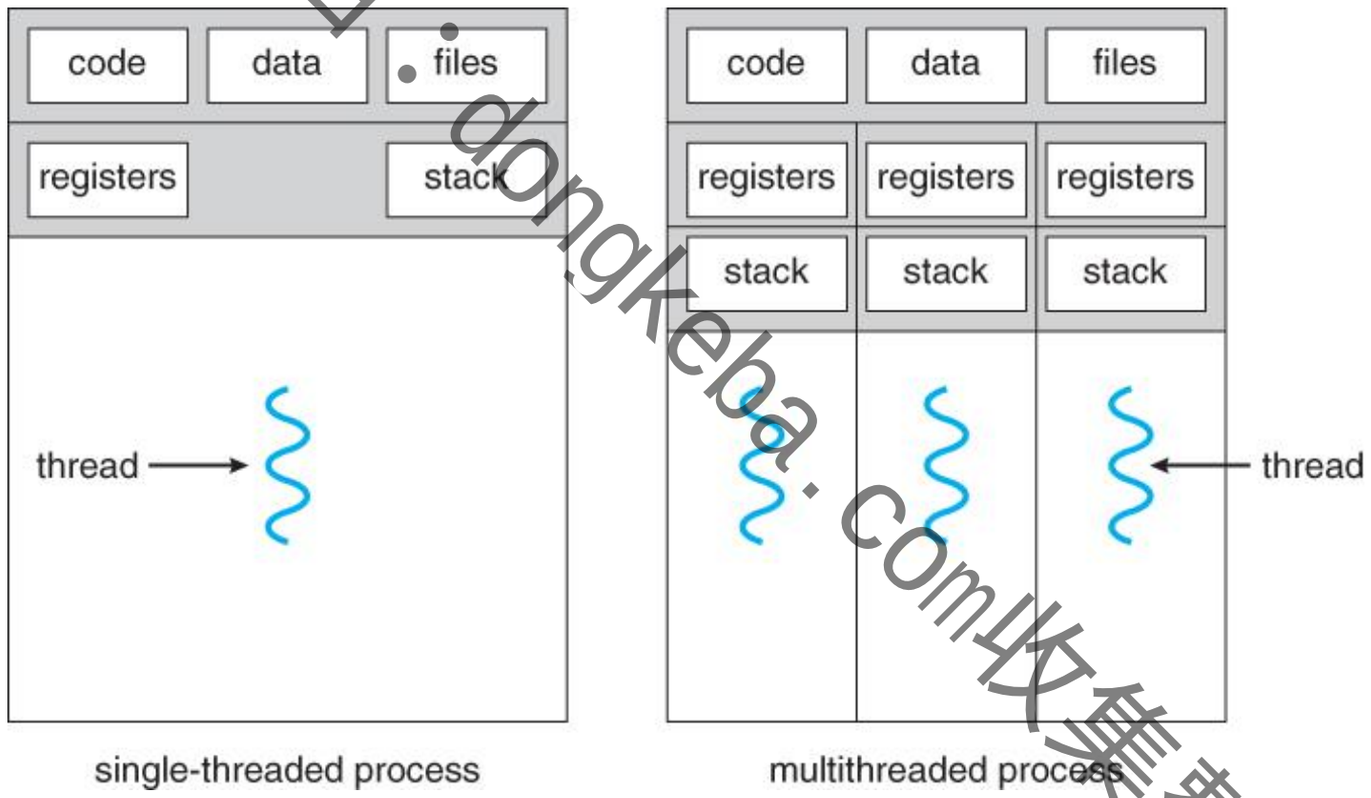
4

多线程管理：梳理错综复杂的世界

4. 多线程管理：梳理错综复杂的世界

4.1 多线程

C++中的多线程编程是指在一个程序中同时执行多个线程，以提高程序的性能或响应速度。多线程编程可以用来执行并行任务，处理I/O操作或其他需要异步处理的任务。



Single-threaded and multithreaded processes, link

C++11标准及以后版本提供了提供了标准库来支持多线程编程，主要包括<thread>、<mutex>、<condition_variable>等头文件。

4. 多线程管理：梳理错综复杂的世界

4.2 创建和释放线程

创建

在C++11中，可以使用std::thread类来创建线程。一个线程对象在创建时会启动一个新线程来执行指定的函数或可调用对象。

```cpp
#include <iostream>
#include <thread>

void DoTask() {
    std::cout << "Hello from a new thread!" << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(10));
}

int main() {
    std::thread thread1(DoTask); // 创建一个线程并运行DoTask函数
    // Do something in main thread
    thread1.join(); // 等待线程t在10s后完成
    return 0;
}
```

4. 多线程管理：梳理错综复杂的世界

4.2 创建和释放线程

结束

- join()：等待线程完成。如果线程已经完成，则立即返回。通常在创建线程后需要调用join，以确保主线程等待子线程完成。
- detach()：将线程与当前的线程对象分离，让线程在后台运行。分离后的线程独立运行，主线程不再等待其完成。

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

void DoTask() {
    std::cout << "Hello from thread!" << std::endl;
}

int main() {
    std::thread thread1(DoTask);
    thread1.detach(); // 将线程分离，允许其在后台运行
    std::this_thread::sleep_for(std::chrono::seconds(1)); // 等待一段时间，确保后台线程有机会运行
    return 0;
}
```

所以，其实使用多线程很简单。困难的是多线程并发之后，可能存在的冲突问题。

4. 多线程管理：梳理错综复杂的世界

4.3 线程互斥

std::mutex用于保护共享数据，确保同一时刻只有一个线程访问数据。

https://cplusplus.com/reference/mutex/

https://en.cppreference.com/w/cpp/thread/mutex

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

std::mutex mtx;

void DoTask(int id) {
    std::lock_guard<std::mutex> lock(mtx); // 加锁
    std::cout << "Thread " << id << " is running." << std::endl;
}

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

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

    return 0;
}
```

4. 多线程管理：梳理错综复杂的世界

4.4 线程同步

多线程程序中，需要同步访问共享数据以避免数据竞争和不一致。C++11提供了多种条件变量<condition_variable>的同步机制，允许线程在条件满足时进行等待和通知，通常与mutex结合使用。

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

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void DoTask() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, []{ return ready; }); // 等待ready为true
    std::cout << "Thread is running." << std::endl;
}

int main() {
    std::thread t(DoTask);
    std::this_thread::sleep_for(std::chrono::seconds(1)); // 模拟一些操作
    {
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;
    }
    cv.notify_one(); // 通知一个等待的线程
    t.join();
    return 0;
}
```

4. 多线程管理：梳理错综复杂的世界

4.5 原子操作

除了使用互斥量和条件变量进行同步，C++标准库还提供了一些线程安全的数据结构，例如std::atomic。原子操作无需显式的锁，可以有效地进行简单的读写操作。

```cpp
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> count(0);

void increment() {
    for (int i = 0; i < 1000; ++i) {
        ++count;
    }
}

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

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

    std::cout << "Final count: " << count << std::endl;
    return 0;
}
```

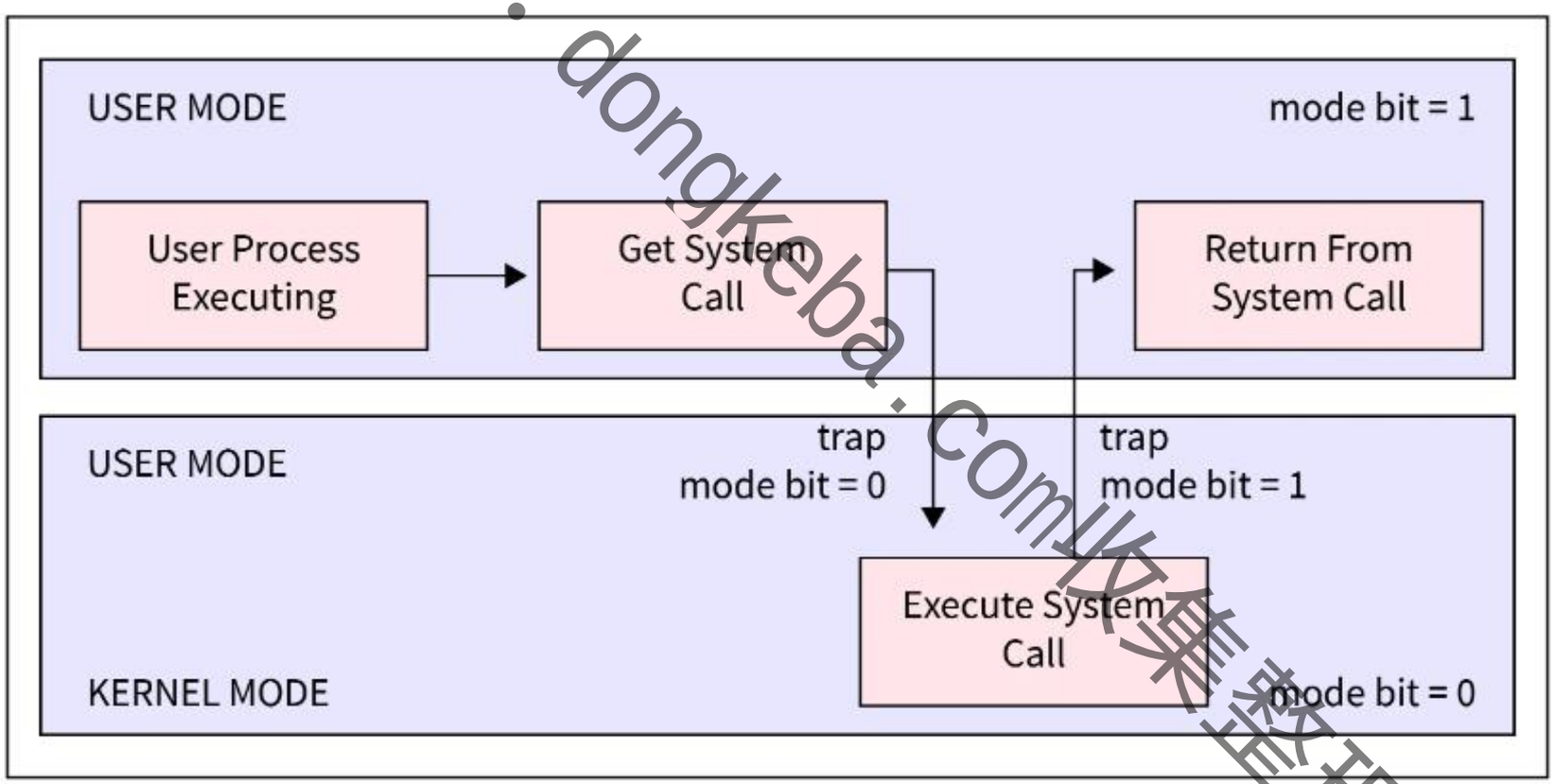
4. 多线程管理：梳理错综复杂的世界

4.6 无锁编程

无锁操作

只要是需要用到操作系统间接调度才能使用的资源，就会发生系统调用，性能就会比较差。

这就包括mutex、condition variable。



System Call from User Mode to Kernel Mode, link

4. 多线程管理：梳理错综复杂的世界

4.6 无锁编程

无锁操作

无锁操作（Lock-free operation）是指一种并发编程技术，使得多个线程在不使用锁（例如互斥锁、读写锁等）的情况下，可以安全地访问共享数据。无锁操作旨在避免锁带来的问题，如死锁、线程饥饿和高开销，从而提高系统的并发性能和响应性。

本质上，无锁操作也是有“锁”的，但这个“锁”无法再拆分，已经细微到一条CPU指令的级别，所以认为是“无锁”。

无锁操作通常依赖于一些底层的原子操作，这些操作在硬件级别上是原子的、不可拆分的、一条指令完成的，不会被中断或干扰。常见的原子操作有：

1. **原子读写**：对变量进行单独的读或写操作，不会被其他线程打断。
2. **比较并交换（Compare and Swap, CAS）**：CAS操作会比较一个变量的当前值是否等于一个预期值，如果相等则将变量的值更新为新值，否则不做任何改变。CAS操作通常由硬件直接支持。

特性	锁	无锁操作
基本原理	通过互斥机制，确保同一时刻只有一个线程访问共享资源	通过原子操作（如CAS）确保多个线程能安全地并发访问共享资源
依赖条件	依赖操作系统提供的锁机制和系统调用	不依赖操作系统的支持，但依赖硬件指令支持
性能开销	高，存在上下文切换和线程阻塞/唤醒的开销	较低，但可能存在自旋重试的开销
适用场景	低并发场景，资源竞争较少的情况	高并发场景，实时性要求高的情况

4. 多线程管理：梳理错综复杂的世界

4.6 无锁编程

CAS无锁队列

CAS无锁队列是一种基于**CAS（Compare-And-Swap，比较并交换）**操作实现的并发数据结构，用于高效地支持多线程并发操作。在CAS无锁队列中，所有的操作都是原子的，不需要使用传统的互斥锁来保护共享数据，从而避免了锁带来的性能开销和可能的竞争条件。

队列结构：CAS 无锁队列通常使用链表实现，其中节点（Node）包含数据和指向下一个节点的指针。

操作方法：

- **入队（push）：**通过 CAS 操作尝试将新节点添加到队列的尾部。如果尾部节点的指针已经被其他线程修改，CAS 操作可能会失败，此时需要重试直到成功。
- **出队（pop）：**通过 CAS 操作尝试将头节点从队列中移除，并返回其数据。如果头节点已被其他线程修改，CAS 操作同样会失败，需要重试直到成功。

Reference: https://github.com/rigtorp/awesome-lockfree

4. 多线程管理：梳理错综复杂的世界

4.7 线程池

线程池（Thread Pool）是一种多线程处理模式，它预先创建一定数量的线程，这些线程可以重复使用来执行任务，而不是每次需要时都创建新线程。
所以，**线程池**的设计思想和**内存池**类似：在操作系统和程序之间**增加一层**资源管理器，来更高效地管理**内存或线程资源**。

核心组件

- **线程池管理器（Thread Pool Manager）：** 负责管理线程池的生命周期、任务分配和线程的调度。线程池会在启动时预先创建一定数量的线程，这些线程在等待任务时处于空闲状态。
- **工作线程（Worker Threads）：** 实际执行任务的线程。线程池中的线程会从任务队列中取出任务并执行，执行完毕后线程不会被销毁，而是继续等待下一个任务。
- **任务队列（Task Queue）：** 用于存储等待执行的任务。当有新的任务需要执行时，这些任务会被放入一个任务队列中等待。

工作原理

- 当提交任务时，如果线程数小于核心线程数，创建新线程
- 如果线程数等于核心线程数，任务进入队列
- 如果队列已满，且线程数小于最大线程数，创建新线程
- 如果队列已满且线程数达到最大，执行拒绝策略

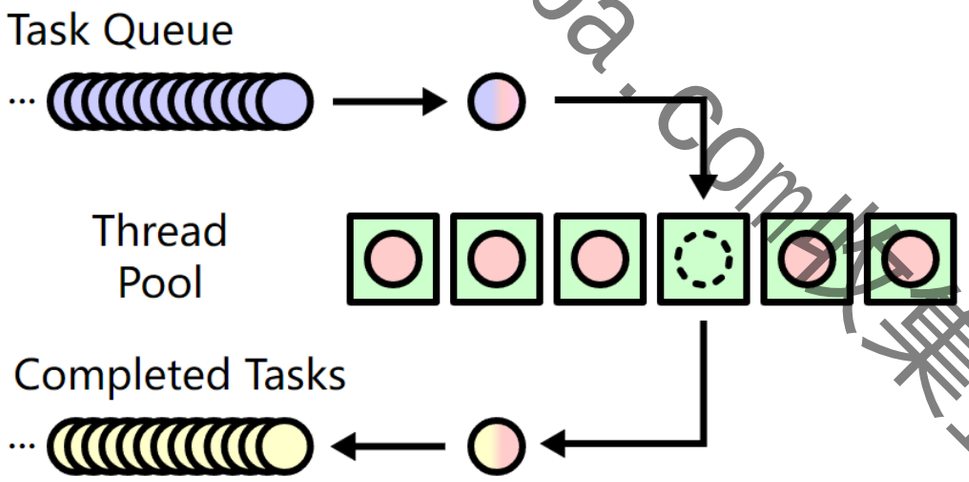
4. 多线程管理：梳理错综复杂的世界

4.7 线程池

收益

- **提高性能：**线程池预先创建线程，减少了频繁创建和销毁线程带来的系统开销，从而提高了系统的性能。
- **简化并发控制：**线程池管理任务的提交、执行和调度，使开发者专注于业务逻辑，而不用处理复杂的线程管理问题。
- **控制并发量：**通过配置线程池的大小，可以控制同时运行的线程数，从而实现限流效果，避免系统过载。

参考代码： https://github.com/ApolloAuto/apollo/blob/r8.0.0/cyber/base/thread_pool.h



Thread Pool, link

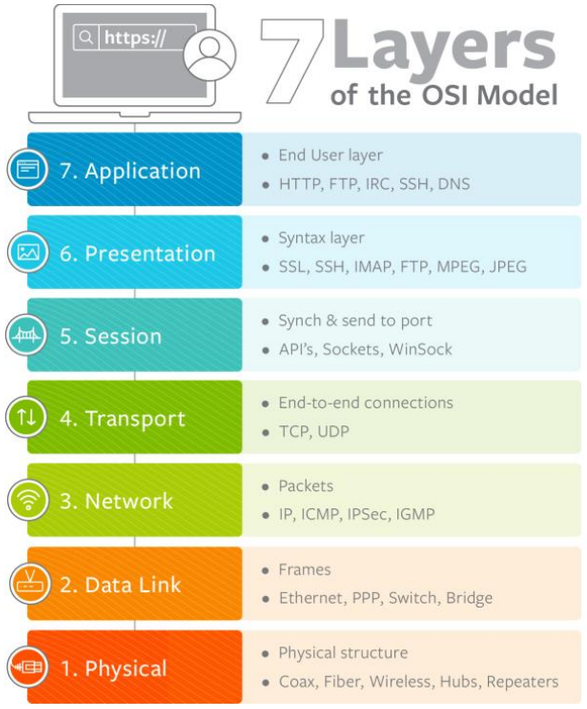
5

网络管理：连接信息高速公路

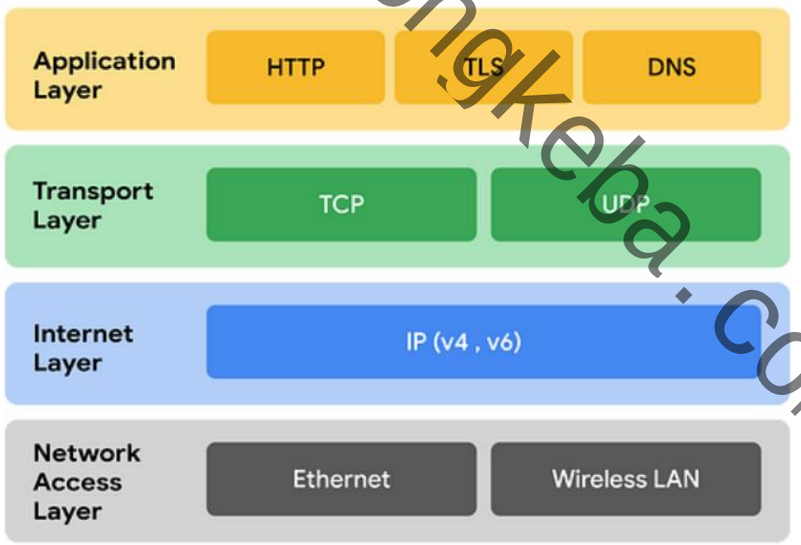
5. 网络管理：连接信息高速公路

5.1 核心概念

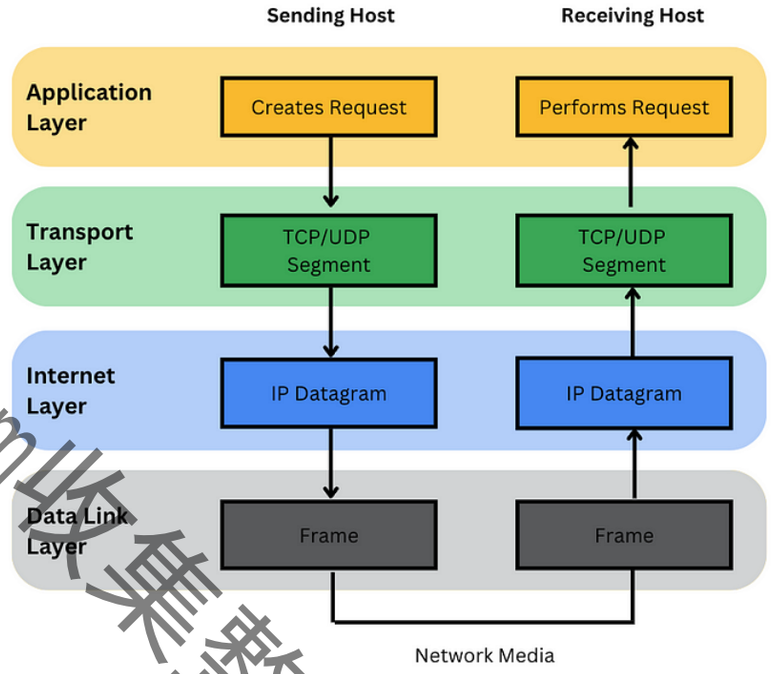
C++ 编程中的网络管理涉及通过编写代码来创建、维护和管理网络连接，以实现数据在计算机和其他设备之间的传输。
参考书籍：《UNIX网络编程》



The 7 layers of the OSI model in *Computer Network*, link



TCP/IP 4 Layer Model, link



Process of Data Sending&Receiving, link

5. 网络管理：连接信息高速公路

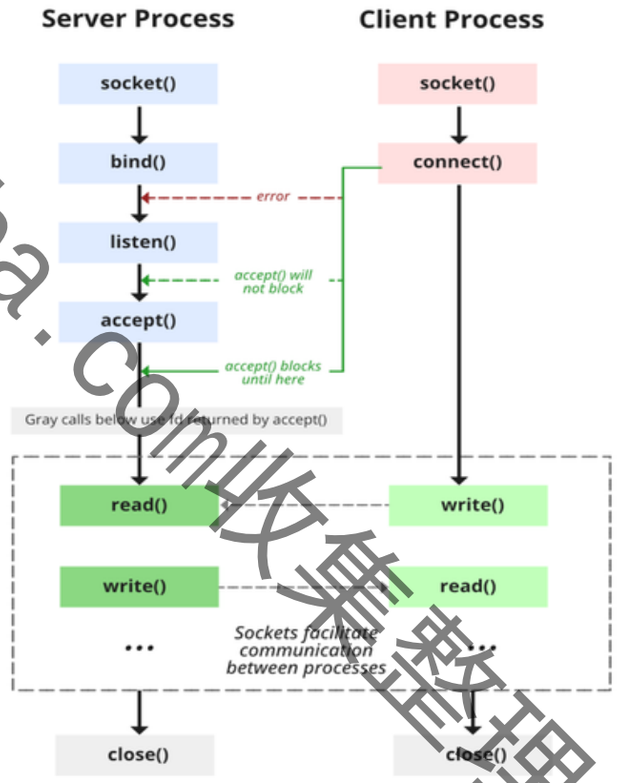
5.2 套接字编程

套接字是网络通信的基本工具。C++ 通过套接字 API 来实现网络通信。主要包括以下几种类型的套接字：

- **流套接字（Stream Socket）**：提供面向连接的、可靠的数据传输，如 TCP。
- **数据报套接字（Datagram Socket）**：提供无连接的、不可靠的数据传输，如 UDP。

常用函数：

- socket()：创建套接字。
- bind()：绑定套接字到一个本地地址和端口。
- listen()：监听连接请求（服务器端）。
- accept()：接受连接请求（服务器端）。
- connect()：连接到服务器（客户端）。
- send()和recv()：发送和接收数据。
- close()：关闭套接字。



State Diagram for server and client model for socket, link

5. 网络管理：连接信息高速公路

5.3 网络库

网络库

C++中有许多常用和著名的网络库，它们提供了丰富的功能和便利的接口，使得网络编程更加高效和易于管理。例如：

1. **Boost.Asio**
   - **简介**：一个广泛使用的跨平台网络库，提供同步和异步I/O操作，支持TCP、UDP、HTTP等协议。
   - **特点**：高效的异步I/O模型，强大的错误处理机制，支持定时器、串口等多种I/O操作
   - **官网**：Boost.Asio
2. **Poco C++ Libraries**
   - **简介**：一个开源的C++类库集合，涵盖网络、并发、文件系统等多种功能。
   - **特点**：支持HTTP、FTP、SMTP等多种协议，提供线程池、任务调度等并发工具
   - **官网**：Poco C++ Libraries
3. **libcurl**
   - **简介**：一个用于客户端URL传输的库，支持HTTP、HTTPS、FTP、SMTP等多种协议。
   - **特点**：强大的数据传输功能，支持SSL/TLS加密，跨平台支持
   - **官网**：libcurl

5. 网络管理：连接信息高速公路

5.3 网络库

网络库

1. **cpp-httplib**
   - **简介：**一个简单的C++ HTTP/HTTPS服务器和客户端库。
   - **特点：**易于使用，轻量级设计，支持SSL/TLS
   - **官网：**cpp-httplib
2. **WebSocket++**
   - **简介：**一个基于C++的WebSocket协议库，用于构建WebSocket服务器和客户端。
   - **特点：**支持WebSocket协议的完整实现，提供同步和异步接口，易于与Boost.Asio集成
   - **官网：**WebSocket++

这些库各具特色，可以根据具体的项目需求选择合适的库来实现高效的网络编程。

5. 网络管理：连接信息高速公路

5.3 网络库

示例

Boost.Asio代码示例

```
#include <boost/asio.hpp>
#include <iostream>
int main() {
    boost::asio::io_context io_context;

    // 解析主机名和服务名
    boost::asio::ip::tcp::resolver resolver(io_context);
    auto endpoints = resolver.resolve("www.example.com", "80");

    // 创建并连接套接字
    boost::asio::ip::tcp::socket socket(io_context);
    boost::asio::connect(socket, endpoints);

    // 发送 HTTP 请求
    std::string request = "GET / HTTP/1.1\r\nHost:
www.example.com\r\n\r\n";
    boost::asio::write(socket, boost::asio::buffer(request));

    // 接收响应
    boost::asio::streambuf response;
    boost::asio::read_until(socket, response, "\r\n");
```

```
    // 输出响应
    std::istream response_stream(&response);
    std::string http_version;
    unsigned int status_code;
    std::string status_message;
    response_stream >> http_version >> status_code;
    std::getline(response_stream, status_message);

    std::cout << "HTTP Version: " << http_version << "\n";
    std::cout << "Status Code: " << status_code << "\n";
    std::cout << "Status Message: " << status_message << "\n";

    return 0;
}
```

5. 网络管理：连接信息高速公路

5.4 应用领域与难点

应用领域

基于C++的网络编程，通常应用于需要**高性能**和**可伸缩性**的系统和应用程序，特别是对网络通信效率和资源利用率有**较高要求**的场景。

例如：

- **服务器端应用程序。**Web服务器、游戏服务器、实时通信服务器等，需要处理大量并发连接和高吞吐量；用于处理和管理大规模数据、分布式计算和存储的数据中心引用等。
- **通信和数据传输。**视频会议系统、流媒体服务等实时音视频传输的任务；云存储服务、大文件传输的业务等。
- **嵌入式系统。**路由器、交换机、防火墙等网络设备，或者物联网IoT设备，需要高效的数据包处理和路由。
- **游戏开发。**需要支持多玩家游戏的实时通信和数据交换的网络游戏服务器。
- **金融和交易系统。**需要低延迟和高并发处理能力的高频交易系统，以及需要实时数据收集、处理和分析金融数据的数据平台。

如果对网络通信的性能、执行效率的要求不高，可以考虑不用C++实现，而是用其他语言更高效地开发实现。

5. 网络管理：连接信息高速公路

5.4 应用领域与难点

难点

- **并发和多线程管理**。当处理大量并发连接和请求时，需要有效地管理多线程或者异步操作，确保线程安全性和性能。
- **内存管理和性能优化。**在内存管理和性能优化方面，需要注意有效地管理和释放动态分配的内存，避免内存泄漏，同时优化内存使用和性能，减少频繁的内存分配和释放对性能的影响。
- **错误和异常处理。**在网络应用中，有效地处理和恢复各种网络异常、超时、连接断开等情况，设计健壮的错误处理机制是必要的。
- **数据安全和隐私。**保护用户数据的安全性和隐私，包括数据加密、身份验证和防范攻击等，是设计和实现网络通信协议时需要重视的方面。

参考书籍

如果希望向互联网方向发展，可以阅读：
《计算机网络》、《UNIX网络编程》、《Boost.Asio C++ 网络编程》、《Linux C++ 网络编程实战》等书籍。

6

GPU管理：
释放计算的潜能

6. GPU管理：释放计算的潜能

6.1 背景

CPU与GPU的区别

CPU（Central Processing Unit，中央处理单元）与**GPU（Graphics Processing Unit，图形处理单元）**的基本区别：

- **CPU**：CPU是一种通用的处理器，主要用于执行程序的**顺序**和**分支逻辑**，对于复杂的**控制流**和**数据依赖关系**有很强的处理能力。每个核心的时钟频率较高，适合**串行任务**和需要大量分支预测的工作负载。
- **GPU**：GPU最初是为了图形渲染而设计的，它包含**成百上千**个小型处理单元（称为流处理器或CUDA核心），能同时处理大量数据并行计算，特别适合于**大规模数据并行**任务。

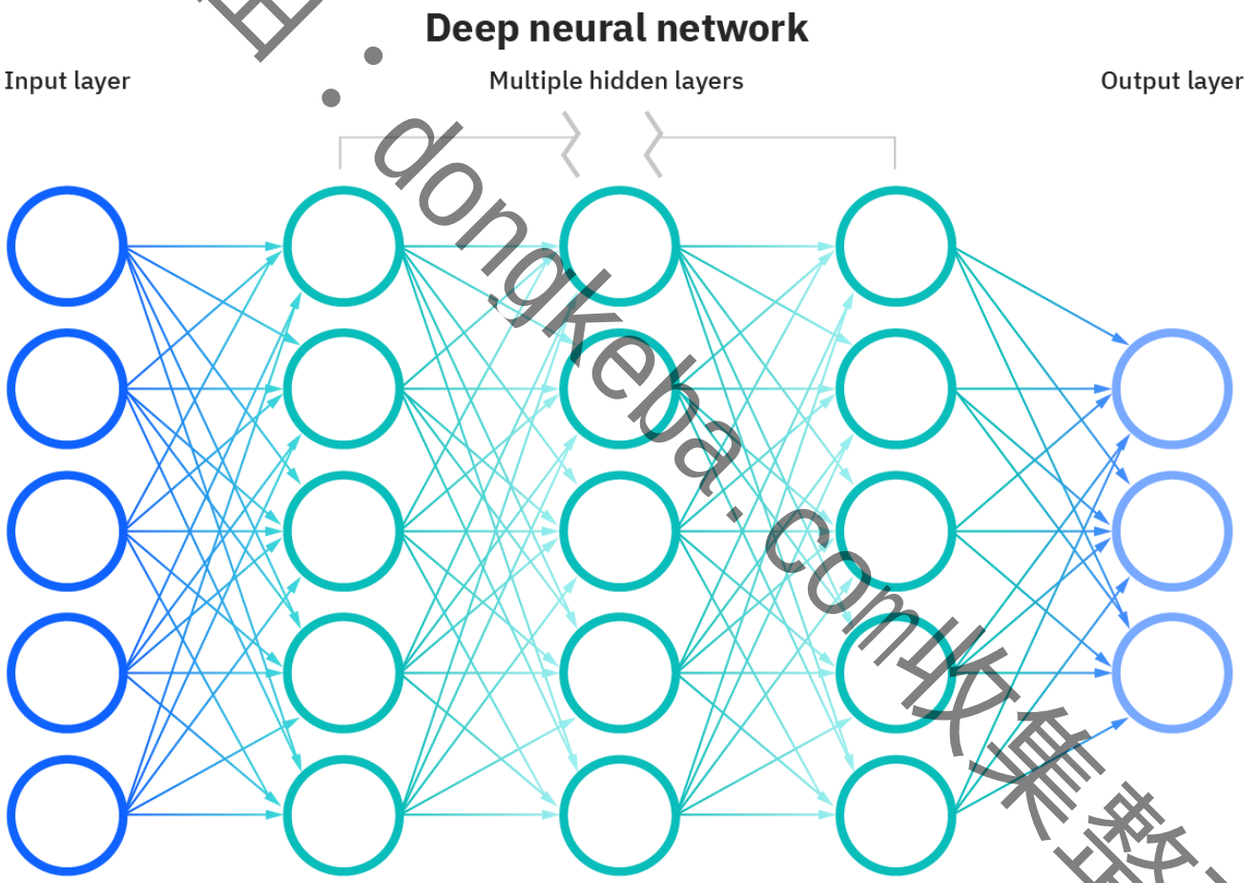
在DNN火爆以前，GPU主要应用在游戏渲染、3D建模、电影视觉特效、科学计算（物理模拟、分子动力学）等领域。

3D Game Rendering, link

6. GPU管理：释放计算的潜能

6.1 背景

CUDA的发展



Deep Neural Network, link

6. GPU管理：释放计算的潜能

6.1 背景

CUDA的发展

随着GPU硬件的发展，**NVIDIA**在2007年推出了并行计算平台和编程模型——CUDA（Compute Unified Device Architecture），GPU开始被用于更广泛的通用计算任务，而不仅仅是图形处理。

- **在CUDA诞生以前**，主要通过**OpenGL**、**DirectX**等图形API进行GPU编程，需要将通用计算任务映射到图形渲染管线上。因此，编程复杂，性能优化困难，需要处理数据编码、纹理映射等问题。
- **在CUDA诞生以后**，它引入了统一的并行计算模型，提供了类似于多线程编程的模式，简化了并行任务的管理和调度。而且CUDA允许程序员直接访问GPU的硬件资源（如核心、寄存器、共享内存），提供了更高级别的控制和优化能力。

NVIDIA最早看到，GPU不仅仅应用于图形学领域计算，在通用任务领域计算也有着巨大的潜力，于是很早就大力开发和推广了CUDA，才有了今天的地位。

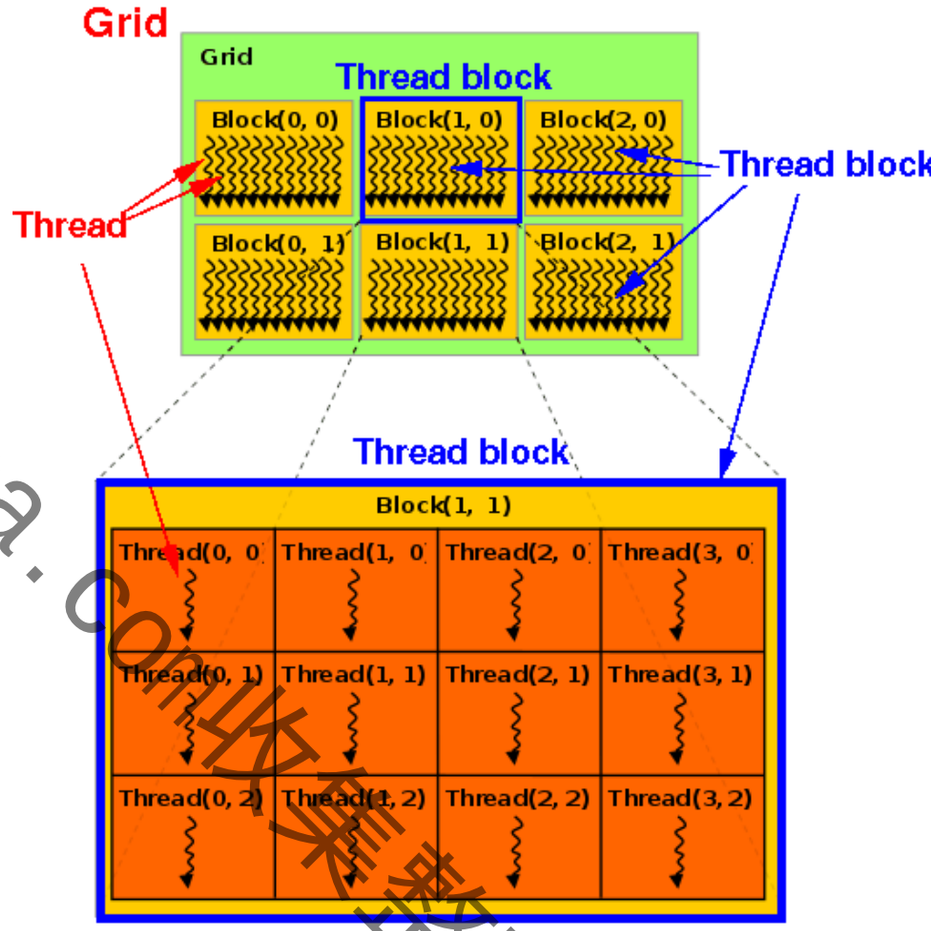
6. GPU管理：释放计算的潜能

6.2 CUDA编程介绍

关键概念

CUDA编程模型基于以下几个关键概念：

- **Kernel函数**：在GPU上并行执行的函数，使用__global__修饰符标记。
- **线程（Thread）**：最小的并行执行单位，在GPU上成百上千个线程同时执行。
- **线程块（Block）**：线程的组合，一起在GPU上的多个处理器（SM，Streaming Multiprocessor）上执行，共享存储器。
- **网格（Grid）**：线程块的集合，整个网格在GPU上执行。
- **GPU内存（显存或VRAM）**：位于GPU卡上，专门用于GPU处理任务。

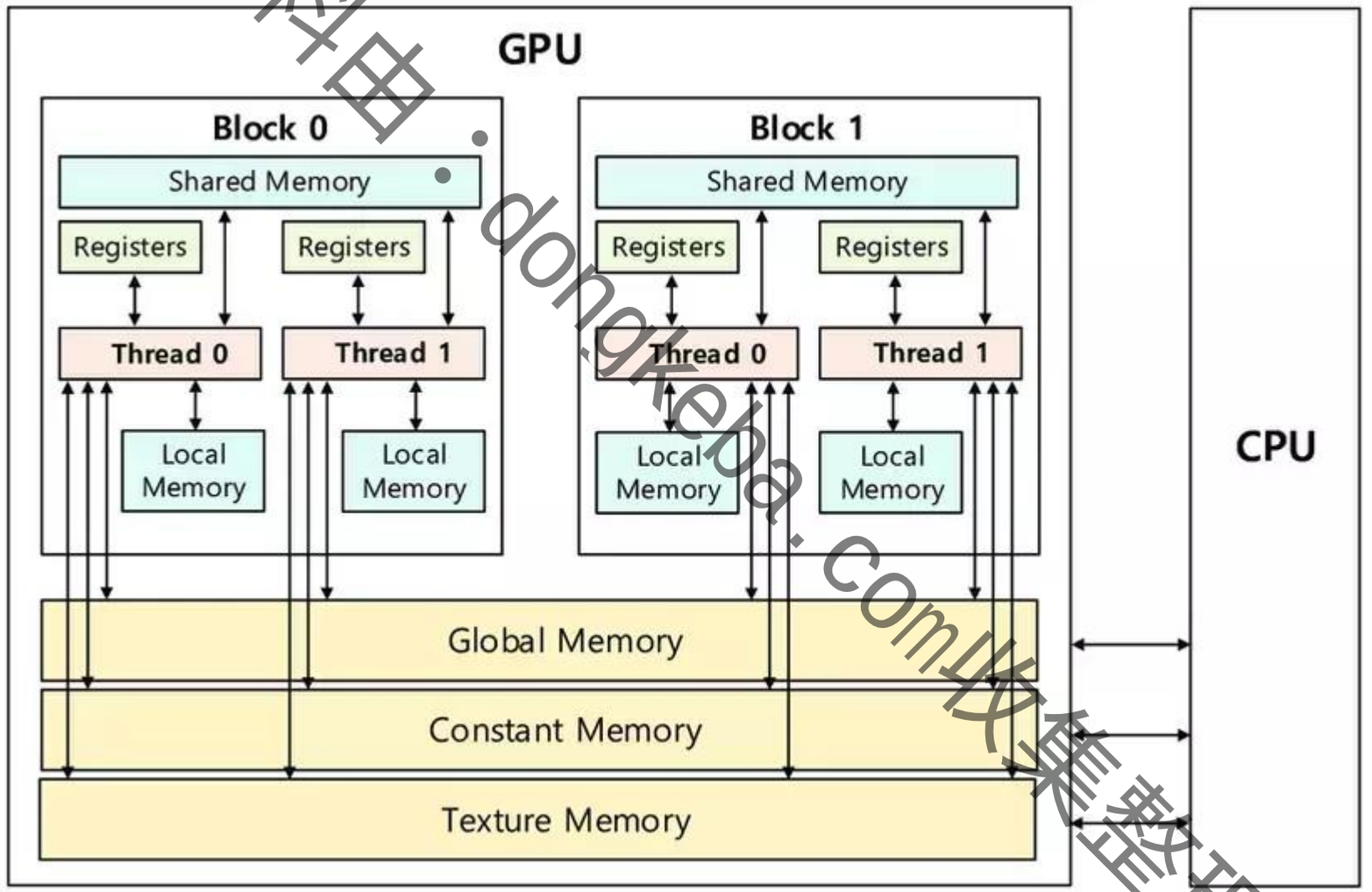


Thread Grid, Thread Block and Thread in GPU, link

6. GPU管理：释放计算的潜能

6.2 CUDA编程介绍

关键概念



Memory in GPU, link

6. GPU管理：释放计算的潜能

6.2 CUDA编程介绍

CUDA编程步骤

使用CUDA编程时，一般会涉及以下基本步骤：

- **分配内存**：使用cudaMalloc在GPU上分配内存。
- **数据传输**：将数据从主机（CPU）传输到设备（GPU），使用cudaMemcpy。
- **执行核函数**：在GPU上执行定义的核函数（kernel function），使用<<<...>>>语法启动核函数执行。
- **将结果传回主机**：将计算结果从GPU传输回主机。
- **释放内存**：使用cudaFree释放在GPU上分配的内存。

函数解释

```
cudaError_t cudaMalloc(void **devPtr, size_t size);
```

d_a是一个未初始化的指针；

&d_a是指针d_a的地址；

当我们调用cudaMalloc((void **)&d_a, size)时，我们将d_a的地址传递给cudaMalloc；

cudaMalloc在GPU上分配一块内存，并将这块内存的地址存储在d_a中。

6. GPU管理：释放计算的潜能

6.2 CUDA编程介绍

示例

```
#include <cuda_runtime.h>
#include <iostream>
#define N 512

__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x;
    c[index] = a[index] + b[index];
}

int main() {
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = N * sizeof(int);

    // 分配内存
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = (int *)malloc(size);
    b = (int *)malloc(size);
    c = (int *)malloc(size);

    // 初始化数据
    for (int i = 0; i < N; i++) {
        a[i] = i; b[i] = i;
    }
```

```
    // 将数据传输到GPU
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // 执行核函数
    add<<<1, N>>>(d_a, d_b, d_c);

    // 将结果传回主机
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // 显示结果
    for (int i = 0; i < N; i++) {
        std::cout << a[i] << " + " << b[i] << " = " << c[i]
<< std::endl;
    }

    // 释放内存
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    free(a);
    free(b);
    free(c);

    return 0;
}
```

6. GPU管理：释放计算的潜能

6.3 更多优化和抽象

更多问题

- 怎么减少主机和GPU之间内存数据复制，提高内存带宽利用率？——多层内存模型（全局、共享、本地等）、共享内存
- 怎样解决并行环境中的数据竞争和一致性问题？——同步和原子操作
- 怎么更好地了解和管理GPU的硬件资源？——设备属性、线程束
- 怎么对任务事件更好地协调？——流、动态并行、事件和异步操作
- 怎么把大量GPU组织成集群，更大规模加速计算？——跨设备计算
- ……

世界对高性能计算提出越来越多、越来越快的要求，CUDA也随着提供越来越多的解决方案和优化方式，以解决用户的需求。

cuDNN

cuDNN（CUDA Deep Neural Network library）是NVIDIA开发的一套用于加速深度学习的GPU加速库。它为深度学习框架（如TensorFlow、PyTorch、Caffe等）提供高度优化的实现，显著提高深度神经网络（DNN）训练和推理的性能。

cuDNN支持多种类型的神经网络层（如卷积层、池化层、归一化层等）和多种数据格式，提供了简单易用的API，可以轻松集成到现有的深度学习框架中。

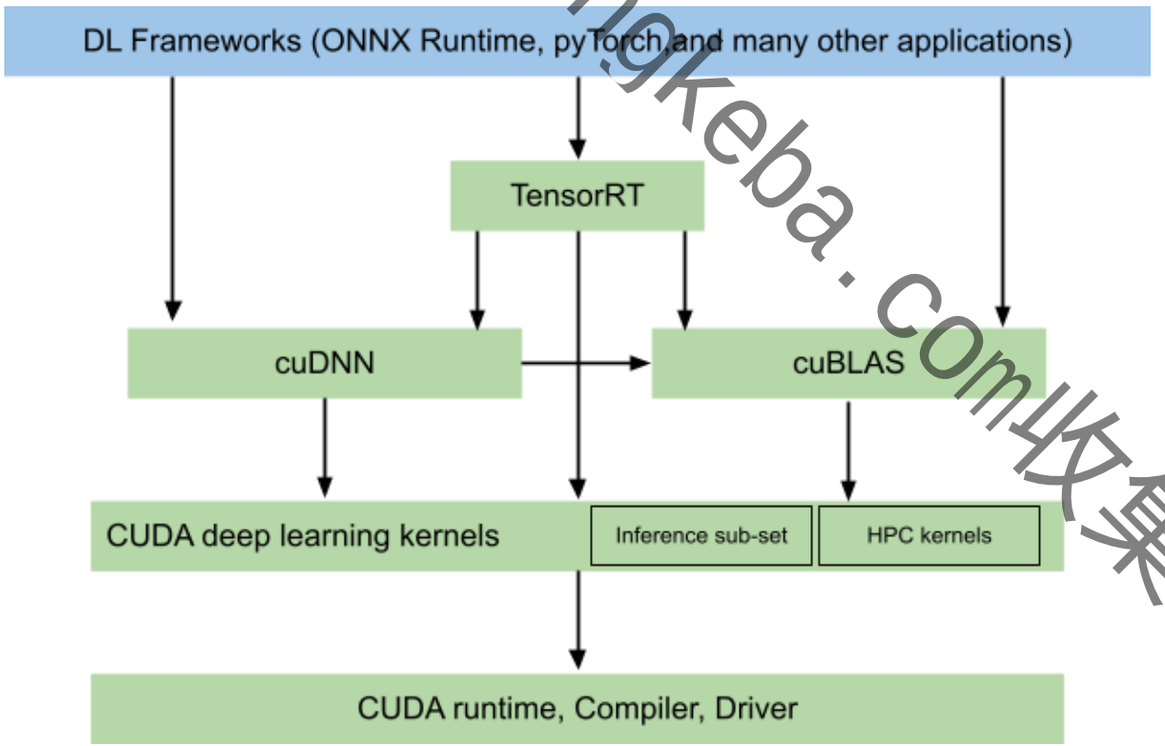
6. GPU管理：释放计算的潜能

6.3 更多优化和抽象

TensorRT

TensorRT是NVIDIA推出的用于高性能深度学习推断（inference）的推理加速库。它主要针对基于深度学习的应用程序，旨在优化和部署训练过的神经网络模型，使其能够在GPU上实现快速的推断。

高层次的推理优化器和运行时引擎，处理整个模型级别的优化；提供更高级的C++和Python API，面向应用开发者。



NVIDIA 推理堆栈, link

6. GPU管理：释放计算的潜能

6.3 更多优化和抽象

参考资料

- 《CUDA By Example: An Introduction to General-Purpose GPU Programming/GPU高性能编程CUDA实战》, by Sason Sanders&Edward Kandrot.
- 《CUDA Programming: A Developer's Guide to Parallel Computing with GPUs/CUDA并行程序设计：GPU编程指南》, by Shane Cook.
- CUDA Toolkit: https://developer.nvidia.com/cuda-toolkit
- NVIDIA cuDNN: https://developer.nvidia.com/cudnn
- NVIDIA TensorRT: https://developer.nvidia.com/tensorrt

7 小结与作业

7. 小结与作业

小结

- **资源概览**：介绍计算机资源的概念，包括各种硬件及对应抽象出来的软件资源，为后续具体资源的深入讨论奠定基础。
- **文件管理**：探讨文件系统和文件流操作，涵盖基本的文件读写到高级的数据格式（如XML、JSON和Protocol Buffer）操作。
- **内存管理**：深入内存的内部布局，讨论常见的内存使用问题，并介绍modern C++先进的内存管理技术，如RAII、内存池以及移动语义和右值引用。
- **线程管理**：讲解C++中的多线程编程，从基本的线程创建和释放，到复杂的线程同步机制，包括互斥、同步、原子操作，以及高级主题如无锁编程和线程池。
- **网络管理**：介绍网络编程的核心概念，包括套接字编程基础，使用现代网络库，并讨论网络编程在各应用领域中的应用和挑战。
- **GPU管理**：探讨GPU编程的背景和重要性，介绍CUDA编程模型，并简单介绍更高级的GPU优化技术和抽象层。

学到这里，我们已经不再是C++小白，而是逐渐向中高级C++开发者靠近。
我们比较全面地了解计算机系统资源及相应的C++编程，对性能问题分析和优化有了一定了解，可以开始面对更复杂的挑战。

7. 小结与作业

感悟

但是很显然，光靠这门课程是不够的。还需要深入学习，系统扎实地掌握计算机知识，才能做好C++资源管理。

资源对象	理论知识	底层/系统层编程实践 高性能优化实践	现代C++实践	对应职位
文件资源管理	计算机组成 计算机体系架构 操作系统	《UNIX环境编程》 《LINUX多线程编程》	<fstream> TinyXML2	Linux系统工程师、 C++研发工程师
内存资源管理			<memory> STL container	
多线程资源管理			<thread><condition_variable> thread pool	
网络资源管理	计算机网络 分布式系统	《UNIX网络编程》	Boost.Asio Websocket++	网络研发工程师
GPU资源管理	GPU体系结构 图形学/深度学习	CUDA/OpenGL	cuDNN TensorRT	高性能计算工程师、 训练/推理优化工程师

为什么大家都说C++难？

- 不仅仅因为C++包含大量的特性、支持多种编程范式；
- 还因为想要成为C++专家，需要掌握背后完整的计算机知识体系，并在至少一个领域有深入的实践。

7. 小结与作业

作业

选择2个和计算机的资源管理有关的项目，进行git下载，按说明去编译，然后运行example程序，并将运行的结果截图。

要求

- 可以从下面的参考项目中选择，也可以选择自己喜欢的其他项目；
- 必须添加或修改至少一行日志输出内容，添加带有个人信息的字符串，来说明自己的确在本地完成了编译和运行；
- 最好进一步添加更多的日志作为调试输出，以更完整、全面地了解整个项目，了解其中关键的C++代码步骤实现了什么。

参考

- TinyXML2
- ThreadPool
- Boost.asio
- Poco.Net
- NVIDIA CUDA Samples
- OpenCUDA

说明

- 知名的项目，往往会提供完善的、多平台的支持，但建议最好在Ubuntu上进行操作；
- 不要求自己实现，只要编译运行即可。最好自行调试、阅读代码、理解核心代码；
- 几乎每个Project都会有使用示例，代码一般为 ./example, ./test, example.cpp, test.cpp, main.cpp 的形式；
- 编译生成的二进制程序一般在 ./build 或 ./bin目录下。

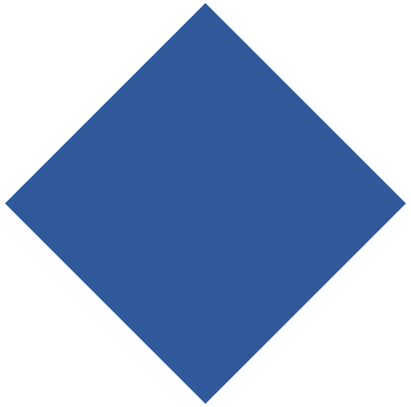
公众号

自动驾驶与计算机视觉
日常干货分享

知识星球

加入自动驾驶之心知识星球，
获取更多硬核干货

自动驾驶与AI全栈技术学习+领域大咖交流+职位内推！



End