高性能服务器必备-线程池

线程池作为提高服务器性能的关键技术之一，其在现代服务器架构中发挥着不可或缺的作用。

基本概念

进程：

- 定义：进程是操作系统分配资源和调度的基本单位。它是一个程序的实例，包含了执行程序的代码和活动路径。
- 特点：每个进程都有自己独立的地址空间，进程间的资源（如内存、文件句柄等）是隔离的。进程间通信（IPC）需要特定的机制，如管道、消息队列、共享内存等。
- 资源消耗：进程的创建、销毁以及上下文切换通常比线程更消耗资源，因为它们涉及更多的系统资源

线程：

- 定义：线程是进程内的一个执行单元，是CPU调度和分派的基本单位。它比进程更轻量级，可以在进程内并发执行。
- 特点：同一进程内的线程共享该进程的资源，如内存、文件句柄等。线程间的通信和数据交换相对更容易，因为它们共享相同的地址空间。
- 资源消耗：线程的创建和销毁、以及上下文切换的资源消耗相对较小。

进程与线程的区别：

1. 资源分配与独立性：进程是资源分配的单位，每个进程拥有独立的地址空间；线程是CPU调度的单位，是进程的一部分，多个线程共享同一进程的资源。
2. 通信方式：进程间通信需要特定的机制，相对复杂；线程间由于共享内存，通信更为简便。
3. 开销大小：创建、销毁进程的开销大于线程，进程间的切换开销也大于线程间的切换。

为什么需要线程池

事件驱动模型下主线程的工作流程

1. 检测新连接：监听到新的连接事件（例如通过 accept）。
2. 注册可读事件：将可读事件注册到事件循环中。
3. 从就绪列表取出事件：当事件准备好（如网络数据已经读取到内核缓冲区），将事件从就绪列表中取出。
4. 处理具体任务：主线程同步地读取数据、解析请求、处理业务逻辑、生成响应。
5. 发送响应：将处理完的响应发送回客户端。

为什么需要线程池

希望服务器主线程不再处理具体工作，只负责分发工作

为什么需要线程池

引入线程池后，服务器的主线程主要负责**事件监听**和**工作分发**，而把**耗时的任务处理**交给线程池中的工作线程。这样做的具体流程如下：

1. **检测新连接**：主线程监听新的连接请求，检测到新连接时建立连接。
2. **注册可读事件**：主线程将该连接的可读事件注册到事件循环中，然后立即返回，继续监听其他事件或新连接。
3. **从就绪列表取出事件**：当连接的数据准备好被读取时，可读事件被触发，加入就绪列表。
4. **分发任务到线程池**：主线程从就绪列表中取出事件后，将实际的任务（如解析请求、处理业务逻辑等）交给线程池中的一个工作线程来处理。主线程继续返回到事件循环中，不阻塞等待任务的处理完成。
5. **工作线程处理具体任务**：线程池中的工作线程接手后，读取内核缓冲区中的数据，解析请求，处理业务逻辑，生成响应等。
6. **发送响应**：处理完请求后，工作线程可能会通过事件循环，将响应数据异步发送回客户端。

线程池基本概念

- 线程池是一组预先初始化并且可重用的线程集合，以处理并发任务，从而避免频繁创建和销毁线程的开销。线程池的主要目标是提高**系统资源利用率**和**并发性能**。

eadPoo

ThreadPool

T1,T2,T3,T4,T5,

线程池的优势

高效的任务管理

线程池通过维持一组活跃的线程来避免频繁地创建和销毁线程，提供了一种高效管理任务的方式。

响应性能提升

服务器利用线程池能迅速响应新的任务请求，因为它无需等待新线程的创建即可开始任务的执行。

资源的有效利用

线程池允许服务器根据当前的工作负载调整线程的数量，从而更有效地利用系统资源。

线程池的核心组件

线程管理器（Thread Manager）：

- 线程池的核心组件，负责线程池的生命周期管理，包括创建初始工作线程、调整线程数量（动态线程池）、销毁不再需要的线程等。
- 它还负责维护任务队列以及任务调度逻辑。
- 线程管理器还可以根据需要调整线程池的大小，即增加或减少工作线程的数量，以适应不同规模的工作负载，从而提高线程池的性能和效率。

工作线程（Worker Threads）：

- 工作线程是线程池中的实际执行单元，每个工作线程都处于循环等待状态，从任务队列中获取并执行任务。
- 当工作线程空闲时，它们会进入阻塞状态等待新的任务到来；当有新任务加入到任务队列时，工作线程会被唤醒并执行任务。

任务队列

- **定义**：一个用于存储待执行任务的队列。

- **作用**：当有新的任务到来时，线程池不会直接为其创建线程，而是将任务添加到任务队列中，等待线程池中的线程来获取和执行。

- **调度机制**：任务队列可以是FIFO（先进先出）队列、**优先级队列**等，调度机制决定了线程池如何选择要执行的任务。

任务（Task / Job）

- **定义**：需要被线程池处理的工作单元，可以是任何可执行的代码块，如函数、对象方法等。

- **作用**：任务是线程池的基本处理对象。当新的任务提交给线程池时，任务会被添加到任务队列中，等待被工作线程执行。

同步机制（Synchronization Mechanisms）：

- 包括互斥锁（mutex）、条件变量（condition variable）、信号量（semaphore）等，用于保护线程池资源的访问，协调任务分配与完成通知。
- 如：在线程池停止、任务入队出队、线程空闲/忙碌状态切换等场景中使用这些机制来保证线程安全。

互斥锁（Mutex）

1

保护共享资源

互斥锁主要用于同一时间内只允许一个线程访问共享资源，确保数据的一致性和完整性。

2

预防竞态条件

Mutex的正确使用可以避免多线程操作中的竞态条件，防止数据损坏和程序错误。

3

控制机制

通过 lock() 和 unlock() 方法，线程可以获取或释放互斥锁，实现对临界区的安全控制。

条件变量（Condition Variable）

线程同步

条件变量是线程间同步的重要工具，它能让线程在某些条件尚未满足时挂起，防止无效的资源消耗。

通知机制

当条件被满足时，其他线程可以通过 notify_one() 或 notify_all() 方法唤醒一个或多个等待的线程。

与互斥锁结合

条件变量通常与互斥锁一起使用，以确保对共享数据访问顺序的逻辑正确性。

条件变量（Condition Variable）和互斥锁（Mutex）区别

条件变量	互斥锁
用于线程间的等待和通知	用于保护共享资源的访问
允许线程等待特定条件的变化	确保一次只有一个线程可以操作资源
通常需要与互斥锁一起使用	可独立使用来实现线程同步

线程池的工作流程

1. **初始化**：线程池初始化时，创建一定数量的工作线程。
2. **任务提交**：当新的任务提交到线程池时，线程池会将任务放入任务队列中。
3. **任务分配**：空闲的工作线程会从任务队列中取出任务，并执行任务代码。
4. **任务执行**：工作线程完成任务后，不会销毁，而是继续从任务队列中取出新的任务执行。如果任务队列为空，则进入空闲状态，直到有新的任务到来。
5. **线程数量调整**：线程池管理器根据当前任务量和线程使用情况，动态调整线程数量（增加、减少线程），以适应并发需求。

本节课用到的C++知识

std::function

std::function 是 C++11 引入的一个**通用可调用对象包装器**，它能够存储、复制、调用各种类型的可调用对象，比如**函数指针**、**Lambda 表达式**、**仿函数**（重载了 operator() 的对象），以及 std::bind 生成的绑定对象等。

std::function 的主要作用是**将不同类型的可调用对象统一封装**起来，形成一个一致的接口，可以像普通函数一样调用它，而不需要关心底层对象的类型。

示例代码

```
#include <functional>
#include <iostream>

// 定义一个接受两个 int 参数并返回 int 的 std::function
std::function<int(int, int)> add;
// 普通函数
int sum(int a, int b) {
    return a + b;
}

add = sum; // 绑定函数指针
std::cout << add(2, 3) << std::endl; // 输出 5
```

示例代码

```
// Lambda 表达式
add = [](int x, int y) { return x + y; };
std::cout << add(4, 5) << std::endl; // 输出 9

// 仿函数（重载 operator() 的类）
struct Adder {
    int operator()(int a, int b) const {
        return a + b;
    }
};
add = Adder();
std::cout << add(6, 7) << std::endl; // 输出 13
```

优势

- **灵活性**：可以将不同类型的可调用对象（函数、Lambda、仿函数等）用相同的接口封装，方便函数参数传递和回调机制的实现。

- **类型安全**：std::function 会检查函数签名匹配，确保调用时参数和返回类型一致。

std::function<void()>

1. std::function<void()> 是C++标准库中的一种类型，它表示可以存储和调用任何无参数且没有返回值的可调用对象（Callable Object）的通用类型。这里的语法知识要点如下：

- std::function 是一种泛型类模板，它提供了一种类型安全的方式来存储、传递和调用不同类型的可调用对象。
- 它能够接受任意符合其签名要求的函数指针、lambda 表达式、bind 表达式结果以及重载了 operator() 的类实例（即函数对象）。

void() :

- 这部分是 std::function 类模板的参数化部分，它定义了可调用对象的类型。
- 在这个例子中，“void()”表示的是一个无参数并且返回 void 的函数签名。
  - void 表示该函数不返回任何值。
  - 参数列表为空括号 “()`”，意味着此可调用对象在调用时不需要任何参数。

使用场景

- 当你需要将某个函数或可调用实体作为一个类成员变量存储，或者作为函数参数传递时，使用 std::function<> 可以使代码更加灵活，因为你可以在运行时决定具体执行哪个操作。
- 例如，在事件处理、回调函数注册、多线程编程中的任务队列等场景下经常用到。

std::mutex（互斥锁）

- 定义与初始化：

```
std::mutex queue_mutex; // 创建一个互斥锁对象
```

锁定与解锁：

- 使用lock()方法来获取锁，如果锁已经被其他线程持有，则当前线程会阻塞直到获取到锁。
  queue_mutex.lock();
- 使用unlock()方法释放锁，使其他等待该锁的线程有机会获取并执行临界区代码。
  queue_mutex.unlock();

示例代码

```
std::mutex queue_mutex; // 定义一个互斥锁
int shared_counter = 0; // 共享资源

// 线程任务函数，增加共享计数器
void incrementCounter(int id) {
    for (int i = 0; i < 5; ++i) {
        // 获取锁
        queue_mutex.lock();

        // 临界区代码：访问共享资源
        ++shared_counter;
        std::cout << "Thread " << id << " incremented counter to " << shared_counter << std::endl;

        // 释放锁
        queue_mutex.unlock();
    }
}
```

自动管理锁的生命周期

- 为了防止忘记解锁导致死锁或资源泄露，可以使用std::lock_guard或std::unique_lock。当它们超出作用域时，会自动调用unlock()释放锁。

示例代码

```
std::mutex queue_mutex; // 定义一个互斥锁
int shared_counter = 0; // 共享资源

// 线程任务函数，增加共享计数器
void incrementCounter(int id) {
    for (int i = 0; i < 5; ++i) {
        // 使用 lock_guard 获取锁
        std::lock_guard<std::mutex> guard(queue_mutex);
        // 临界区代码：访问共享资源
        ++shared_counter;
        std::cout << "Thread " << id << " incremented counter to " << shared_counter << std::endl;

        // 离开作用域时，lock_guard 会自动释放锁
    }
}
```

示例代码

```
std::mutex queue_mutex; // 定义一个互斥锁
int shared_counter = 0; // 共享资源

// 线程任务函数，增加共享计数器
void incrementCounter(int id) {
    for (int i = 0; i < 5; ++i) {
        // 使用 unique_lock 获取锁
        std::unique_lock<std::mutex> lock(queue_mutex);

        // 临界区代码：访问共享资源
        ++shared_counter;
        std::cout << "Thread " << id << " incremented counter to " << shared_counter << std::endl;

        // 离开作用域时，unique_lock 会自动释放锁
        // 或者可以选择提前解锁： lock.unlock();
    }
}
```

对比

特性	std::lock_guard	std::unique_lock	queue_mutex.lock()/unlock()
性能	高效，轻量级	性能略低于 std::lock_guard，但差距很小	性能最高，因为是手动控制加解锁
自动解锁	是，在作用域结束时解锁	是，在作用域结束时解锁	否，需要手动调用 unlock()
灵活度	低，不能提前解锁、延迟加锁、移动	高，支持提前解锁、延迟加锁、移动	高，完全手动控制加锁和解锁

std::condition_variable（条件变量）

- 定义与初始化：

```
std::condition_variable condition;
```

条件变量通常与互斥锁一起使用

用于线程间同步，当满足特定条件时唤醒线程。通过调用wait()函数，线程会释放互斥锁并进入休眠状态，直到被其他线程通过notify_one()或notify_all()唤醒，并重新获得锁后继续执行。

```
 std::mutex cv_mutex;
 std::condition_variable condition;
 void waitingThread() {
    std::unique_lock
std::mutex lock(cv_mutex);
    while (!data_ready) {
      // 检查某个共享变量是否满足条件
      condition.wait(lock); // 条件不满足时等待通知
    } // 当条件满足时，这里可以安全地访问和修改数据
    processData();
}
```

通知等待线程：

- notify_one()：唤醒至少一个正在等待此条件变量的线程（如果有多个线程在等待，会选择其中一个唤醒）。

```
void notifierThread() {
    // 更新数据或改变条件
    data_ready = true;

    std::lock_guard<std::mutex> guard(cv_mutex); // 获取锁
    condition.notify_one(); // 唤醒一个等待的线程
}
```

- `notify_all()`：唤醒所有正在等待此条件变量的线程。

```
condition.notify_all(); // 唤醒所有等待的线程
```

总结

std::mutex用于实现互斥访问,

std::condition_variable则提供了基于条件的等待和唤醒机制，使得多线程编程中的复杂同步问题得以解决

右值引用

值类别（Value Category） 在C++中，每个表达式都有一个值类别，分为两种：

- 左值（lvalue）：具有持久存储位置的表达式，可以出现在赋值操作符的左边或右边，例如变量名、数组元素、解引用的指针等。
- 右值（rvalue）：临时对象或者将要销毁的对象，不能作为左值使用，通常不会有一个固定的内存地址。包括字面量、函数返回值、运算结果等。

左值

变量名

```
int x = 42; // x 是一个左值
int y = x;  // 这里， x 在赋值操作符的左边， 作为左值被使用
```

数组元素

```
int arr[5] = {1, 2, 3, 4, 5};
int secondElement = arr[1]; // arr[1] 是一个左值， 可以赋值给其他变量
```

解引用的指针

```
int value = 10;
int* ptr = &value;
int copiedValue = *ptr; // *ptr 是一个左值， 它指向存储在内存中的实际整数值
```

右值

字面量

```
int sum = 10 + 20; // 这里的 "10 + 20" 是一个右值，它是临时计算的结果，没有独立的名字或持久存储位置
```

函数返回值

```
int createInt() {
    return 42;
}
void useInt(int&& rvalue) {
    std::cout << "Rvalue: " << rvalue << std::endl;
}
// 使用：
10useInt(createInt());
// 创建并传递给useInt的42是一个右值，因为它是函数createInt的返回值
```

引用类型

左值引用（lvalue reference）：表示为 T&，只能绑定到左值上。左值引用允许你给一个已存在的对象起一个新的名字，并且通过这个新名字操作原有对象。

```
int x = 10;
int& ref_to_x = x; // ref_to_x 是 x 的左值引用
```

右值引用（rvalue reference）：表示为 T&&，可以绑定到右值和即将被销毁的左值（通过std::move()转换）。主要目的是为了支持移动语义和完美转发。

移动构造

右值引用最核心的应用是实现资源的有效转移，而非复制。通过定义移动构造函数和移动赋值运算符，你可以“窃取”右值的资源，而不需要拷贝这些资源，从而提高效率。

```
int main() {
    std::vector<int> v1 = {1, 2, 3, 4, 5};

    // v2 通过右值引用 v1 转移资源，而不是复制
    std::vector<int> v2 = std::move(v1);
    // v1 被"移动"，变为空，v2 接管了 v1 的数据
    std::cout << v1.size() << std::endl;
    std::cout << v2.size() << std::endl;

    return 0;
}
```

与传统指针的区别

- 当你直接传递一个指向动态分配内存或堆上对象的原始指针时，不会发生所有权转移。调用方仍然需要负责释放该内存。
- 使用传统指针时，内存没有被“移动”，而是被**共享**，容易造成资源管理问题，比如**双重释放**。

```
int main() {
    // 使用传统指针分配数组
    int* arr1 = new int[5]{1, 2, 3, 4, 5};
    // 直接将指针赋值给 arr2（共享同一块内存）
    int* arr2 = arr1;
    // 释放资源
    delete[] arr2;
    // 如果再次释放 arr1，就会导致未定义行为（双重释放）
    // delete[] arr1; // 错误：指向同一块内存
    return 0;
}
```

特性	**传统指针**	**移动构造**
资源共享	多个指针指向同一块内存，容易混淆	内存和资源被**唯**一地转移给另一个对象
内存管理	手动管理，可能出现双重释放或内存泄漏	自动管理，避免双重释放、内存泄漏，资源更安全
效率	直接指针赋值，没有资源拷贝	**高效转移资源**，无拷贝，只“窃取”指向的数据
所有权	不改变，多个指针拥有同一内存	**转移所有权**，只有一个对象拥有资源
代码安全性	需要小心管理指针的生命周期	构造和析构自动管理资源，降低出错风险

线程池任务传递的问题

•线程池通常需要接受用户提交的任务（如std::function、lambda等），然后将其交给线程执行。
•任务可能包含复杂的资源（如动态内存、文件句柄等），频繁的拷贝会导致性能下降。

理想目标：以最小的性能开销、安全地将任务转交给线程池管理的工作线程。

线程池任务传递的问题

•线程池通常需要接受用户提交的任务（如std::function、lambda等），然后将其交给线程执行。
•任务可能包含复杂的资源（如动态内存、文件句柄等），频繁的拷贝会导致性能下降。
•如果任务是通过引用传递，可能会引发悬垂引用的问题。

理想目标：以最小的性能开销、安全地将任务转交给线程池管理的工作线程。

为什么不直接传递引用呢

```
void submit(Task& task) {
    tasks_.push(task); // 保存任务的引用
}
int main() {
    ThreadPool pool;
    {
        Task t(1000); // 局部任务
        pool.submit(t); // 提交任务的引用
    } // t超出作用域，被销毁
    pool.processTasks(); // 悬垂引用，未定义行为
}
```

代码示例

```
// 处理函数，接受左值引用
void process(int& x) {
    std::cout << "Lvalue processed: " << x << std::endl;
}

// 处理函数，接受右值引用
void process(int&& x) {
    std::cout << "Rvalue processed: " << x << std::endl;
}
```

代码示例

```
// 包装函数（没有完美转发）
int main() {
    int a = 10;
    // 传入左值
    process(a);  // 输出：Lvalue processed: 10
     // 传入右值
    process(20); // 输出：Lvalue processed: 20 (错误地调用了左值版本)

    return 0;
}
```

完美转发

以传入参数的原始形式（左值或右值）将其传递给其他函数。

完美转发的主要目的是实现对可调用对象（如函数、lambda表达式、成员函数指针等）**及其参数的无损传递**，使得接收这些参数的目标函数可以按照传入参数的原始形式处理它们。

通过使用`std::forward`和模板参数推导，模板函数可以保持传入参数原有的**左值引用或右值引用性质**，从而决定是在目标函数内部进行拷贝操作、移动操作还是直接使用。

原因

因为在 C++ 中，**所有命名变量（包括函数参数）都是左值**，即使它们是右值引用类型

为什么 arg 是左值？

• 在 wrapper 函数中，当 arg 被传递到另一个函数（比如 process）时，arg 是**一个命名变量**，而在 C++ 中，**所有命名变量都是左值**。

• 即使 arg 的类型是 T&&（右值引用），但在使用 arg 时，它仍然是一个左值，因为它有一个名字，可以被多次访问和使用。

怎么实现完美转发

std::forward 是 C++11 中引入的一个**类型转换工具**，用于在模板函数中实现**完美转发**。它能够根据参数的类型特性（左值/右值、const/非 const）进行正确地转发，确保参数在转发时保持其原始类型特性

• 在模板函数中，它通常用于将函数参数传递给另一个函数，同时确保参数是左值时被传递为左值，是右值时被传递为右值。

怎么实现完美转发

```
// 包装函数，使用完美转发
template <typename T>
void wrapper(T&& arg) {
    // 使用 std::forward 完美转发参数
    process(std::forward<T>(arg));
}

int main() {
    int a = 10;
    // 传入左值
    wrapper(a);  // 输出：Lvalue processed: 10
    // 传入右值
    wrapper(20); // 输出：Rvalue processed: 20 (正确地调用了右值版本)
    return 0;
}
```

std::future

传统的函数调用是**同步阻塞的**：一个函数在执行完之前，主线程是无法继续做其他事情的

std::future 是 C++11 引入的用于处理**异步操作结果**的对象，提供了一种可以获取异步任务结果的机制。它主要解决**任务执行和结果获取分离**的问题，使得我们可以**非阻塞地执行任务**，并在需要时获取其结果。

std::future

用法

1. **异步任务的创建与结果获取**

使用 std::async 可以方便地创建一个异步任务，并返回一个 std::future 来获取其结果。

2. **获取结果**

使用 get() 方法获取 future 中的结果（如果任务还未完成，会阻塞直到任务完成）

std::future

std::future 的用法总结

- **创建** future：可以通过 std::async 或 std::packaged_task 来创建一个 future，用于异步执行任务。
- get() **方法**：用于获取异步任务的结果，阻塞直到任务完成。
- wait() **方法**：只等待异步任务完成，不获取结果。
- valid() **方法**：检查 future 是否有可用的异步结果。

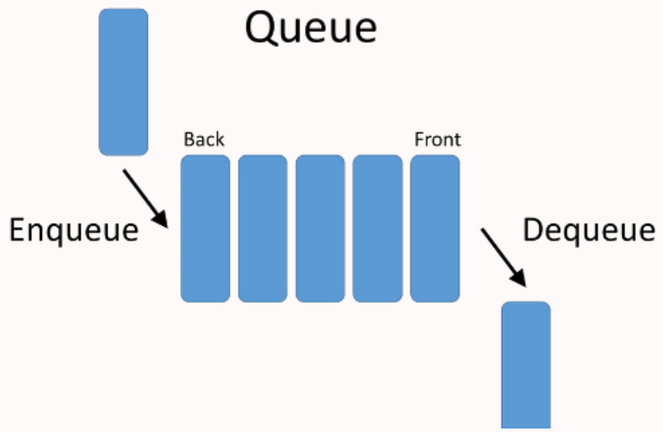
示例代码

```
// 一个模拟耗时的函数
int longComputation(int x) {
std::this_thread::sleep_for(std::chrono::seconds(2)); // 模拟耗时操作
    return x * 2;
}
int main() {
    // 使用 std::async 创建一个异步任务
    std::future<int> result = std::async(std::launch::async, longComputation, 10);

    // 在等待结果的同时做其他事情，此处省略一万行

    // 获取异步任务的结果（阻塞直到任务完成）
    int value = result.get();
    return 0;
}
```

代码详解

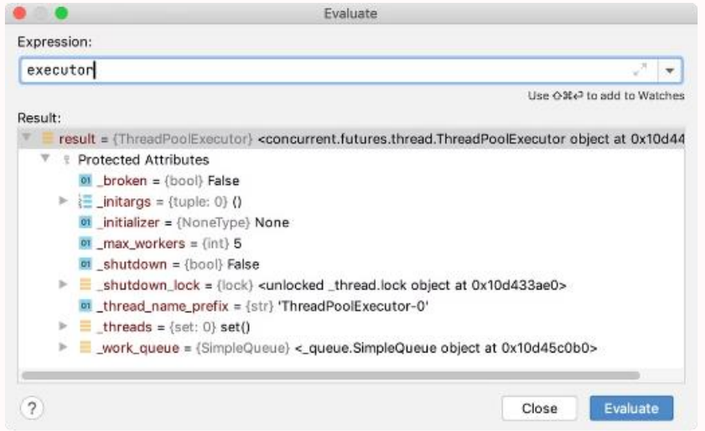
构造函数解析

深入理解线程池的构造函数



任务添加机制

分析如何通过 enqueue 方法将新任务添加到线程池中



线程池实际应用

探索服务器如何使用 ThreadPool 类处理并发请求

复杂代码专讲

```
auto task = std::make_shared<std::packaged_task<return_type()>>(
std::bind(std::forward(f), std::forward(args)...)
);
/*
创建一个std::packaged_task实例，封装了参数化模板中的可调用对象f及其参数args。
通过std::bind将可调用对象与其参数绑定在一起，形成一个新的可调用实体，这个实体在调用时会执行原函数。
将此std::packaged_task实例封装到一个std::shared_ptr中，便于在多线程环境下安全地共享、调度和执行任务
*/。
```

std::bind

需求背景

1.简化函数调用：让一个函数只需要传入部分参数，其他参数预先绑定好。

2.延迟函数调用：在某个时间点创建函数，但在另一个时间点执行它。

3.改变参数顺序：根据实际需要，调整函数参数的顺序或部分预设。

std::bind

使用例子，我已经有了下面的函数，但我的需求是每次输出hello xxx !，只有xxx是变化的。

```
void printMessage(const std::string& prefix, const std::string& message, const std::string& suffix) {
    std::cout << prefix << message << suffix << std::endl;
}
```

std::bind

传统方案

```
void sayHello(const std::string& message) {
    printMessage("Hello, ", message, "!");
}
```

问题：

1. **可重用性低**：如果有不同的 prefix 和 suffix 组合，就得写多个函数。

2. **不灵活**：不能在运行时动态改变参数。

std::bind

使用示例

```
int main() {
    // 使用 std::bind 将 prefix 和 suffix 预设成 "Hello, " 和 "!"
    auto sayHello = std::bind(printMessage, "Hello, ", std::placeholders::_1, "!");

    // 现在 sayHello 只需要一个参数
    sayHello("World"); // 输出：Hello, World!
    sayHello("C++");   // 输出：Hello, C++!
    return 0;
}
```

std::bind

std::bind 解决了以下问题：

1.参数绑定：将原始函数的某些参数固定，生成一个新的可调用对象 sayHello，只需要一个参数 message。
2.灵活调用：可以随时调用 sayHello，传入不同的 message，但 prefix 和 suffix 是固定的。
3.复用函数逻辑：你不需要写多个版本的函数来满足不同的参数组合，std::bind 可以帮你动态调整。
4.更高阶的用法可以改变原始函数的参数顺序，异步调用函数，详情见课程文档

std::packaged_task

需求背景

在现代 C++ 编程中，有一种常见的需求：我们希望将一个函数、Lambda 表达式或者其他可调用对象**封装成任务**，可以在某个时间点异步执行，并在稍后某个时间点获取到它的结果。

std::packaged_task

在 C++11 中，可以通过 std::packaged_task 来解决这个问题。std::packaged_task 可以将一个可调用对象**封装为异步任务**，并与 std::future 绑定，以便在任务完成后获取结果。

std::packaged_task

```
// 模拟一个耗时的计算
int longComputation(int x) {
    std::this_thread::sleep_for(std::chrono::seconds(2)); // 模拟耗时操作
    return x * 2;
}
```

std::packaged_task

```
int main() {
// 使用 std::packaged_task 封装 longComputation 函数 std::packaged_task<int(int)> task(longComputation);
// 获取与任务关联的 future
std::future<int> result = task.get_future();

// 在另一个线程中执行 task
std::thread t(std::move(task), 10);

// 主线程可以做其他事情
std::cout << "Doing other work while waiting for the result...\n";

// 获取异步任务的结果
std::cout << "Result: " << result.get() << std::endl; // 阻塞，直到获取结果

// 确保线程完成
t.join();

return 0;
}
```

std::packaged_task

封装任务：

std::packaged_task<int(int)> task(longComputation);

packaged_task 将 longComputation 封装起来，使其成为一个可以异步执行的任务。

• 模板参数 <int(int)> 指定了被封装的函数的签名：int 返回值， 参数是 int。

std::packaged_task

`std::packaged_task<return_type()>` 表示：

- 任务的**返回类型**是 `return_type`。
- 任务**不接受任何参数**，或者说参数为空。

std::packaged_task

```
auto task = std::make_shared<std::packaged_task<return_type()>>(
std::bind(std::forward(f), std::forward(args)...)
);
```

- **原始任务 f**：

原始任务 f 可以是一个函数、Lambda 表达式、或其他可调用对象，**且它可以接受参数**。

- **绑定参数**：

`std::bind` 将 `f` 和 `args...` 绑定在一起，生成一个新的可调用对象。这个新的对象不需要传递参数，因为 `args...` 已经被提前绑定。

- `std::packaged_task` **包装的最终任务**：

`std::packaged_task` 包装的就是 `std::bind` 生成的这个可调用对象，**不接受任何参数**，但返回类型是 `return_type`。

std::make_shared

`std::make_shared<std::packaged_task<return_type()>>`

`std::make_shared`是一个工厂函数，用于创建一个`std::shared_ptr`实例，指向动态分配的对象。在这里，它被用来创建一个指向`std::packaged_task<return_type()>`类型的实例的智能指针。

使用`std::make_shared`的好处在于它可以一次性分配所有需要的内存（包括管理块和对象本身），从而减少内存分配次数并提高效率，同时确保资源正确释放。

服务器中使用线程池

异步任务处理

服务器监听到新的IO事件或请求后，通过线程池分配任务，加快主线程返回并继续监控其他事件的速度。

效率提升

线程池允许服务器以非阻塞方式处理请求，极大提高了处理效率和并发处理的能力。

资源优化

合理地利用线程池中的线程来处理任务，可降低服务器的资源占用，并保持高效的运行。

线程池的任务执行和管理

1 统一任务执行

线程池中的线程统一从任务队列中获取任务，并按要求执行，保障服务的连贯性和稳定性。

2 线程生命周期

管理线程的生命周期，确保线程在完成任务后能够有效地被回收或用于下一个任务。

3 任务结果处理

通过 future 对象获取任务执行结果，为服务器响应提供即时反馈。

服务器主循环与线程池结合

1

监听事件

服务器主循环使用 epoll 等机制监听网络事件，实现高效的事件驱动模型。

2

快速任务分配

主线程将检测到的IO事件转换为任务，并快速分配至线程池中的线程进行处理。

3

异步执行指令

线程池内的线程异步执行指令，主线程得以不断监听新事件，确保服务器的持续响应能力。

代码实操