第12课 多线程服务器提高性能

铺垫概念：

1. 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，包括内存分配、加载程序等。
2. 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。

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。

为什么要引入线程池

工作流程的变化：引入线程池

在没有线程池的情况下，服务器主线程要完成整个事件处理流程：

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。

由于主线程既要**负责监听和管理事件**，又要**处理实际的任务**，这可能会导致瓶颈，尤其是在高并发场景下。

引入线程池后的优化

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（如读取数据、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。

线程池的优点

- **降低主线程压力**：主线程只负责事件循环和任务分发，而不需要处理每个具体任务，确保主线程高效地处理大量并发连接。

- **提高并发性能**：引入线程池后，多个工作线程可以并行处理具体的任务请求，大大提高了并发能力，减少了主线程被耗时任务阻塞的可能。

- **控制资源使用**：线程池中的线程数量是可控的，避免了为每个连接创建新线程导致的资源开销过大。线程池也可以重用线程，降低线程的创建和销毁成本。

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，用于执行多个任务。这种方法比为每个任务单独创建和销毁线程更加高效。

线程池的核心组件

1. **任务队列**（Task Queue / Work Queue）

- **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。

2. **工作线程**（Worker Threads）

- **定义**：线程池中预先创建的、实际执行任务的线程集合。

- **作用**：这些线程从任务队列中取出任务并执行它们。在执行完一个任务后，它们会继续从任务队列中获取下一个任务，而不是销毁自身。

- **线程数控制**：线程池通常允许设置线程的**最小数量**和**最大数量**，以及**闲置线程的存活时间**。当任务数多时，线程池会创建更多线程来处理任务；当任务数少时，空闲的线程会被销毁或挂起。

3. **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。

4. **线程池管理器**（Thread Pool Manager）

- **定义**：负责管理整个线程池的生命周期、任务分配、线程创建与销毁等工作。

- **作用**：线程池管理器监控线程池的状态（如工作线程数量、任务队列长度、线程空闲时间等），并做出相应的调整。它的功能包括：

  - **线程管理**：根据任务数量和配置，决定是否增加或减少线程。

  - **任务分配**：将任务从任务队列分配给空闲的工作线程。

  - **错误处理**：处理线程运行中的异常、失败的任务等。

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，通常是最小线程数。

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。

其他重要概念

- **创建线程**
  - 在 `ThreadPool` 类的构造函数中创建固定数量的线程。这些线程在创建时进入等待状态，等待执行任务。

- **互斥锁（Mutex）**：
  - 互斥锁是一种同步原语，用于保护共享资源或临界区，确保在任何时刻只有一个线程可以访问这些资源。
  - 在多线程环境中，如果不使用互斥锁来保护共享数据，可能会导致竞态条件和数据损坏。
  - 使用 `std::mutex` 类可以创建互斥锁，通过 `lock()` 和 `unlock()` 方法来控制互斥锁的加锁和解锁。
- **条件变量（Condition Variable）**：
  - 条件变量用于线程间的通信和协同工作。它允许一个线程等待某个条件的发生，而其他线程可以在满足条件时通知等待线程。
  - `std::condition_variable` 是C++标准库中的条件变量类，用于实现线程的等待和唤醒。
  - 常见的用法是结合互斥锁使用，等待线程在等待某个条件时调用 `wait()` 方法挂起，而其他线程在满足条件时调用 `notify_one()` 或 `notify_all()` 方法来通知等待线程继续执行。
- **线程执行任务**
  - 线程从任务队列中取出任务并执行。执行完任务后，线程不会结束，而是继续等待下一个任务。

- **线程池销毁**
  - 在 `ThreadPool` 类的析构函数中，通知所有线程停止等待并完成当前任务，然后退出。

线程池在服务器中的应用

1. 集成线程池：
   - 在服务器的 `main` 函数中，通常会创建一个线程池（`ThreadPool`）的实例。这个线程池包含了一组预先创建的线程，这些线程在服务器启动时就已准备好处理任务。
2. 处理新连接：
   - 当服务器使用 `epoll_wait` 或类似的机制检测到新的连接请求时，服务器会接受这些连接，并将每个连接作为一个新的任务添加到线程池的任务队列中。
   - 这些任务通常包括读取客户端的请求数据、处理请求并生成响应，然后将响应发送回客户端。
3. 任务处理：
   - 服务器中的每个任务都是一个需要在某个线程中执行的工作单元。这些任务可以是读取请求数据、解析请求、处理业务逻辑、发送响应等。
   - 当服务器检测到有新的IO事件（例如，套接字可读）时，它会创建一个任务并将其添加到线程池的任务队列中等待执行。
4. 异步执行：
   - 通过线程池，服务器可以异步处理任务。这意味着主线程可以继续监听新的事件，而不必等待某个任务完成。
   - 异步执行提高了服务器的并发性能，允许服务器同时处理多个请求，而不会被阻塞在单个请求上。
5. 线程安全：
   - 在服务器中处理共享资源（例如套接字、内存等）时，需要特别注意线程安全性。多线程环境下，多个线程可能同时访问共享资源，因此必须采取适当的同步措施，以避免数据竞态和一致性问题。

线程池是一种常见的多线程编程技术，用于提高服务器的并发性

本节课用到的C++知识

std::function<>

`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：

1. std::function：
   - `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）。

2. 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。
3. 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<void()>` 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。

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

// 普通函数
void regular_function() {
    std::cout << "This is a regular function." << std::endl;
}

// 函数对象（仿函数）
struct Functor {
    void operator()() const {
        std::cout << "This is a functor." << std::endl;
    }
};

int main() {
    // 使用普通函数
    std::function<void()> func1 = regular_function;
    func1();

    // 使用 lambda 表达式
    std::function<void()> func2 = []() {
        std::cout << "This is a lambda expression." << std::endl;
    };
    func2();

    // 使用函数对象
    Functor functor;
```

```
    std::function<void()> func3 = functor;
    func3();

    // 使用成员函数
    struct MyClass {
        void member_function() {
            std::cout << "This is a member function." << std::endl;
        }
    };

    MyClass obj;
    // 需要使用 std::bind 绑定对象
    std::function<void()> func4 = std::bind(&MyClass::member_function, &obj);
    func4();

    return 0;
}
```

锁和信号量

1. 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();
```

- **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。

```cpp
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();
    }
}

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
    }
}

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();
    }
}
```

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

2. std::condition_variable（条件变量）

- **定义与初始化**：

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

- **等待特定条件**：

- 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
```

总结来说，`std::mutex` 用于实现互斥访问，而 `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

理解右值引用需要对C++中的值类别、左值引用和右值引用的概念有深入的了解。

1. 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。

2. 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。

3. 移动语义与移动构造函数/移动赋值运算符

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。

```
class ResourceIntensiveClass {
public:
    // 移动构造函数，接受一个右值引用参数
    ResourceIntensiveClass(ResourceIntensiveClass&& other)
        : data(std::move(other.data)) {
        other.data = nullptr; // 资源从other转移到当前对象，并清空other
    }

    // 移动赋值运算符
    ResourceIntensiveClass& operator=(ResourceIntensiveClass&& other) {
        std::swap(data, other.data); // 交换数据指针，相当于资源的移动
```

```
        return *this;
    }

private:
    BigData* data; // 假设data指向一块大内存
};
```

1. 移动构造函数：

```
ResourceIntensiveClass(ResourceIntensiveClass&& other)
    : data(std::move(other.data)) {
    other.data = nullptr;
}
```

1. 当创建一个新的 `ResourceIntensiveClass` 对象时，如果传入的是右值引用（即临时对象或即将被销毁的对象），那么这个构造函数会被调用。通过 `std::move`，我们将 `other` 中的 `data` 指针的所有权转移给新创建的对象。接着将 `other.data` 设置为 `nullptr`，这样原对象不再拥有该资源，避免了资源的复制和泄漏，并且原对象处于可安全析构的状态。

2. 移动赋值运算符：

```
ResourceIntensiveClass& operator=(ResourceIntensiveClass&& other) {
    std::swap(data, other.data);
    return *this;
}
```

1. 移动赋值运算符用于将一个右值引用的 `ResourceIntensiveClass` 对象的资源所有权转移到已存在的对象上。通过 `std::swap` 函数交换两个对象的 `data` 指针，实现了资源的“移动”，即将 `other` 的资源赋予当前对象，同时使 `other` 放弃对资源的所有权。

4. 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。

不使用完美转发的示例：

```cpp
#include <iostream>

// 打印左值引用
void print(int& x) {
    std::cout << "Non-const Lvalue reference: " << x << std::endl;
}

// 打印 `const` 左值引用
void print(const int& x) {
    std::cout << "Const Lvalue reference: " << x << std::endl;
}

// 模板包装函数
template <typename T>
void wrapper(T&& arg) {
    print(arg); // 直接传递参数
}

int main() {
    int a = 10;
    const int b = 20;

    wrapper(a);  // 输出：Non-const Lvalue reference: 10
    wrapper(b);  // 输出：Non-const Lvalue reference: 20 (错误地调用了非 `const` 版本)
}
```

```cpp
#include <iostream>

// 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;
}

// 包装函数（没有完美转发）
template <typename T>
void wrapper(T&& arg) {
    // 直接传递参数到 process 函数
    process(arg);
}
```

```
19
20 int main() {
21     int a = 10;
22
23     // 传入左值
24     wrapper(a);  // 输出: Lvalue processed: 10
25
26     // 传入右值
27     wrapper(20); // 输出: Lvalue processed: 20 (错误地调用了左值版本)
28
29     return 0;
30 }
```

在模板编程中，通过使用右值引用和 `std::forward()` ，可以实现完美转发，即能以传入参数的原始形式（左值或右值）将其传递给其他函数。

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。

```
1 template<typename T>
2 void forward_func(T&& arg) {
3     func_impl(std::forward<T>(arg)); // 完美转发arg到func_impl
4 }
5
6 // ...
7 void func_impl(ResourceIntensiveClass& obj) {...} // 对左值版本的处理
8 void func_impl(ResourceIntensiveClass&& obj) {...} // 对右值版本的处理
```

完美转发改进后

```
1 #include <iostream>
2 #include <utility> // std::forward
3
4 // 处理函数，接受左值引用
5 void process(int& x) {
6     std::cout << "Lvalue processed: " << x << std::endl;
7 }
8
```

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

// 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;
}
```

复杂语句

```cpp
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
*/。
```

从内到外，

std::bind

需求背景

在实际开发中，有时我们希望：

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。

这些需求非常常见。例如，假设你有一个复杂的函数，你希望简化调用时的参数。看看这个例子：

```
1 void printMessage(const std::string& prefix, const std::string& message, const
  std::string& suffix) {
2     std::cout << prefix << message << suffix << std::endl;
3 }
```

这个函数需要传入 `prefix`、`message` 和 `suffix`。如果你有一些常用的 `prefix` 和 `suffix`，你可能不希望每次都传入它们，只想传入中间的 `message`，这样代码会更简洁。

怎么解决需求?

在传统的方式中，你可能需要编写一个额外的函数来解决：

```
1 void sayHello(const std::string& message) {
2     printMessage("Hello, ", message, "!");
3 }
```

但这种方式存在问题：

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。

引入 `std::bind` 解决需求

为了动态地绑定参数，避免重复编写很多函数，C++11 引入了 `std::bind`。它允许我们**提前绑定**一部分参数，并生成一个新的可调用对象（类似函数）。这样，你可以把通用函数转化为不同场景下可用的函数对象。

```cpp
#include <iostream>
#include <functional> // std::bind

// 复杂的原始函数
void printMessage(const std::string& prefix, const std::string& message, const std::string& suffix) {
    std::cout << prefix << message << suffix << std::endl;
}

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` 的作用

`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。

更复杂的需求

改变参数顺序

如果你希望改变原始函数的参数顺序，也可以使用 `std::bind`。

```cpp
// 假设希望交换 prefix 和 message 的顺序
auto reverseArgs = std::bind(printMessage, std::placeholders::_2,
std::placeholders::_1, "!");
reverseArgs("World", "Hey, "); // 输出：Hey, World!
```

异步调用函数

有时，你希望将函数绑定后，在另一个时间点调用，比如用 `std::thread`。

```cpp
#include <thread>

// 绑定参数，将 message 固定为 "Async Task"
auto asyncTask = std::bind(printMessage, "Start: ", "Async Task", " Done!");

std::thread t(asyncTask); // 异步调用
t.join(); // 等待线程结束
```

这样，函数调用就被延迟到了线程中异步执行。

std::async 和 std::future

std::async 是 C++11 标准库中的异步执行函数，用于启动一个异步任务，可能在新线程中执行。

```cpp
#include <iostream>
#include <future>

int compute(int x) {
    return x * x;
}

int main() {
    // 启动异步任务
    std::future<int> result = std::async(std::launch::async, compute, 10);

    // 在需要结果时获取
    std::cout << "Result: " << result.get() << std::endl;

    return 0;
```

```
16 }
```

std::future是 C++11 标准库中的模板类，用于获取异步操作的结果。

常用成员函数

- `get()`：获取异步操作的结果，若结果未准备好，会阻塞当前线程。
- `wait()`：等待异步操作完成，不获取结果。
- `valid()`：检查 `future` 是否包含有效的共享状态。

代码详解

构造函数

```
// 构造函数，初始化线程池
ThreadPool(size_t threads) : stop(false) {
    // 创建指定数量的工作线程
    for(size_t i = 0; i < threads; ++i) {
        workers.emplace_back([this] {
            while(true) {
                std::function<void()> task;
                {
                    // 创建互斥锁以保护任务队列
                    std::unique_lock<std::mutex> lock(this->queue_mutex);
                    // 使用条件变量等待任务或停止信号
                    this->condition.wait(lock, [this]{ return this->stop ||
!this->tasks.empty(); });
                    // 如果线程池停止且任务队列为空，线程退出
                    if(this->stop && this->tasks.empty()) return;
                    // 获取下一个要执行的任务
                    task = std::move(this->tasks.front());
                    this->tasks.pop();
                }
                // 执行任务
                task();
            }
        });
    }
```

```
24 }
```

这里初始化了指定数量的线程，每个线程等待执行队列中的任务。

任务添加

```
1  // 将函数调用包装为任务并添加到任务队列，返回与任务关联的 future
2  template<class F, class... Args>
3  auto enqueue(F&& f, Args&&... args)
4      -> std::future<typename std::result_of<F(Args...)>::type> {
5      using return_type = typename std::result_of<F(Args...)>::type;
6      // 创建共享任务包装器
7      auto task = std::make_shared<std::packaged_task<return_type()>>(
8          std::bind(std::forward<F>(f), std::forward<Args>(args)...)
9      );
10     // 获取与任务关联的 future
11     std::future<return_type> res = task->get_future();
12     {
13         // 使用互斥锁保护任务队列
14         std::unique_lock<std::mutex> lock(queue_mutex);
15         // 如果线程池已停止，则抛出异常
16         if(stop) throw std::runtime_error("enqueue on stopped ThreadPool");
17         // 将任务添加到队列
18         tasks.emplace([task](){ (*task)(); });
19     }
20     // 通知一个等待的线程去执行任务
21     condition.notify_one();
22     return res;
23 }
24
25
```

`enqueue` 方法用于添加新的任务到线程池。当有新的IO事件或者请求处理任务时，就会调用这个方法。

这段代码是线程池的一个关键部分，用于将任务添加到线程池的队列中，并返回一个 `std::future` 对象，该对象与任务的结果相关联。下面是对这段代码的详细解释：

函数模板定义

```
1  template<class F, class... Args>
```

```
auto enqueue(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type> {
```

- `template<class F, class... Args>`：这是一个模板函数，可以接受任何类型的函数 `F` 和参数 `Args`。
- `auto enqueue(F&& f, Args&&... args)`：`enqueue` 函数接受一个函数 `f` 和它的参数 `args`。
- `std::future<typename std::result_of<F(Args...)>::type>`：返回一个 `std::future` 对象，该对象将持有函数 `f` 执行后的结果类型。

创建任务和获取未来结果

```
using return_type = typename std::result_of<F(Args...)>::type;

auto task = std::make_shared<std::packaged_task<return_type()>>(
    std::bind(std::forward<F>(f), std::forward<Args>(args)...)
);
std::future<return_type> res = task->get_future();
```

- `using return_type`：定义了函数 `f` 的返回类型。
- `std::make_shared<std::packaged_task<return_type()>>`：创建一个 `std::packaged_task` 对象，它包装了函数 `f` 和它的参数。这使得函数可以在未来某个时刻执行。
- `std::bind(std::forward<F>(f), std::forward<Args>(args)...)`：将函数 `f` 和它的参数绑定起来，为延迟执行做准备。
- `task->get_future()`：获取一个与任务关联的 `std::future` 对象，该对象稍后可以用来获取函数 `f` 的执行结果。

将任务添加到队列

```
{
    std::unique_lock<std::mutex> lock(queue_mutex);
    if(stop) throw std::runtime_error("enqueue on stopped ThreadPool");
    tasks.emplace([task](){ (*task)(); });
}
```

- `std::unique_lock<std::mutex> lock(queue_mutex)`：使用互斥锁保护任务队列，确保同时只有一个线程可以访问。
- `if(stop) throw std::runtime_error("enqueue on stopped ThreadPool")`：如果线程池已停止，则抛出异常。
- `tasks.emplace([task](){ (*task)(); })`：将任务添加到线程池的任务队列中。任务是以 lambda 表达式的形式添加的，当这个 lambda 被执行时，它将调用 `task`。

通知线程池中的线程

```
1 condition.notify_one();
```

- `condition.notify_one()`：通知线程池中的一个等待线程（如果有的话），告诉它有新的任务可以执行了。

返回 `std::future` 对象

```
1 return res;
```

- 这个函数最终返回一个 `std::future` 对象，调用者可以用它来获取任务执行的结果。

服务器中使用线程池

```
1  ThreadPool pool(4);  // 创建线程池
2  while (true) {
3      nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
4      for (int n = 0; n < nfds; ++n) {
5          if (/* 新连接 */) {
6              // 处理新连接
7          } else {
8              // 添加任务到线程池
9              pool.enqueue([/* 任务 */] { /* 处理请求 */ });
10         }
11     }
```

```
12 }
```

在这里，服务器主循环使用 `epoll` 监听事件，并使用线程池异步处理请求，使得主线程可以快速地返回到监听新事件的状态。

```
// 在线程池中异步处理每个就绪的socket
pool.enqueue([fd = events[n].data.fd]() {
    // 使用 lambda 表达式并捕获当前处理的 socket 文件描述符
    LOG_INFO("Handling request on socket: " + std::to_string(fd));

    // 设置缓冲区用于读取请求数据
    char buffer[1024] = {0};
    read(fd, buffer, 1024); // 从socket读取数据
    std::string request(buffer); // 将读取的数据转换为字符串

    // 解析HTTP请求
    auto [method, uri, body] = parseHttpRequest(request); // 解析请求，获取方法、URI和请求体

    // 处理HTTP请求并获取响应体
    std::string response_body = handleHttpRequest(method, uri, body);

    // 构建HTTP响应
    std::string response = "HTTP/1.1 200 OK\nContent-Type: text/plain\n\n" + response_body;
    send(fd, response.c_str(), response.size(), 0); // 发送响应到客户端

    // 关闭处理完的socket
    close(fd);
    LOG_INFO("Request handled and response sent on socket: " + std::to_string(fd));
});


//上面使用了lambda。如果不用lambda应该是下面这样
void handleSocketRequest(int fd) {
    LOG_INFO("Handling request on socket: " + std::to_string(fd));
    char buffer[1024] = {0};
    read(fd, buffer, 1024);
    std::string request(buffer);
    auto [method, uri, body] = parseHttpRequest(request);
    std::string response_body = handleHttpRequest(method, uri, body);
    std::string response = "HTTP/1.1 200 OK\nContent-Type: text/plain\n\n" + response_body;
    send(fd, response.c_str(), response.size(), 0);
```

```
    close(fd);
    LOG_INFO("Request handled and response sent on socket: " +
std::to_string(fd));
}

// 主函数中...
pool.enqueue(std::bind(&handleSocketRequest, events[n].data.fd));
```

面试常问问题和答案

问题1：进程和线程的区别是什么？

回答：

- **进程**是操作系统资源分配的基本单位，每个进程有自己独立的内存空间、文件描述符等资源。
- **线程**是操作系统调度的基本单位，一个进程可以包含多个线程，线程共享进程的资源。
- **区别**：
  - **资源**：进程之间资源独立，线程共享进程资源。
  - **开销**：进程创建和切换开销大，线程开销小。
  - **通信**：进程间通信复杂，线程间通信方便。
  - **稳定性**：一个进程崩溃不影响其他进程，一个线程崩溃可能导致整个进程崩溃。

问题2：什么是线程池？它有哪些优点？

回答：

- **线程池**是一种线程使用模式，预先创建一定数量的线程，等待任务到来时分配线程执行，完成后线程回到池中。
- **优点**：
  - **降低资源消耗**：重复利用线程，减少创建和销毁线程的开销。

- **提高响应速度**：任务到来时，无需等待新线程的创建，立即执行。
- **增强稳定性**：控制线程数量，避免资源耗尽。
- **便于管理**：统一调度和管理线程，提高资源利用率。

问题3：如何实现一个线程池？

回答：

- **设计思路**：
  - **任务接口**：定义任务的抽象接口。
  - **任务队列**：线程安全的队列，存放待执行的任务。
  - **工作线程**：创建固定数量的线程，从任务队列中获取任务并执行。
  - **同步机制**：使用互斥锁和条件变量，保证线程安全和同步。
- **实现步骤**：
1. 创建任务队列和工作线程。
2. 主线程将任务添加到任务队列。
3. 工作线程等待任务队列的任务，获取任务并执行。
4. 任务完成后，线程回到池中，继续等待任务。

问题4：在多线程环境下，如何保证数据的安全性？

回答：

- **使用互斥锁（std::mutex）**：保护共享数据，防止多个线程同时访问导致数据不一致。
- **使用条件变量（std::condition_variable）**：在线程间同步，等待某个条件满足。
- **使用原子操作（std::atomic）**：对于简单的整数、自增操作，可以使用原子类型，避免使用锁。

- **避免数据竞争**：尽量减少共享数据的使用，使用线程本地存储。

问题5：什么是死锁？如何避免死锁？

回答：

- **死锁**是指两个或多个线程相互等待对方持有的资源，导致所有线程都无法继续执行。
- **避免方法**：
  - **资源有序分配**：按照一定的顺序申请和释放锁，避免循环等待。
  - **避免长时间持有锁**：缩小锁的作用域，减少锁的持有时间。
  - **尝试锁（std::try_lock）**：尝试获取锁，获取失败则避免阻塞。
  - **使用更高级的并发容器**：如线程安全的队列、集合等，减少显式加锁。

问题6：什么是右值引用？有什么作用？

回答：

- **右值引用**是 C++11 引入的特性，使用 `&&` 表示，可以绑定到右值（临时对象）。
- **作用**：
  - **实现移动语义**：通过移动构造函数和移动赋值运算符，避免不必要的拷贝，提高性能。
  - **完美转发**：在模板中保持参数的左值或右值属性，使用 `std::forward` 实现。

问题7：std::function<> 和函数指针有什么区别？

回答：

- **std::function<>** 是一个通用的函数包装器，可以存储任意可调用对象（函数、函数指针、lambda、函数对象等）。

- **函数指针** 只能指向函数或静态成员函数。

- **区别**：

  - **灵活性**：`std::function<>` 更加通用，支持各种可调用对象。

  - **类型安全**：`std::function<>` 可以检查参数和返回值类型。

  - **开销**：`std::function<>` 有一定的额外开销，函数指针更轻量。

问题8：在 C++ 中，如何实现线程间的通信？

回答：

- **使用条件变量和互斥锁**：一个线程等待条件变量，另一个线程通知。

- **使用线程安全的队列**：如 `std::queue` 配合互斥锁，实现生产者-消费者模型。

- **使用 `std::promise` 和 `std::future`**：通过设置和获取异步结果，实现通信。
