# 【C++】深入理解C++回调函数：从原理到实践

原创 lcg空间 lcg技术空间 2025年03月31日 17:33 广东

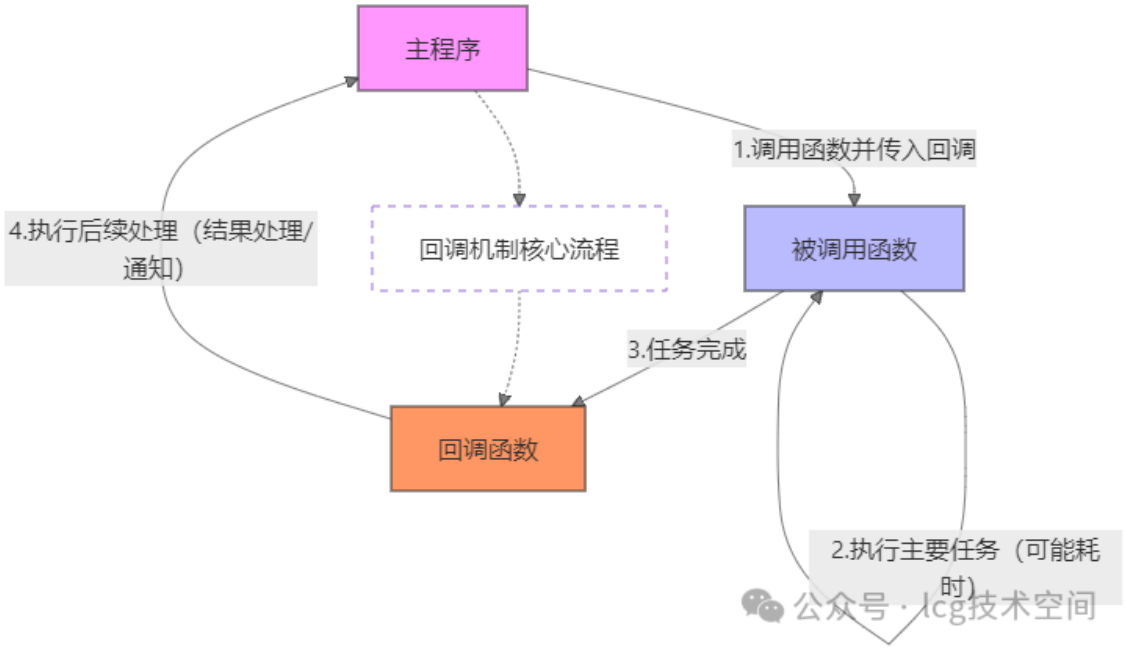
## 为什么需要回调函数?

在传统的同步编程中，我们直接调用函数并等待其返回结果。这种模式简单直观，但当遇到耗时操作时（如I/O操作、网络请求等），线程会被阻塞，导致资源利用率低下。

回调函数是解决以下问题的关键：

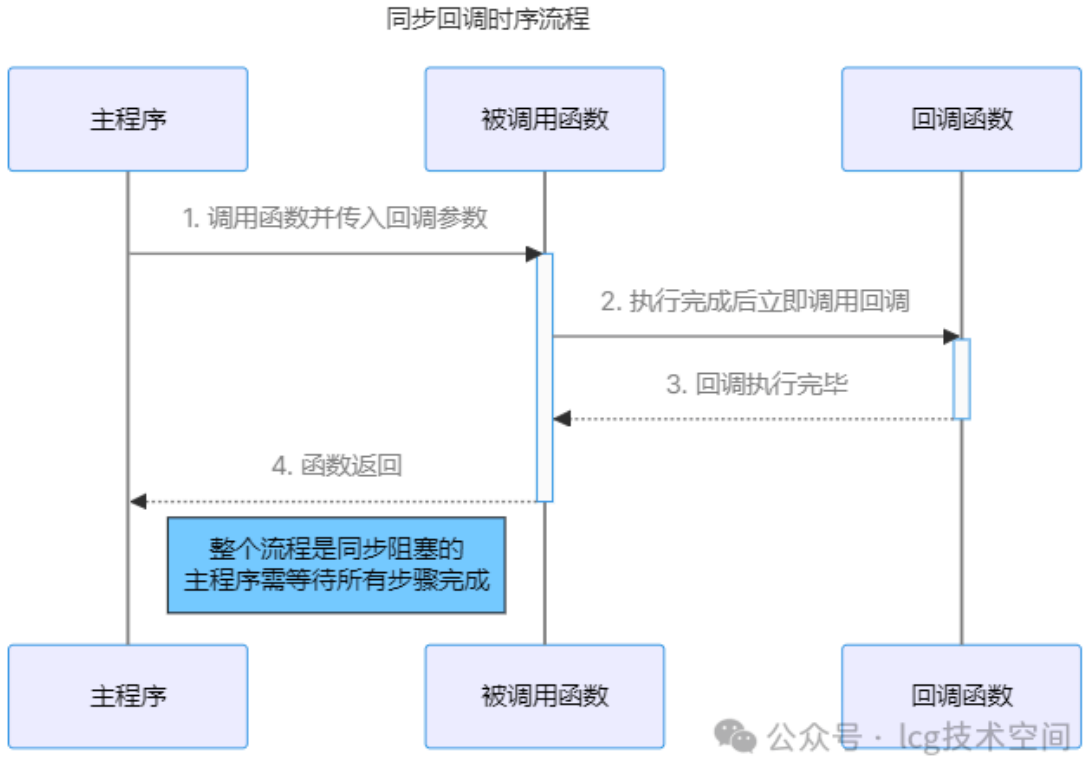
1. **解耦**：将功能实现与调用分离
2. **异步处理**：避免阻塞主线程
3. **扩展性**：允许不同调用方自定义处理逻辑

## 回调函数的工作流程

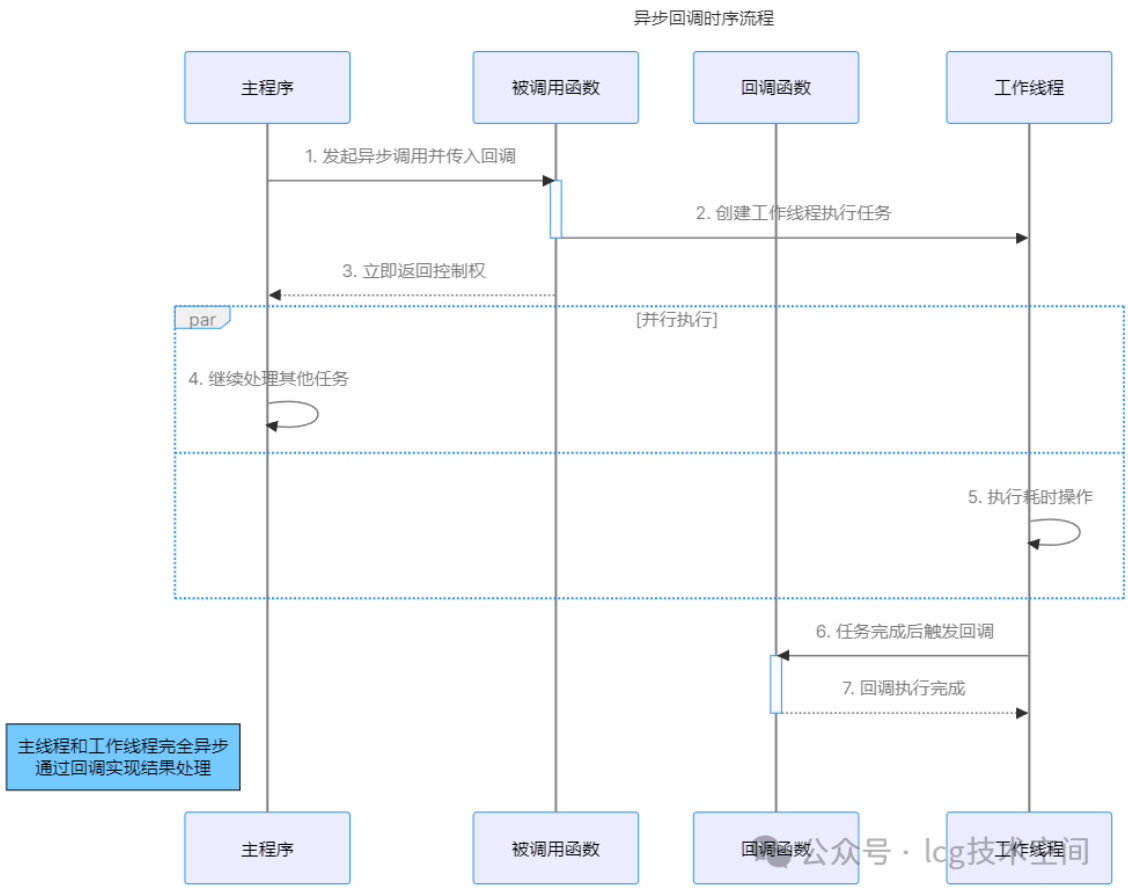



## 回调函数的类型

## 同步回调流程图


同步回调时序流程
主程序
被调用函数
回调函数
1. 调用函数并传入回调参数
2. 执行完成后立即调用回调
3. 回调执行完毕
4. 函数返回
整个流程是同步阻塞的
主程序需等待所有步骤完成
主程序
被调用函数
回调函数
公众号 · lcg技术空间

## 异步回调流程图


异步回调时序流程
主程序
被调用函数
回调函数
工作线程
1. 发起异步调用并传入回调
2. 创建工作线程执行任务
3. 立即返回控制权
par
[并行执行]
4. 继续处理其他任务
5. 执行耗时操作
6. 任务完成后触发回调
7. 回调执行完成
主线程和工作线程完全异步
通过回调实现结果处理
主程序
被调用函数
回调函数
工作线程
公众号 · lcg技术空间

## 示例分析：明日油条App案例

### 同步调用的问题

```
// 同步版本
void make_youtiao(int num) {
    // 耗时制作过程
    for(int i=0; i<num; i++) {
        // 制作每个油条...
    }
}

// 调用方式
make_youtiao(10000);  // 阻塞10分钟
sell();               // 必须等待制作完成
```

### 回调函数解决方案

```
// 回调函数版本
void make_youtiao(int num, std::function<void()> callback) {
    // 耗时制作过程
    for(int i=0; i<num; i++) {
        // 制作每个油条...
    }
    callback();  // 制作完成后调用回调
}

// 调用方式
make_youtiao(10000, [](){
    std::cout << "油条制作完成，开始售卖" << std::endl;
    sell();
});
```

### 异步回调实现

```
#include <thread>
#include <functional>
```

```
void async_make_youtiao(int num, std::function<void()> callback) {
    std::thread([num, callback](){
        // 在新线程中执行耗时操作
        for(int i=0; i<num; i++) {
            // 制作每个油条...
        }
        callback();  // 完成后调用回调
    }).detach();  // 分离线程，使其独立运行
}

// 调用方式
async_make_youtiao(10000, [](){
    std::cout << "油条制作完成，开始售卖" << std::endl;
    sell();
});
std::cout << "主线程继续执行其他任务..." << std::endl;
```

## C++中实现回调的几种方式

### 1. 函数指针

```
typedef void (*Callback)(int);

void process(Callback cb) {
    // 处理...
    cb(42);  // 调用回调
}
```

### 1. std::function

```
void process(std::function<void(int)> cb) {
    // 处理...
    cb(42);  // 调用回调
}
```

### 1. Lambda表达式

```
process([](int result) {
    std::cout << "Got result: " << result << std::endl;
});
```

### 1. 类成员函数作为回调

```
class Processor {
public:
    void handleResult(int result) {
        std::cout << "Result: " << result << std::endl;
    }
};

Processor p;
process(std::bind(&Processor::handleResult, &p, std::placeholders::_1));
```

## 回调函数的最佳实践

### 1. 错误处理

```
void async_op(std::function<void(Error*, Data*)> callback) {
    try {
        Data* data = do_work();
        callback(nullptr, data);  // 成功时error为null
    } catch (const std::exception& e) {
        callback(new Error(e.what()), nullptr);
    }
}
```

### 1. 资源管理

```
void with_file(const std::string& path,
               std::function<void(FILE*)> processor) {
    FILE* f = fopen(path.c_str(), "r");
    if (!f) throw std::runtime_error("File open failed");

    try {
        processor(f);
    } catch (...) {
        fclose(f);
        throw;
    }
    fclose(f);
}
```

### 1. 线程安全回调

```
class EventHandler {
    std::mutex mtx;
    std::function<void()> callback;

public:
    void setCallback(std::function<void()> cb) {
        std::lock_guard<std::mutex> lock(mtx);
        callback = cb;
    }

    void trigger() {
        std::lock_guard<std::mutex> lock(mtx);
        if(callback) callback();
    }
};
```

## 总结

回调函数是异步编程的核心机制，它通过将"完成后的处理逻辑"作为参数传递，实现了：

1. 非阻塞调用，提高系统吞吐量
2. 模块解耦，调用方决定后续处理
3. 资源高效利用，避免线程空等

在C++中，我们可以通过多种方式实现回调，从简单的函数指针到现代的 `std::function` 和Lambda表达式。理解并正确使用回调函数，是成为高效C++程序员的重要一步。

lcg空间

喜欢作者

C/C++ · 目录

上一篇 · 【C++】typename和class关键字的使用方法和注意事项

个人观点，仅供参考