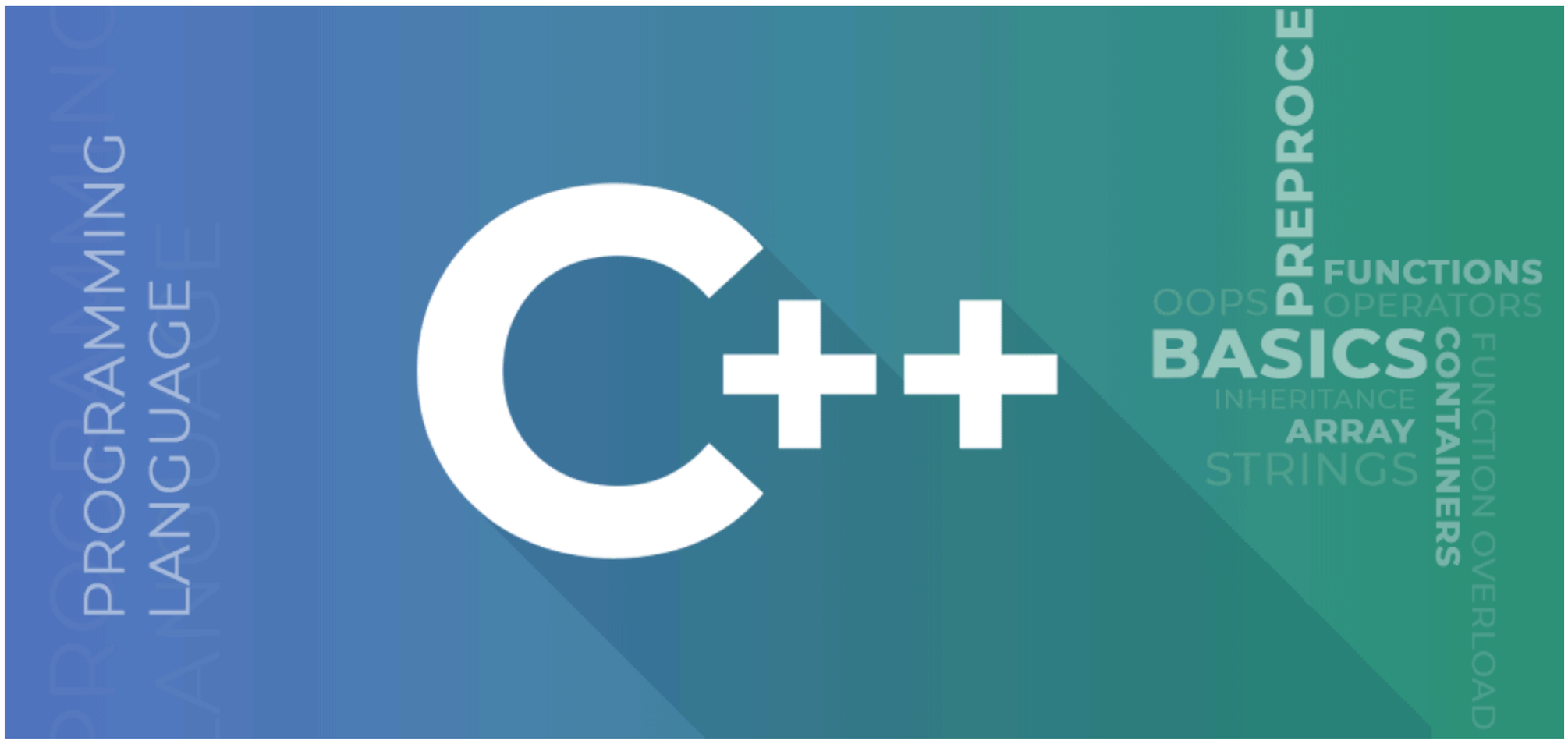
C++并发操作解密：轻松搞定数据同步

作者：微信公众号：【架构师老卢】 11-28 9:52



概述：在C++中，通过互斥锁解决并发数据同步问题。定义共享数据和互斥锁，编写线程函数，使用互斥锁确保操作的原子性。主函数中创建并启动线程，保障线程安全。实例源代码演示了简单而有效的同步机制。

在C++中解决并发操作时的数据同步问题通常需要使用互斥锁（Mutex）来确保线程安全。以下是详细的步骤以及附带的源代码示例：

步骤1：包含必要的头文件

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

步骤2：定义共享数据和互斥锁

```
// 共享的数据
int sharedData = 0;

// 互斥锁，用于保护共享数据
std::mutex mutex;
```

步骤3：编写线程函数

```
void threadFunction(int threadId) {
    for (int i = 0; i < 5; ++i) {
        // 使用互斥锁保护共享数据
        std::lock_guard<std::mutex> lock(mutex);

        // 对共享数据进行操作
        sharedData++;

        // 输出当前线程对共享数据的操作
        std::cout << "Thread " << threadId << ": Shared Data = " << sharedData << std::endl;
    }
}
```

步骤4：主函数中创建并启动线程

```
int main() {
    // 创建两个线程，并启动它们
    std::thread thread1(threadFunction, 1);
    std::thread thread2(threadFunction, 2);

    // 等待两个线程执行完毕
    thread1.join();
    thread2.join();

    return 0;
}
```

步骤5：编译和运行

使用C++编译器编译上述代码，并运行生成的可执行文件。观察输出结果，确认互斥锁成功保护了共享数据，避免了竞态条件和数据不一致性的问题。

以上步骤演示了一个基本的线程同步机制。在实际应用中，可能需要根据程序的需求选择更复杂的同步工具，如条件变量、信号量等。同时，注意控制互斥锁的粒度，以免过多地使用锁导致性能问题。