

华中科技大学

课程报告

课程名称：高级分布式系统

专业班级	计算机硕 2402
学 号	M202474038
姓 名	徐锦慧
授课老师	金海

2024 年 12 月 27 日

计算机科学与技术学院

目 录

1 写穿算法(WRITE THROUGH).....	1
1.1 实验目的.....	1
1.2 实验内容.....	1
1.3 实验方法.....	1
1.4 实验结果.....	3
2 PAXOS 算法.....	5
2.1 实验目的.....	5
2.2 实验内容.....	5
2.3 实验方法.....	5
2.4 实验结果.....	9
3 统一共享内存（USM）	10
3.1 实验目的.....	10
3.2 实验内容.....	10
3.3 实验方法.....	10
3.4 实验结果.....	12
4 总结与收获.....	14

1 写穿算法(Write Through)

1.1 实验目的

本实验的主要目的是通过实现写穿算法 (Write Through)，加深对分布式文件系统的理解和实践应用。写穿算法是一种缓存管理策略，旨在确保缓存与主存储数据的一致性。通过完成实验，学生将能够掌握分布式文件系统的基础概念、缓存一致性管理以及高效的读写操作。

1.2 实验内容

1) 理解分布式文件系统

学生需要掌握分布式文件系统的基本工作原理，了解如何通过多个节点协同工作来管理文件的存储和访问。并能够使用 API 接口进行文件操作，如文件创建、移动、删除和读写等。

2) 实现写穿算法

在本实验中，学生需要补充并实现写穿算法。该算法确保当用户修改高速缓存中的文件时，新的数据会立即写回到服务器，并保证读取操作时缓存与主服务器数据的一致性。学生需编写以下两个核心功能：

①写函数：实现当用户修改文件时，将新的数据写入缓存并同步到服务器

②读函数：当用户读取文件时，先与服务器进行版本号比对，若一致则从缓存中读取数据，否则从服务器获取最新的数据并更新缓存。

3) 测试功能实现

实验平台会自动进行测试，通过给定的输入调用实现的 read 和 write 函数，确保缓存管理的正确性和数据的一致性。

通过完成本实验，学生将能够深入理解缓存一致性协议，学习如何在分布式系统中高效管理文件的读取与写入操作。

1.3 实验方法

1. read 函数实现

`read` 函数的目标是从缓存中读取数据。如果缓存中没有该文件，或者缓存中的版本与服务器上的版本不一致，则需要从服务器读取数据，并将该数据缓存到本地。具体实现流程为：

- 1) 检查缓存：首先，检查缓存中是否已存在目标文件（`_search_cache`）。如果存在，则进一步检查缓存中的版本号是否与服务器上的版本号一致。如果一致，则直接返回缓存中的数据。
- 2) 从服务器读取：如果缓存中没有该文件，或者缓存的文件版本不一致，则从服务器读取文件。通过调用 `self._target_server.read(target_file)` 获取服务器上的文件。
- 3) 更新缓存：如果从服务器成功读取到文件，使用 `_write_cache` 方法将该文件数据写入缓存中。
- 4) 返回数据：最终返回文件的数据。如果文件不存在，则返回 `None`。

核心代码如下：

```
def read(self,target_file):
    #检查缓存中是否已缓存文件
    if self._search_cache(target_file):
        cache_file = self._cache_file[target_file].get_data()
        #向 server 确认 version 是否一致
        server_version = self._target_server.get_version(target_file)
        if cache_file[2] == server_version:
            #选择从 cache/server 中读取文件
            return cache_file[1]

    #缓存从 server 中读取的文件
    server_file = self._target_server.read(target_file)
    if server_file:
        # 将文件缓存到本地
        self._write_cache(target_file, server_file[1], server_file[2])
        return server_file[1]

    # there is no file in the system
    return
```

2. `write` 函数实现

`write` 函数的目标是将数据写入缓存和服务端。当文件的数据发生修改时，写穿算法要求同时将数据同步到缓存和服务端中。具体实现流程为：

- 1) 生成新版本号：每次写入数据时，我们会生成一个新的版本号。这个版本号

是基于当前时间戳（通过调用_get_new_version）生成的。

- 2) 更新缓存：使用_write_cache 方法将新的数据写入缓存，并更新缓存中的版本号。
- 3) 更新服务器：同时，使用 self._target_server.write 方法将数据写入服务器，确保服务器上的数据与缓存保持一致。

核心代码如下：

```
def write(self,target_file,data):
    #生成新 version
    new_version = self._get_new_version()

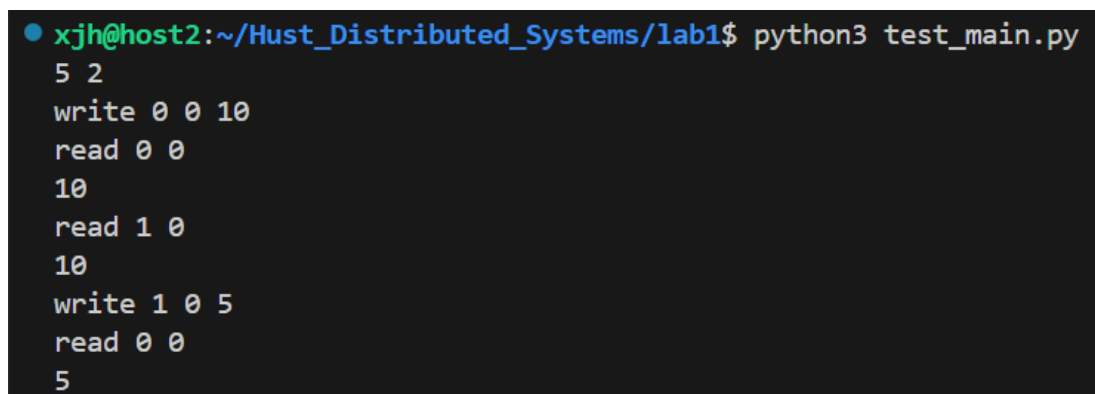
    #向 cache 中写入数据并更新 version
    self._write_cache(target_file, data, new_version)

    #向 server 中写入数据并更新 version
    self._target_server.write(target_file, data, new_version)

    return
```

1.4 实验结果

在自己的主机上测试用例一的结果如图 1-1 所示。对每个操作的具体分析如下：



```
xjh@host2:~/Hust_Distributed_Systems/lab1$ python3 test_main.py
5 2
write 0 0 10
read 0 0
10
read 1 0
10
write 1 0 5
read 0 0
5
```

图 1-1 实验 1 自测结果

- 1) write 0 0 10：用户 0 向文件 0 写入数据 10。在 write 操作中，数据会写入缓存并同步到服务器。这时，缓存中和服务器上的文件 0 的内容都变为 10。
- 2) read 0 0：用户 0 读取文件 0。在此时，文件 0 已经存在于缓存中，且缓存中的数据版本与服务器一致，因此用户 0 读取到的是缓存中的数据，即 10。
- 3) read 1 0：用户 1 读取文件 0。由于文件 0 已被用户 0 写入且缓存中存在该文

件，且缓存中的版本与服务器一致，所以用户 1 读取到的仍然是缓存中的数据，即 10。

- 4) **write 1 0 5**: 用户 1 再次向文件 0 写入数据 5。在 write 操作中，数据 5 被写入缓存并同步到服务器，更新了缓存和服务器中的文件 0 的内容为 5。
- 5) **read 0 0**: 用户 0 再次读取文件 0。此时，文件 0 在缓存中的数据已经被更新为 5(由用户 1 的写操作更新)。因此，用户 0 读取到的是缓存中的最新数据，即 5。

在头歌上通过了所有测试用例，结果如图 1-2 所示。

测试结果			
✔ 6/6 全部通过			
▶ 测试集1	消耗内存80.96MB	代码执行时长: 0.05秒	✔
▶ 测试集2	消耗内存80.96MB	代码执行时长: 0.03秒	✔
▶ 测试集3	消耗内存80.96MB	代码执行时长: 0.03秒	✔
▶ 测试集4	消耗内存80.96MB	代码执行时长: 0.05秒	✔
▶ 测试集5	消耗内存80.96MB	代码执行时长: 0.03秒	✔
▶ 测试集6	消耗内存80.96MB	代码执行时长: 0.05秒	✔

图 1-2 实验 1 头歌结果

2 Paxos 算法

2.1 实验目的

本实验的目的是通过实现 Paxos 算法的核心成员函数，深入理解 Paxos 算法的工作原理，掌握如何设计和实现分布式一致性算法。通过实现 Proposer、Acceptor 等角色，模拟分布式系统中各个节点如何进行提案、接受提案并达成一致，最终实现分布式系统的高可用性与一致性。

在实验中，将重点学习以下内容：

- 1) Paxos 算法的工作原理：理解如何通过提案、接受和学习等步骤来达成一致。
- 2) 分布式一致性：掌握在不同节点间达成一致的方法，确保多个分布式节点即使在面对网络延迟或部分故障时，仍能达成一致。
- 3) 角色分工与协作：了解 Paxos 中各个角色（Proposer、Acceptor、Learner）的职责，并实现其在算法中的交互。
- 4) 核心成员函数的设计与实现：通过编程实践，完成 Proposer 和 Acceptor 类的关键成员函数，推动 Paxos 算法的正确执行。

2.2 实验内容

根据任务描述，要求实现 Paxos 算法中的核心部分，主要包括 Proposer 类和 Acceptor 类的功能。具体要求如下：

- 1) Proposer 类：作为提议者，Proposer 会发起提案并与 Acceptor 进行交互。
在 Proposer 类中实现提案流程，包括提议编号的生成、发送提案以及等待确认等操作。
- 2) Acceptor 类：作为批准者，Acceptor 会接受或拒绝提案。在 Acceptor 类中实现提案的接受机制，包括对提案的验证和响应等。Acceptor 会在收到 Proposer 提案后，根据提案编号和当前状态决定是否接受提案。

2.3 实验方法

1. Paxos 算法设计

Paxos 算法的目标是在分布式系统中就某个值（proposal value）达成一致，保证多个分布式节点在面对网络故障、消息丢失等问题时能够保持一致性。Paxos 算法有三个主要角色：Proposer（提议者）、Acceptor（接受者）和 Learner（学习者）。算法流程图如图 2-1 所示。

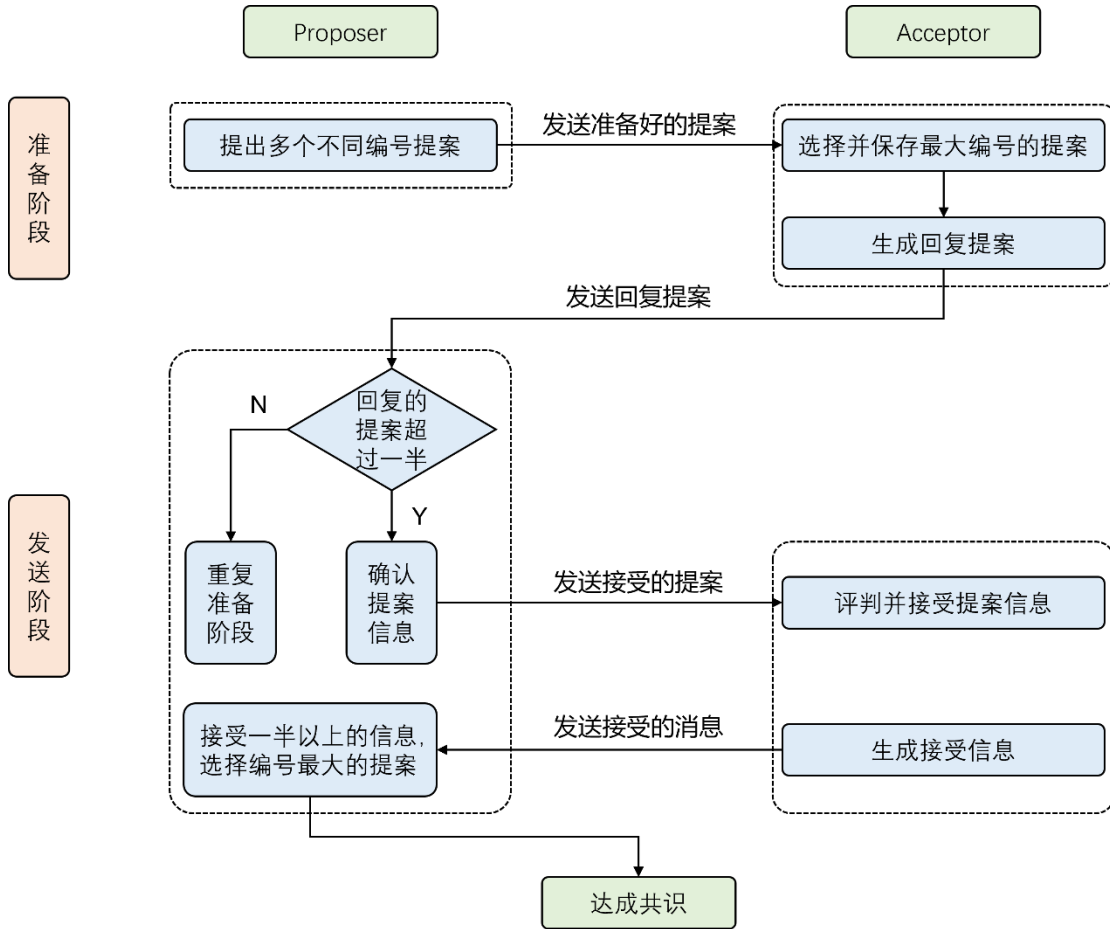


图 2-1 Paxos 算法流程图

1) 角色定义

Proposer: 提出某个值并请求 Acceptor 进行投票。Proposer 会根据一定规则发起提案。

Acceptor: 对 Proposer 提出的值进行投票，决定是否接受提案。当 Acceptor 接受了某个提案，它会告知所有 Learner 该提案已被选中。

Learner: 从 Acceptor 处得知最终的选定值，即被选定的提案值。

2) Paxos 算法工作流程

在 Prepare 阶段，Proposer 会生成一个提案编号，并向多个 Acceptor 发送 Prepare 请求。该请求要求 Acceptor 回复它已经接受的最大提案编号以及该提案编号对应的值。Proposer 通过这些信息来确认它是否可以提交一个新的提案。

Proposer 发送 Prepare 请求，并带上一个唯一的编号 N（一般为递增的整数）。Acceptor 收到 Prepare 请求后，如果该提案编号 N 大于它已经接受的任何提案编号，它会返回一个承诺，承诺不会再接受编号小于 N 的提案。Acceptor 还会返回它接受过的最大编号的提案值。

在 Accept 阶段，Proposer 在收到足够的 Prepare 响应后，会向 Acceptor 提交提案值，并请求其接受。Proposer 提交的提案值必须是 Acceptor 已经接受的最大编号的提案值，或者是 Proposer 自己提出的新值。如果 Proposer 收到的响应足够多（多数），它会提交提案并向 Acceptor 请求接受。Acceptor 收到 Accept 请求后，如果该提案编号 N 大于它已经接受的编号，它会接受该提案，并通知所有 Learner。

2. Proposer 实现

在本实验中，需要实现 Proposer 的两个关键方法：Proposed 和 Accepted，分别对应 Paxos 算法的第一阶段（Prepare 阶段）和第二阶段（Accept 阶段）的提案处理逻辑。下面分别介绍两个函数的实现。

1) Proposed 函数实现

首先检查提案是否完成：如果 m_proposeFinished 为 true，说明当前提案已经成功完成，Proposer 不需要再处理后续的响应，直接返回 true。

接着处理拒绝响应。如果 Acceptor 的响应 ok 为 false，表示该 Acceptor 拒绝了 Proposer 的提案，Proposer 增加拒绝计数 m_refuseCount。如果拒绝的次数超过一半（即 $m_refuseCount > m_acceptorCount / 2$ ），Proposer 会认为当前提案失败。这时 Proposer 修改提案的 serialNum（提案编号），并调用 StartPropose(m_value) 重新开始 Propose 阶段，发起一个新的提案。如果拒绝的次数不到一半，Proposer 继续等待其他 Acceptor 的响应。关键代码如下：

```
if (m_refuseCount > m_acceptorCount / 2) {  
    m_value.serialNum = m_value.serialNum + m_proposerCount;  
    StartPropose(m_value);  
    return false;  
}
```

处理接受响应。如果 Acceptor 同意了 Proposer 的提案（即 ok 为 true），则增加接受计数 m_okCount。Proposer 根据 Acceptor 返回的 lastAcceptValue 更新自己的提案：如果 lastAcceptValue.serialNum（Acceptor 接受的提案编号）大于当前

Proposer 记录的 `m_maxAcceptedSerialNum` (Proposer 已知的最大接受提案编号), 则更新 `m_maxAcceptedSerialNum`, 并更新 Proposer 的提案值 `m_value.value` 为 `lastAcceptValue.value`。关键代码如下:

```
if(lastAcceptValue.serialNum > m_maxAcceptedSerialNum) {
    m_maxAcceptedSerialNum = lastAcceptValue.serialNum;
    m_value.value = lastAcceptValue.value;
}
```

提案达成一致的判断。当 Proposer 收到超过一半 Acceptor 的接受响应时 (即 `m_okCount > m_acceptorCount / 2`), Proposer 认为提案已经完成, 设置 `m_proposeFinished` 为 `true`, 表示该提案已被选定。

2) Accepted 函数实现

检查提案是否完成。如果 `m_proposeFinished` 为 `true`, 说明 Proposer 已经完成了当前提案, Acceptor 的响应是迟到的, 不需要再处理, 因此直接返回 `true`。

处理拒绝响应。如果 Acceptor 拒绝了当前提案, Proposer 增加拒绝计数 `m_refuseCount`。如果拒绝的次数超过一半, Proposer 认为当前提案失败, 修改提案编号 `serialNum` 并重新发起新的 Propose 阶段。如果拒绝的次数不到一半, Proposer 继续等待其他 Acceptor 的响应。

```
if (m_refuseCount > m_acceptorCount / 2) {
    m_value.serialNum = m_value.serialNum + m_proposerCount;
    StartPropose(m_value);
    return false;
}
```

处理接受响应。如果 Acceptor 同意了当前提案, Proposer 增加接受计数 `m_okCount`。当 `m_okCount` 超过总接受者数的一半时, Proposer 认为提案已达成一致, 设置 `m_isAgree` 为 `true`。

3. Acceptor 实现

和 Proposer 类似, Acceptor 类主要实现了两个函数: Propose 和 Accept, 它们分别对应 Paxos 协议的不同阶段。

1) Propose 函数

函数首先检查 Proposer 发送的序列号是否有效 (不为 0)。接着判断 Proposer 的提案序列号是否大于当前 Acceptor 已接受的最大序列号, 如果小于或等于, 则拒绝。如果通过检查, 更新 Acceptor 的最大序列号并返回当前 Acceptor 已接受的提案值。关键代码如下:

```
m_maxSerialNum = serialNum;           // 更新最大流水号
lastAcceptValue = m_lastAcceptValue;  // 返回已接受的提议
```

2) Accept 函数

首先检查 Proposer 发来的提案序列号是否有效（不为 0）。接着判断提案的序列号是否大于当前 Acceptor 已接受的最大序列号，如果小于或等于，则拒绝该提案。如果提案有效，Acceptor 批准并保存该提案。关键代码如下：

```
if (m_maxSerialNum > value.serialNum) return false;  
m_lastAcceptValue = value;           // 更新最新已接收的提议
```

2.4 实验结果

根据以上方案实现 Paxos 算法，可以顺利通过头歌平台的测试。运行结果如图 2-2 所示：



图 2-2 运行结果

3 统一共享内存（USM）

3.1 实验目的

- 1) 理解分布式共享内存（DSM）：掌握分布式共享内存的概念及其提供的逻辑统一地址空间。理解 DSM 如何利用消息传递实现对全局地址空间的透明访问。
- 2) 掌握统一共享内存模型（USM）：学习 USM 模型的基本原理，了解其在异构系统中的应用。掌握 USM 支持的三种内存分配类型（设备内存、主机内存、共享内存）的特点和适用场景。
- 3) 熟悉 Intel oneAPI 中的 USM 内存管理 API：掌握 SYCL_queue 类和 SYCL_memory 类的基本操作。通过模拟实现，了解 USM 模型在异构编程中的应用场景。

3.2 实验内容

- 1) 熟悉 USM 内存管理 API：根据代码提示完成编程，并能够通过正确性验证。
- 2) 基于 USM 模型完成代码实现：设计实现指定功能的 kernel 函数，并在 Host 端获取正确计算结果。

3.3 实验方法

该实验一共分为两个关卡，以下分别介绍两个关卡的实现。

1. 熟悉 USM 内存管理 API

该关卡使用模拟的 SYCL 框架完成了统一共享内存（USM）模型的操作，包括内存分配、数据初始化、设备并行计算、结果传输及验证，最后释放资源。主要流程如下：

- 1) 设备队列初始化：使用 SYCL_queue 类初始化一个与设备（GPU）绑定的计算队列 dev_q。使用 init_queue_from_device(0, "gpu")方法指定使用 ID 为 0 的 GPU 设备。
- 2) USM 共享内存分配：调用 malloc_shared()在 dev_q 队列所绑定的设备中分配共享内存 dev_data。共享内存支持主机（CPU）和设备（GPU）之间的透明数据访问。

- 3) 数据初始化：使用循环遍历 `dev_data.mem`，将共享内存的每个元素初始化为索引值 `i`。
- 4) 数据传输：调用 `copy_from_device()` 将设备共享内存 `dev_data` 的数据拷贝到主机分配的内存 `data` 中。
- 5) 核函数定义：核函数 `kernel` 接收一个共享内存对象 `input`，对其进行逐元素操作。每个元素的值更新为原值与主机内存 `data` 中对应位置值的乘积。
- 6) 并行计算：通过 `parallel_for()` 将核函数 `kernel` 提交到设备队列 `dev_q` 上运行，传入共享内存 `dev_data` 作为参数。调用 `copy_from_device()` 将设备共享内存 `dev_data` 中的计算结果拷贝到主机内存 `result` 中。

关键代码如下：

```
# STEP1: Create device queue on the gpu-0
dev_q = SYCL_queue()
dev_q.init_queue_from_device(0, "gpu")

# STEP2: Create USM shared allocation dev_data in device queue for data
dev_data = SYCL_memory()
dev_data.malloc_shared(dev_q, N)

# STEP3: Initialize USM shared allocation
for i in range(N):
    dev_data.mem[i] = i

# STEP4: Copy USM shared allocation to host allocation
data.copy_from_device(dev_data)

# STEP5: Write kernel code to update shared data on device with the product of host data and
device shared data
def kernel(input):
    for i in range(len(input.mem)):
        input.mem[i] = input.mem[i] * data.mem[i]

# STEP6: Run kernel in device queue, wait until the kernel is complete and obtain the result
on the host side
dev_q.parallel_for(kernel, dev_data).wait()
```

2. 基于 USM 模型完成代码实现

该关卡模拟使用统一共享内存（USM）在 GPU 上进行计算，包括以下操作：

- 1) 初始化两个数组 `data1` 和 `data2`。
- 2) 分别将两个数组的数据传输到 GPU 上的共享内存中。
- 3) 在 GPU 上分别对 `data1` 和 `data2` 执行计算，包括：`data1` 元素平方；`data2` 元素乘以 2；`data1` 和 `data2` 逐元素相加。
- 4) 将计算结果传回主机内存并验证正确性。
- 5) 最后释放所有分配的内存资源。

代码实现部分和第一关类似，主要是 kernel 函数部分有所不同，核心代码如下：

```
# STEP3 : Write kernel code to update data1 on device with square of value, and Run the
kernel function on device
def kernel1(input):
    for i in range(len(input.mem)):
        input.mem[i] = input.mem[i] * input.mem[i]
    dev_q.parallel_for(kernel1, dev_data1).wait()

# STEP4 : Write kernel code to update data2 on device with twice of value, and run the
kernel function on device
def kernel2(input):
    for i in range(len(input.mem)):
        input.mem[i] = input.mem[i] * 2
    dev_q.parallel_for(kernel2, dev_data2).wait()

# STEP5 : Write kernel code to add data2 on device to data1, and run the kernel function on
device
def kernel3(input1, input2):
    for i in range(len(input1.mem)):
        input1.mem[i] = input1.mem[i] + input2.mem[i]
    dev_q.parallel_for(kernel3, dev_data1, dev_data2).wait()
```

3.4 实验结果

根据以上方案实现统一共享内存（USM）模型编程，可以顺利通过头歌平台的测试。第一关和第二关的运行结果分别如图 3-1 和图 3-2 所示。



图 3-1 第一关运行结果

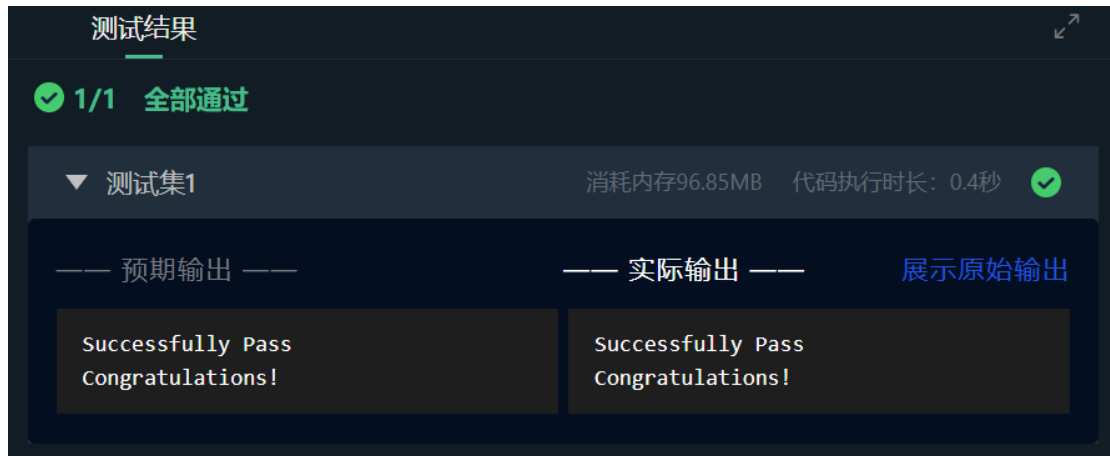


图 3-2 第二关运行结果

4 总结与收获

在本课程的实验中，我分别完成了写穿算法（Write Through）、Paxos 算法以及统一共享内存（USM）三个实验。

写穿算法通过在更新缓存数据时同步更新主存，确保主存数据的一致性。在改实验中，我完成了写穿逻辑的实现与测试，包括对不同数据访问模式的支持，以及写操作在主存和缓存之间的同步机制。通过实验，我深刻体会到写穿算法在分布式系统中保障一致性的重要性，同时认识到其对性能的影响。

Paxos 作为分布式系统中经典的一致性算法，其核心在于通过提议者、接受者和学习者的协作，确保分布式环境中节点对决策的一致认知。在该实验中，我实现了 Proposer 和 Acceptor 两个模块的主要逻辑，并分析了其在不同场景下的表现，如提议被拒绝或修改的情况。实验让我理解了 Paxos 的运行机制和其高容错性的理论基础，同时也认识到其复杂性带来的实现与性能挑战。

USM 模型通过统一虚拟地址空间简化了异构编程中主机与设备之间的数据迁移。在该实验中，我熟悉了 USM 内存管理 API，并在多个场景下实现了基于 USM 的计算任务，如数据的并行初始化、核函数执行、数据迁移与共享等。实验展示了 USM 在分布式系统中提升开发效率和性能的优势。