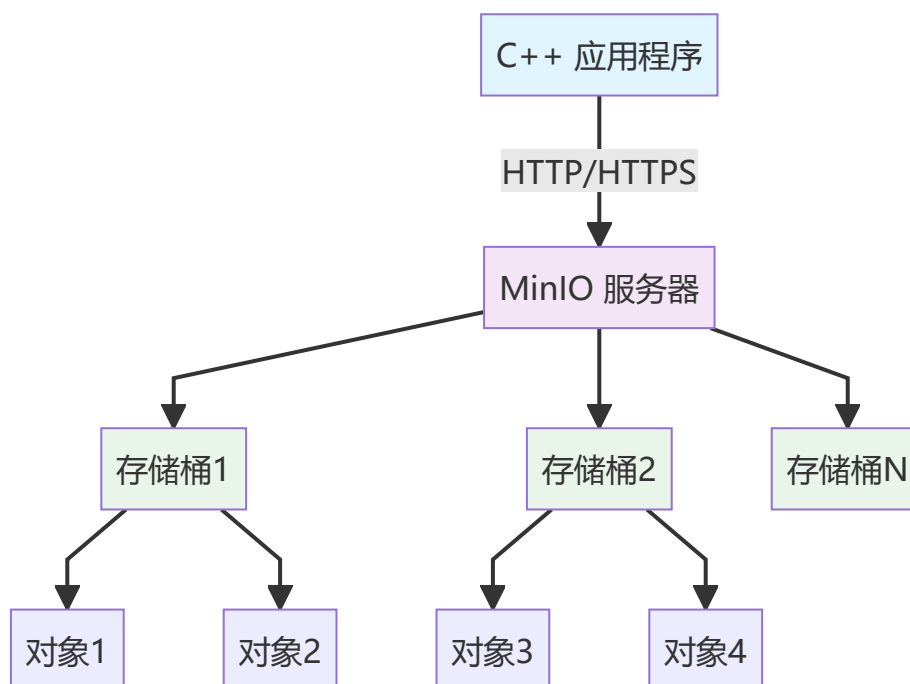


1 MinIO 基础概念

1.1 核心概念

- **MinIO服务器**: 高性能的对象存储服务器, 兼容Amazon S3 API
- **存储桶(Bucket)**: 存储对象的容器, 类似于文件夹
- **对象(Object)**: 存储在桶中的文件, 可以是任意类型的数据
- **访问密钥**: 用于身份验证的密钥对(Access Key + Secret Key)

1.2 系统架构



2 minio非docker单机快速部署

2.1 下载 MinIO 二进制文件

首先, 你需要从 MinIO 官方 GitHub 发布页面下载最新版本的 MinIO 二进制文件。根据你的操作系统选择合适的版本。

对于 Linux 系统, 你可以使用 `wget` 或者 `curl` 命令来下载 MinIO 的二进制文件。例如, 在 Linux 上你可以这样操作:

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio --no-check-certificate
```

如果你使用的是 macOS 或者 Windows，请访问 [MinIO 官方发布页面](#) 找到适合你系统的版本并下载。

2.2 赋予执行权限

下载完成后，你需要为下载的 MinIO 二进制文件添加执行权限。在 Linux 或 macOS 上，你可以通过以下命令实现：

```
chmod +x minio
```

2.3 设置环境变量（可选）

为了增强安全性，建议设置 `MINIO_ROOT_USER` 和 `MINIO_ROOT_PASSWORD` 这两个环境变量来指定 MinIO 的 root 用户和密码。如果不设置，默认会生成随机凭证，这将使得登录变得困难。

在 Linux 或 macOS 中，你可以通过如下命令设置环境变量：

```
export MINIO_ROOT_USER='minioadmin'  
export MINIO_ROOT_PASSWORD='minioadmin'
```

请确保替换 `'minioadmin'` 和 `'minioadmin'` 为你自己的用户名和密码。

2.4 启动 MinIO 服务器

现在，你可以通过运行以下命令启动 MinIO 服务器：

```
mkdir data  
./minio server ./data
```

在这个命令中，`/data` 是你希望 MinIO 存储数据的目录路径。你可以根据需要更改这个路径。确保该目录存在并且有适当的读写权限。

启动后，你会看到类似如下的输出，其中包含了访问 MinIO 控制台的 URL、Access Key 和 Secret Key：

```
Endpoint:  http://192.168.80.128:9000  http://127.0.0.1:9000  
AccessKey: YOUR-ACCESSKEYHERE  SecretKey: YOUR-SECRETKEYHERE  
Browser Access:  
    http://192.168.80.128:9000  http://127.0.0.1:9000
```

完整打印：

```

lqf@ubuntu:~/minio$ ./minio server ./data
INFO: Formatting 1st pool, 1 set(s), 1 drives per set.
INFO: WARNING: Host local has more than 0 drives of set. A host failure w
MinIO Object Storage Server
Copyright: 2015-2025 MinIO, Inc.
License: GNU AGPLv3 - https://www.gnu.org/licenses/agpl-3.0.html
Version: RELEASE.2025-07-23T15-54-02Z (go1.24.5 linux/amd64)

API: http://192.168.80.128:9000 http://172.17.0.1:9000 http://127.0.0.1
RootUser: minioadmin
RootPass: minioadmin

WebUI: http://192.168.80.128:40285 http://172.17.0.1:40285 http://127.0.0.1
RootUser: minioadmin
RootPass: minioadmin

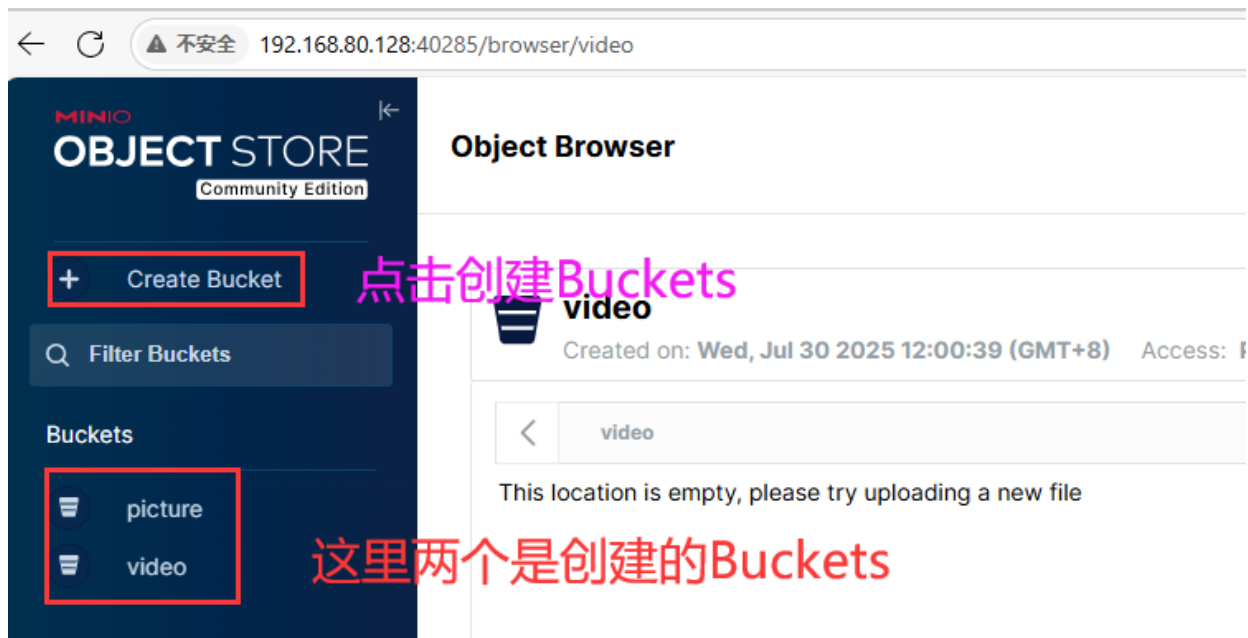
CLI: https://min.io/docs/minio/linux/reference/minio-mc.html#quickstart
$ mc alias set 'myminio' 'http://192.168.80.128:9000' 'minioadmin' 'mi

Docs: https://docs.min.io
WARN: Detected default credentials 'minioadmin:minioadmin', we recommend
ROOT_USER' and 'MINIO_ROOT_PASSWORD' environment variables

```

2.5 访问 MinIO 控制台

打开浏览器，输入提供的 URL（通常是 `http://<your-machine-ip>:9000`，比如我目前是<http://192.168.80.128:9000>），然后使用提供的 Access Key 和 Secret Key 登录 MinIO 控制台。



3 C++ SDK上传和下载文件

3.1 官方C++ sdk安装

需要依赖vcpkg

```
# 下载vcpkg
git clone https://github.com/microsoft/vcpkg.git
# 进入vcpkg
cd vcpkg
./bootstrap-vcpkg.sh

# 安装minio-cpp, 时间有点久, 耐心等待
./vcpkg install minio-cpp
```

3.2 编译范例

以cmake方式构建的范例工程

```
mkdir build && cd build
cmake ..
make
```

3.3 执行程序

3.3.1 minio_basic文件上传下载

```
# 将源码目录的test-file.txt拷贝到build目录
lqf@ubuntu:~/minio/app/build$ cp ../test-file.txt .
lqf@ubuntu:~/minio/app/build$ ./minio_basic
```

执行成功打印:

```
开始上传文件到MinIO...
文件上传成功, ETag:
存储位置: video/test-file.txt

=== 上传响应详细信息 ===
状态码: 0
ETag: d249d192f7ab36c408f21125c2bc44d0
版本ID: 无
请求ID: 无
主机ID: 无
开始从MinIO下载文件...
文件下载成功!
保存位置: ./downloaded-file.txt
程序执行完成!
```

下载的内容保存为downloaded-file.txt。

3.3.2 minio_stream文件流式上传

```
# 将源码目录的time.flv拷贝到build目录
lqf@ubuntu:~/minio/app/build$ cp ../time.flv .
lqf@ubuntu:~/minio/app/build$ ./minio_stream time.flv
```

执行成功打印：

```
=== 开始模拟web上传流式传输 ===
源文件: time.flv
目标位置: video/time.flv
每次读取: 32768 字节 (32KB)
文件总大小: 20184976 字节 (19711KB)

文件大于等于5MB, 使用Multipart Upload...
Multipart Upload创建成功, Upload ID:
Zju2YzkyMDMtn2VjOS00MDM4LTlkY2MtnZRhYTM3ZjA1MTlkLjgxNjEzZTk2LTc3MTUtNDC4MS04NWFjLWY0
ZWU5NjZhMTE4NngxNzUzODkyODQ4NjM1NTA0NTI1
从文件读取到内存: 32768 字节 (总计: 32768/20184976)
.....

从文件读取到内存: 32768 字节 (总计: 5242880/20184976)
从内存上传分块 1 到MinIO, 大小: 5242880 字节
分块 1 上传成功, ETag: d871d94bf0052b021048d9bf49d09ce9
从文件读取到内存: 32768 字节 (总计: 5275648/20184976)
从文件读取到内存: 32768 字节 (总计: 5308416/20184976)
.....

从文
从文件读取到内存: 32768 字节 (总计: 10485760/20184976)
从内存上传分块 2 到MinIO, 大小: 5242880 字节
分块 2 上传成功, ETag: f53d0a4fa53433a602045abafdd78fd3
从文件读取到内存: 32768 字节 (总计: 10518528/20184976)
从文件读取到内存: 32768 字节 (总计: 10551296/20184976)
.....

从文件读取到内存: 32768 字节 (总计: 15695872/20184976)
从文件读取到内存: 32768 字节 (总计: 15728640/20184976)
从内存上传分块 3 到MinIO, 大小: 5242880 字节
分块 3 上传成功, ETag: bb624d69050b1e1eec98897d64aa6e5c
从文件读取到内存: 32768 字节 (总计: 15761408/20184976)
从文件读取到内存: 32768 字节 (总计: 15794176/20184976)
.....

从文件读取到内存: 32768 字节 (总计: 20119552/20184976)
从文件读取到内存: 32768 字节 (总计: 20152320/20184976)
从文件读取到内存: 32656 字节 (总计: 20184976/20184976)
从内存上传分块 4 到MinIO, 大小: 4456336 字节
```

分块 4 上传成功, ETag: 0ee96f2a4d95d43db356889f57f50cbe

完成Multipart Upload...

=== 大文件上传完成 ===

文件上传成功!

总分块数: 4

最终ETag:

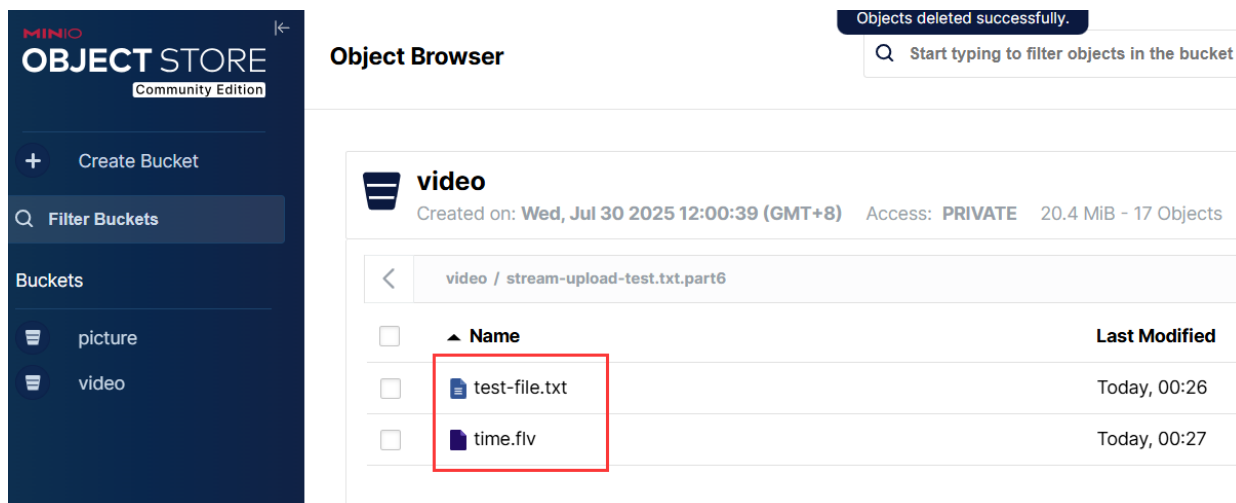
文件位置:

注意: 文件已成功上传为完整文件。

这模拟了从文件读取32KB数据到内存, 然后从内存上传到MinIO的场景。

=== 程序执行完成 ===

3.4 查看上传的文件



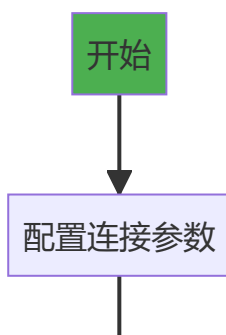
The screenshot shows the MinIO Object Browser interface. On the left is a sidebar with the 'OBJECT STORE' logo and a 'Create Bucket' button. Below it is a search bar and a list of buckets: 'picture' and 'video'. The main area is titled 'Object Browser' and shows a bucket named 'video' with details: 'Created on: Wed, Jul 30 2025 12:00:39 (GMT+8)', 'Access: PRIVATE', and '20.4 MiB - 17 Objects'. A search bar is at the top right. Below the bucket details, a list of objects is shown. The first object is 'test-file.txt' and the second is 'time.flv'. Both objects are highlighted with a red box. The 'Last Modified' column shows 'Today, 00:26' for 'test-file.txt' and 'Today, 00:27' for 'time.flv'.

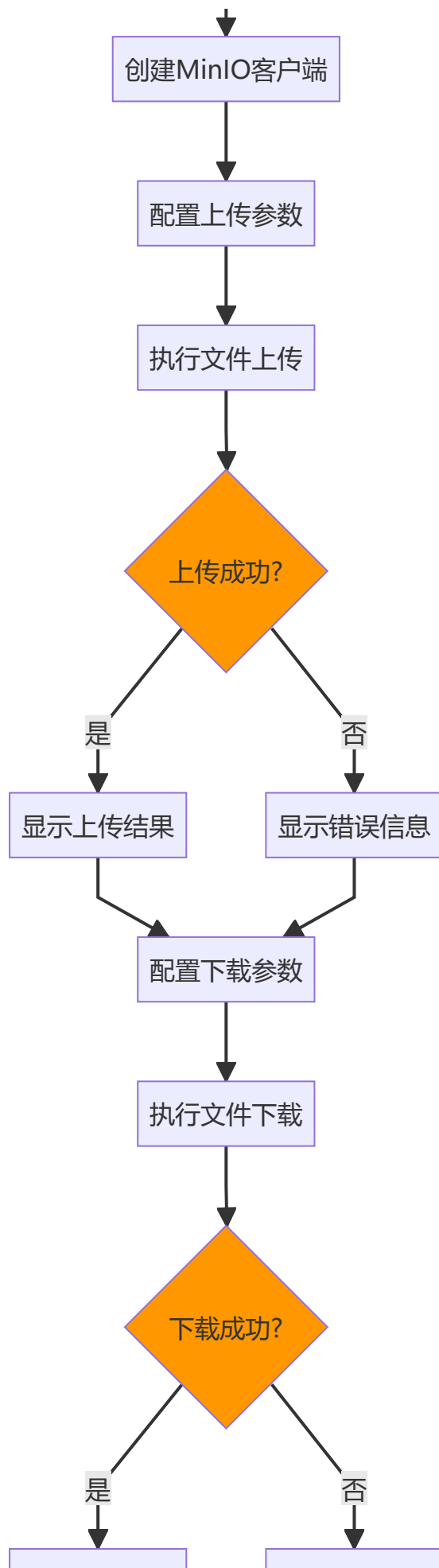
4 范例解析-minio_basic文件上传下载

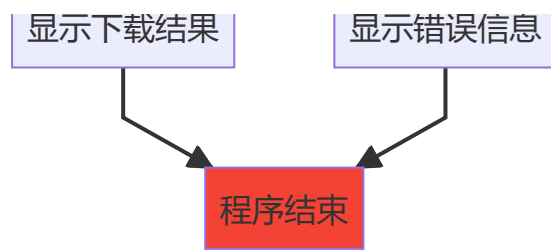
本文档将详细介绍 `minio_basic.cpp` 文件中的代码实现, 帮助开发者理解如何使用 MinIO C++ SDK 进行对象存储的基本操作, 包括文件上传和下载。

4.1 程序流程分析

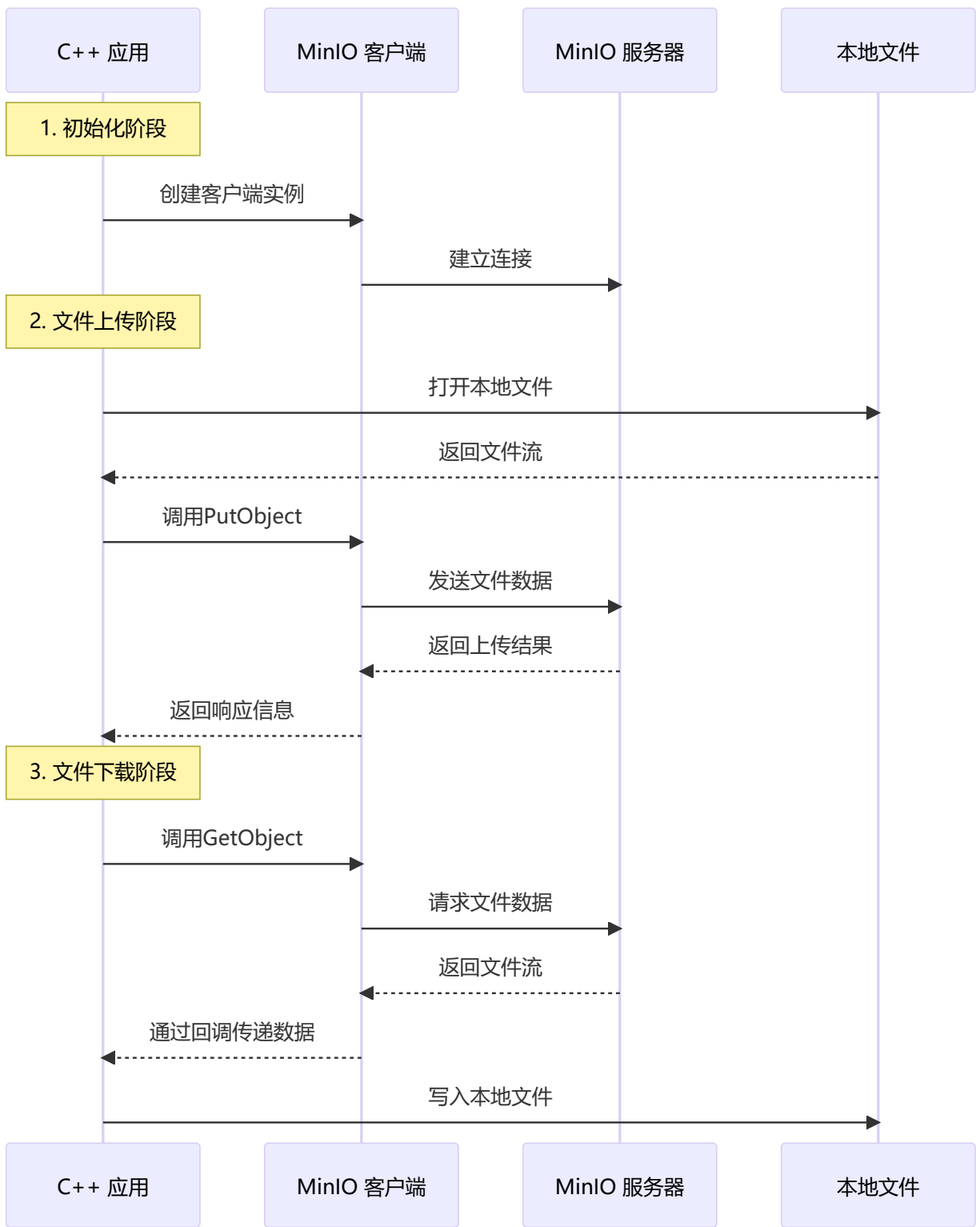
4.1.1 整体流程图







4.1.2 详细操作流程



4.2 代码详解

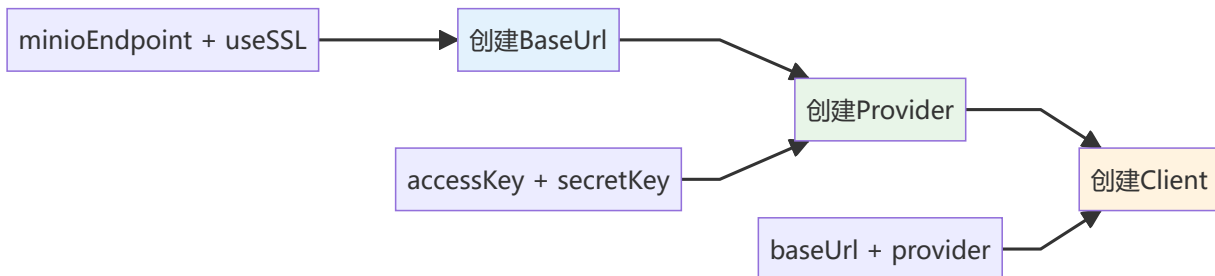
4.2.1 连接配置部分

```
// MinIO服务器连接配置
std::string minioEndpoint = "localhost:9000"; // 服务器地址
std::string accessKey = "minioadmin";        // 访问密钥
std::string secretKey = "minioadmin";        // 秘密密钥
bool useSSL = false;                        // 是否使用SSL
```

配置说明:

- `minioEndpoint`: MinIO服务器的地址和端口
- `accessKey/secretKey`: 用于身份验证的密钥对
- `useSSL`: 控制是否使用HTTPS连接

4.2.2 客户端初始化



```
// MinIO服务器地址和端口
// 本地部署: localhost:9000
// 远程服务器: your-server-ip:9000
// 使用HTTPS: your-server-ip:9000
std::string minioEndpoint = "localhost:9000";

// 访问密钥ID (Access Key ID)
// 在MinIO控制台 -> Identity -> Users 中创建用户获取
// 默认管理员账号的Access Key是: minioadmin
std::string accessKey = "minioadmin";

// 秘密访问密钥 (Secret Access Key)
// 在MinIO控制台 -> Identity -> Users 中创建用户获取
// 默认管理员账号的Secret Key是: minioadmin
std::string secretKey = "minioadmin";

// 是否使用SSL/TLS加密连接
// true: 使用HTTPS (https://)
// false: 使用HTTP (http://)
bool useSSL = false;

// ===== 创建MinIO客户端 =====
```

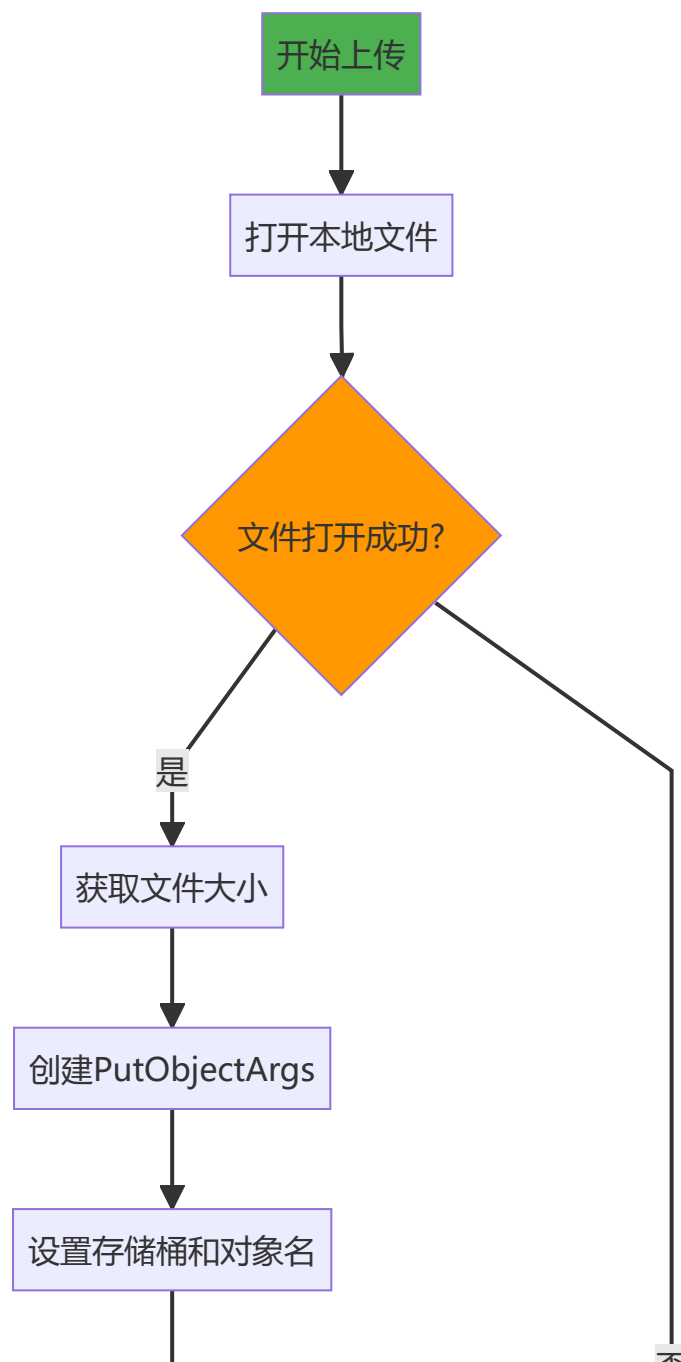
```
// 初始化MinIO客户端对象，建立与服务器的连接
minio::s3::BaseUrl baseUrl(minioEndpoint, useSSL);
minio::creds::StaticProvider provider(accessKey, secretKey);
minio::s3::Client minio(baseUrl, &provider);
```

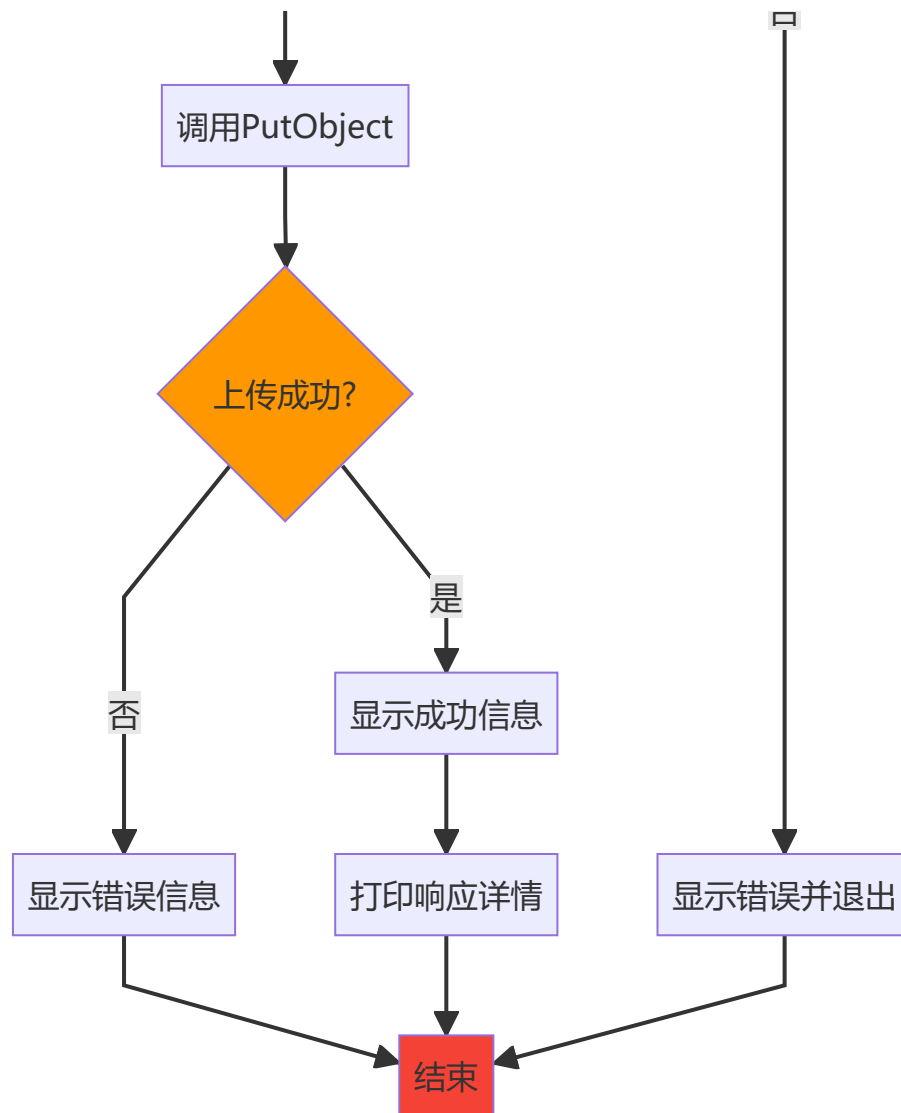
组件说明:

- `BaseUrl`: 封装服务器地址和协议信息
- `StaticProvider`: 提供静态的访问凭证
- `Client`: MinIO操作的主要接口

4.2.3 文件上传实现

4.2.3.1 上传流程图





4.2.3.2 核心代码分析

```
// 1. 打开文件流
std::ifstream fileStream(filePath, std::ios::binary);
if (!fileStream.is_open()) {
    std::cerr << "无法打开文件: " << filePath << std::endl;
    return 1;
}

// 2. 获取文件大小
fileStream.seekg(0, std::ios::end);
long fileSize = fileStream.tellg();
fileStream.seekg(0, std::ios::beg);

// 3. 创建上传参数
minio::s3::PutObjectArgs args(fileStream, fileSize, 0);
args.bucket = bucketName;
args.object = objectName;

// 4. 执行上传
```

```
minio::s3::PutObjectResponse resp = minio.PutObject(args);
```

参数说明:

- `fileStream`: 文件输入流, 以二进制模式打开
- `filesize`: 文件大小, 用于告知服务器预期数据量
- `args.bucket`: 目标存储桶名称
- `args.object`: 对象在存储桶中的名称

4.2.4 响应处理

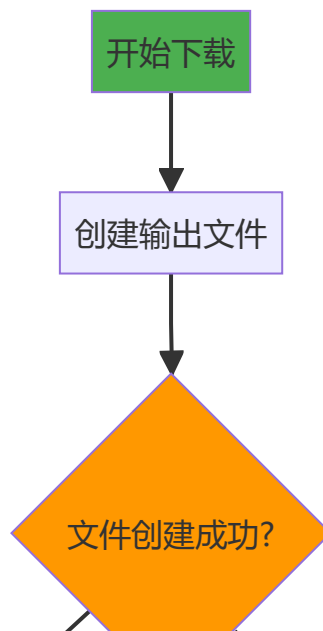
```
if (!resp) {  
    std::cerr << "上传失败: " << resp.Error().String() << std::endl;  
    return 1;  
}  
  
// 获取响应信息  
std::string etag = resp.headers.GetFront("etag");  
std::cout << "状态码: " << resp.status_code << std::endl;  
std::cout << "ETag: " << resp.etag << std::endl;
```

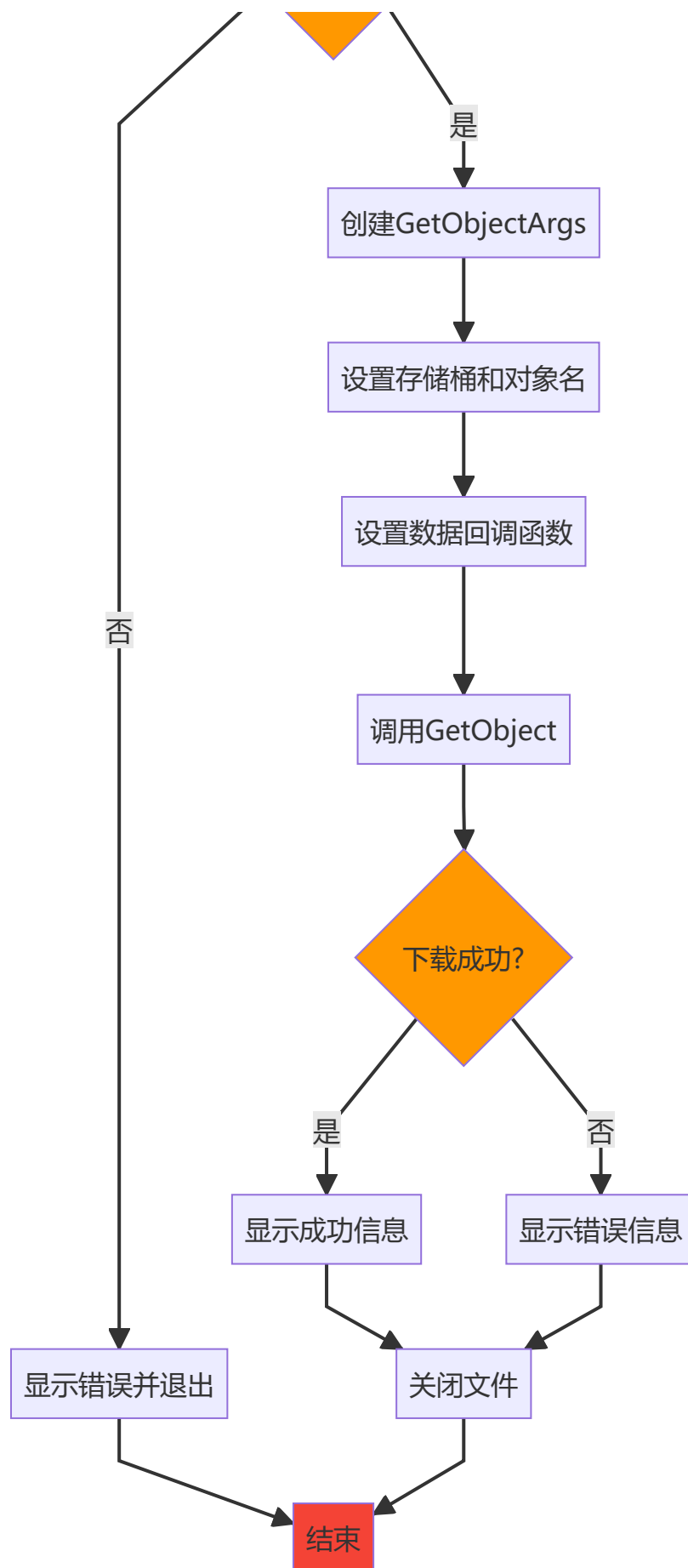
响应字段说明:

- `status_code`: HTTP状态码
- `etag`: 对象的唯一标识符
- `version_id`: 对象版本ID (如果启用版本控制)
- `headers`: 完整的HTTP响应头

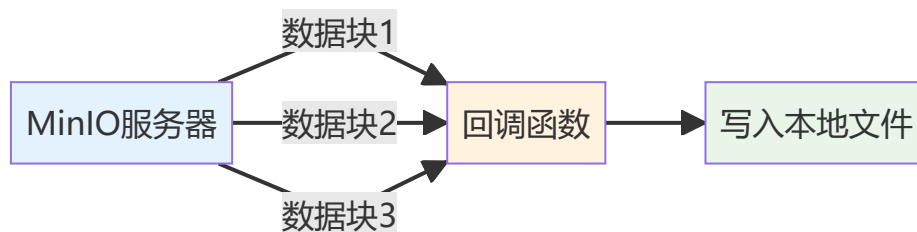
4.2.5 文件下载实现

4.2.5.1 下载流程图





4.2.5.2 回调函数机制



```
// 设置数据回调函数
args.datafunc = [&outFile](minio::http::DataFunctionArgs dataArgs) -> bool {
    outFile.write(dataArgs.datachunk.c_str(), dataArgs.datachunk.length());
    return true;
};
```

回调函数说明:

- 采用Lambda表达式实现
- `dataArgs.datachunk`: 接收到的数据块
- 返回 `true` 表示继续接收, 返回 `false` 会中断下载

5 范例解析minio_stream文件流式上传

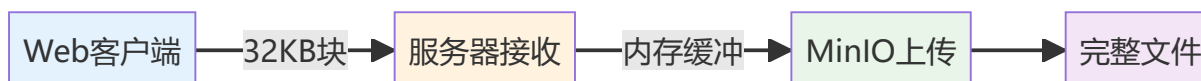
5.1 概述

本文档详细介绍 `minio_stream.cpp` 程序的实现原理, 该程序展示了如何模拟Web上传场景, 从文件按32KB块读取数据到内存, 然后从内存上传到MinIO, 最终形成完整文件。

5.1.1 核心特性

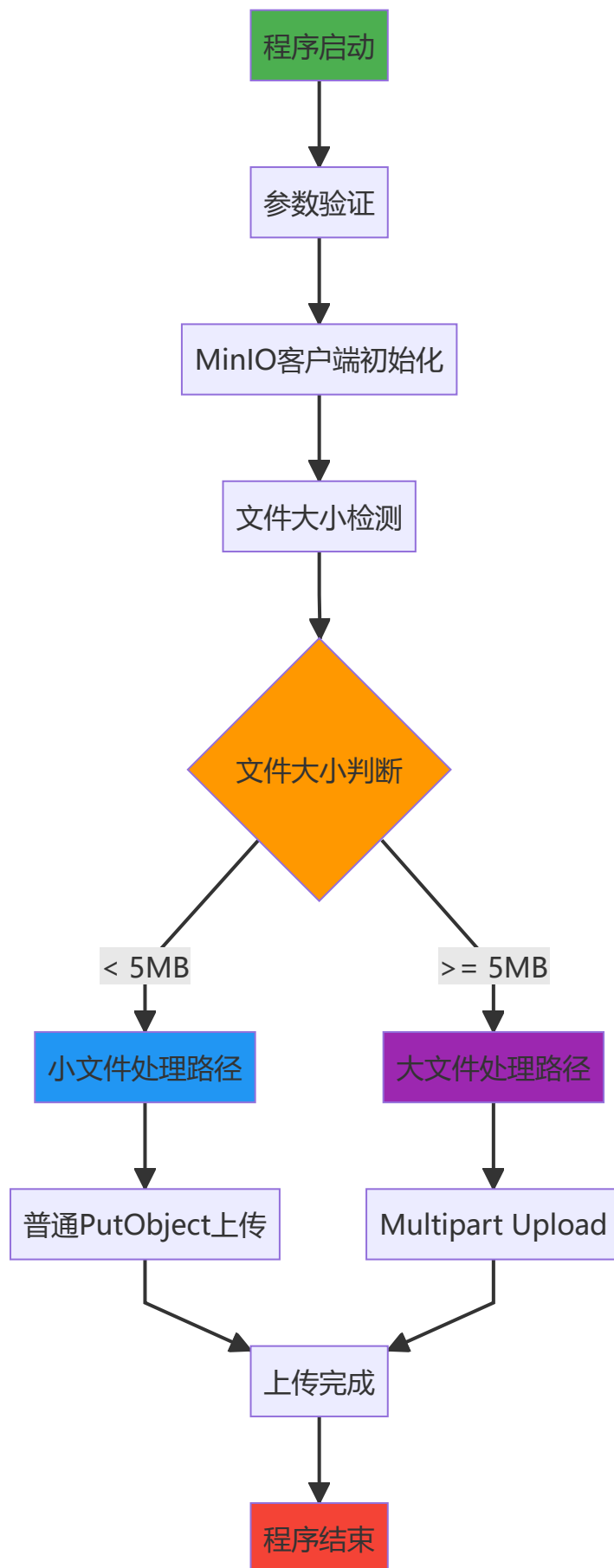
- **内存缓冲**: 模拟Web客户端分块上传行为
- **双路径处理**: 根据文件大小自动选择上传策略
- **完整文件**: 确保MinIO中存储的是完整文件而非分块文件
- **流式处理**: 边读取边处理, 减少内存占用峰值

5.1.2 应用场景

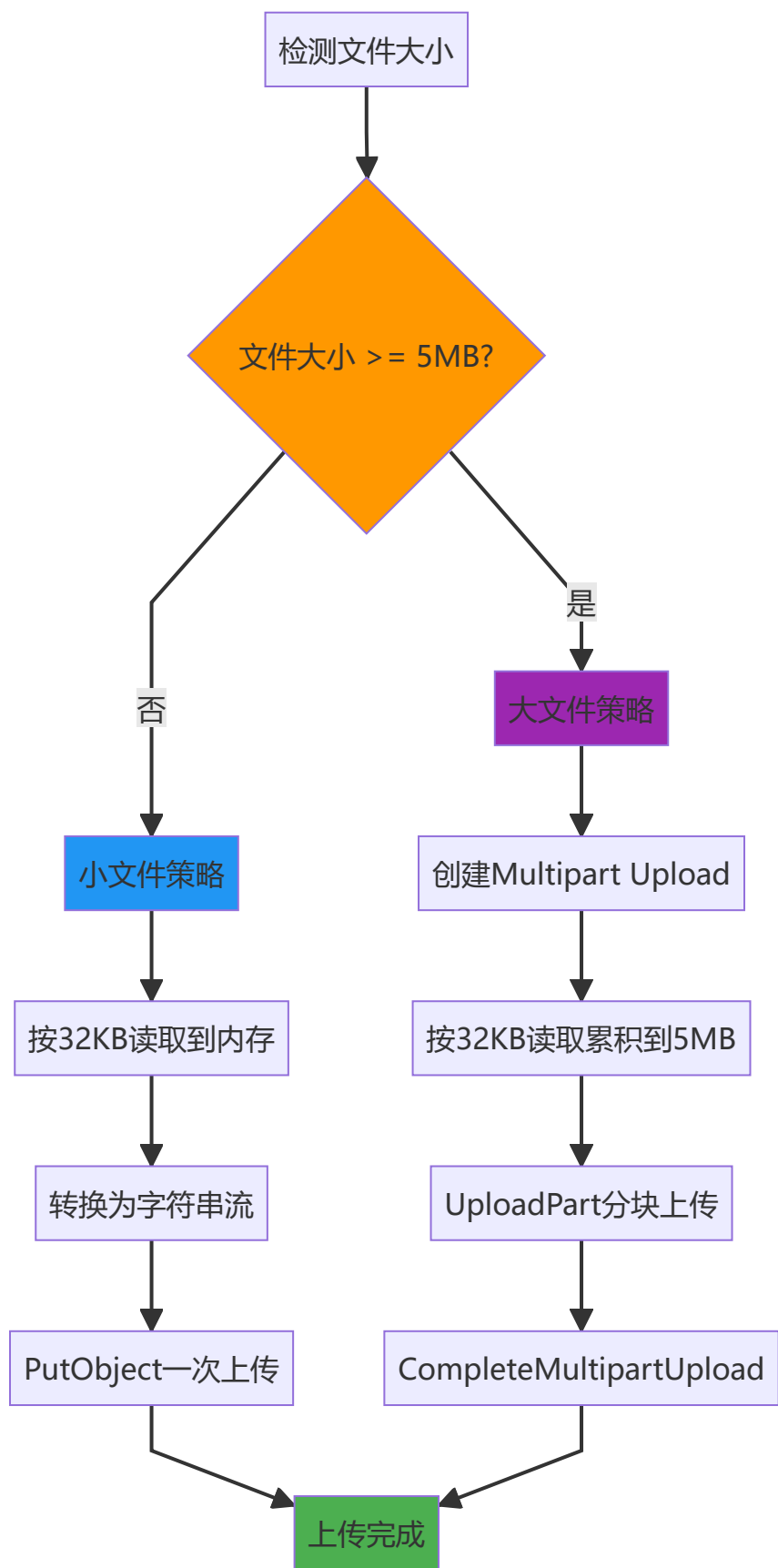


5.2 程序流程分析

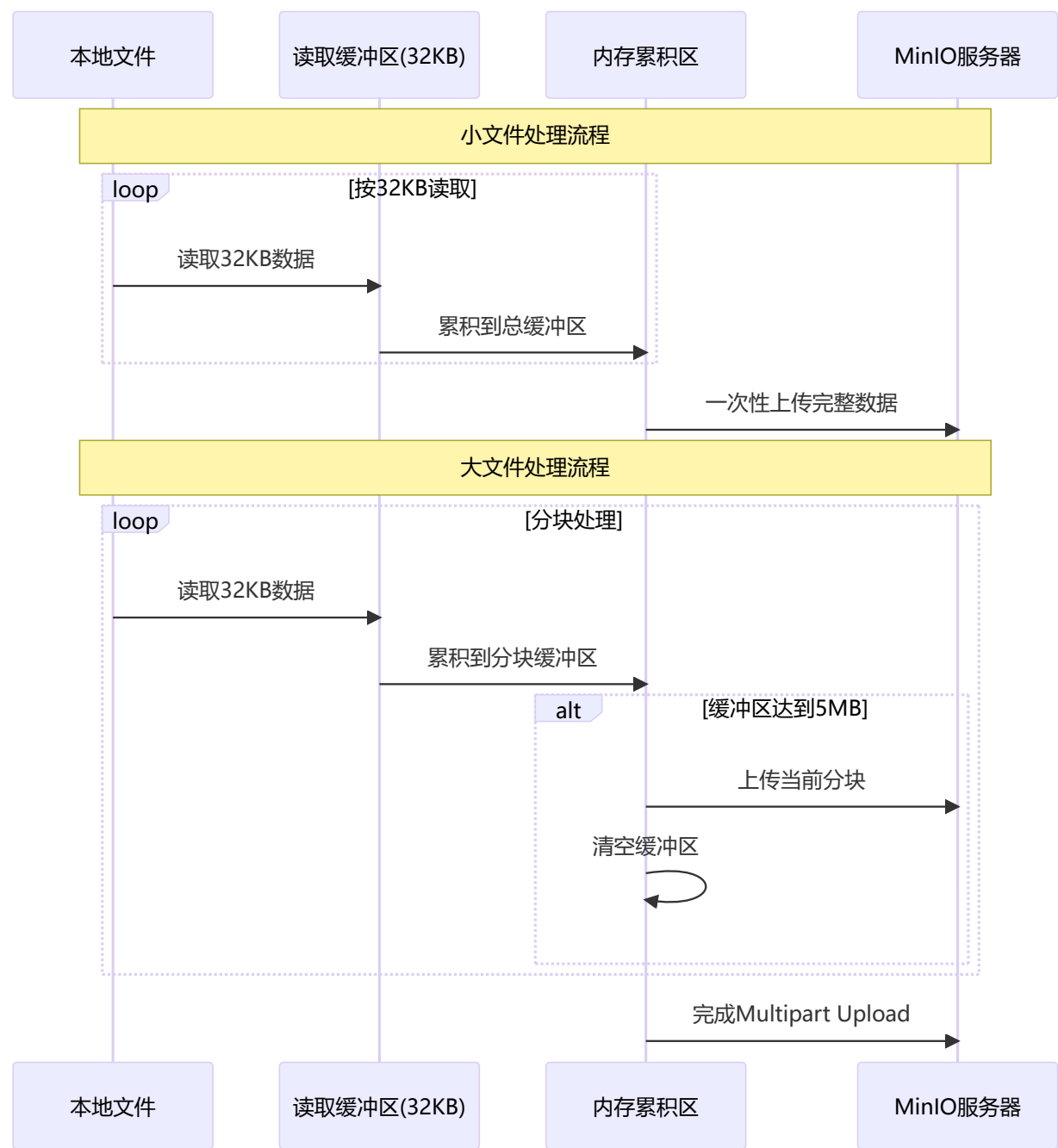
5.2.1 整体架构图



5.2.2 文件大小决策流程



5.2.3 内存管理流程



5.3. 代码详解

5.3.1 程序初始化

5.3.1.1 参数验证和配置

```
int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "使用方法: " << argv[0] << " <source_file>" << std::endl;
        return 1;
    }
    std::string sourceFile = argv[1];
}
```

说明: 程序要求用户提供源文件路径作为命令行参数。

5.3.1.2 MinIO客户端初始化

```
// MinIO服务器连接配置
std::string minioEndpoint = "localhost:9000";
std::string accessKey = "minioadmin";
std::string secretKey = "minioadmin";
bool useSSL = false;

// 创建MinIO客户端
minio::s3::BaseUrl baseUrl(minioEndpoint, useSSL);
minio::creds::StaticProvider provider(accessKey, secretKey);
minio::s3::Client minio(baseUrl, &provider);
```

组件解析:

- `BaseUrl`: 封装服务器地址和协议
- `StaticProvider`: 提供静态认证凭据
- `Client`: MinIO操作的核心接口

5.3.1.3 关键常量定义

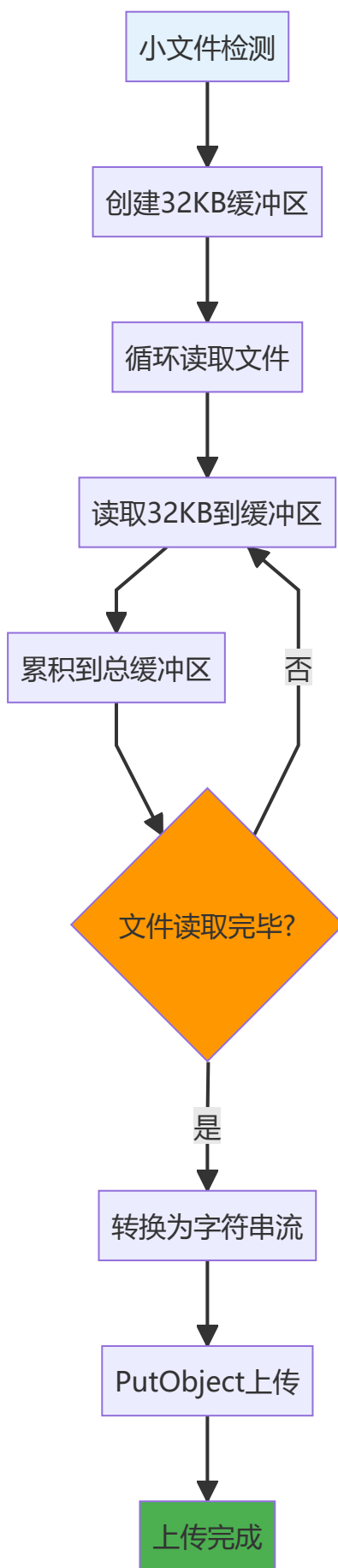
```
const size_t CHUNK_SIZE = 32 * 1024; // 32KB - 模拟web分块大小
const size_t MIN_PART_SIZE = 5 * 1024 * 1024; // 5MB - MinIO最小分块要求
```

设计考虑:

- `CHUNK_SIZE`: 模拟Web客户端常用的分块大小
- `MIN_PART_SIZE`: 满足MinIO Multipart Upload的最小分块要求

5.3.2 小文件处理路径 (< 5MB)

5.3.2.1 流程图



5.3.2.2 核心代码分析

```
// 按32KB块读取文件到内存
std::vector<char> allData;           // 总数据缓冲区
std::vector<char> buffer(CHUNK_SIZE); // 32KB读取缓冲区
size_t totalRead = 0;

while (file.read(buffer.data(), CHUNK_SIZE) || file.gcount() > 0) {
    size_t bytesRead = file.gcount(); // 实际读取字节数
    totalRead += bytesRead;

    std::cout << "从文件读取到内存: " << bytesRead << " 字节 (总计: "
                << totalRead << "/" << totalSize << ")" << std::endl;

    // 将数据添加到总缓冲区
    allData.insert(allData.end(), buffer.begin(), buffer.begin() + bytesRead);
}
```

关键点:

- `file.gcount()`: 获取实际读取的字节数
- `allData.insert()`: 将读取的数据追加到总缓冲区
- 实时显示读取进度

5.3.2.3 内存到流的转换

```
// 从内存创建字符串流并上传
std::string dataStr(allData.begin(), allData.end());
std::stringstream dataStream(dataStr);

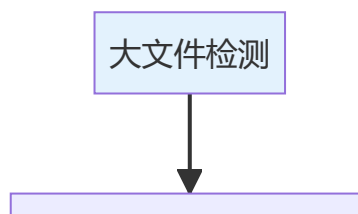
minio::s3::PutObjectArgs args(dataStream, allData.size(), 0);
args.bucket = bucketName;
args.object = objectName;
```

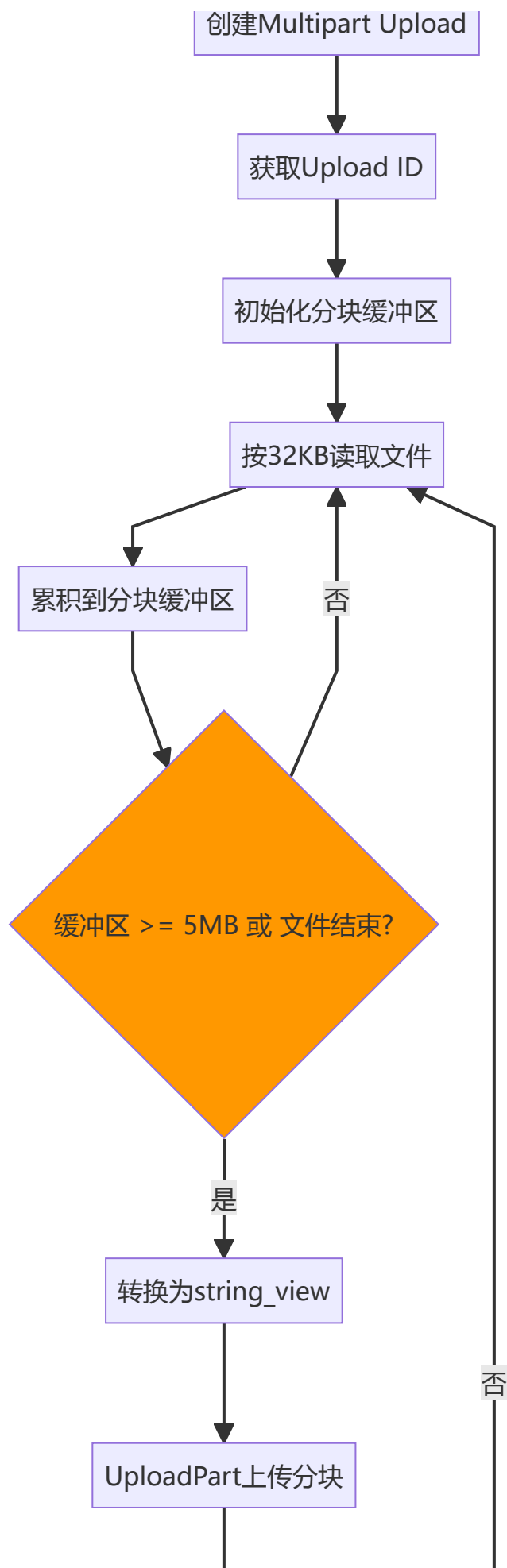
技术细节:

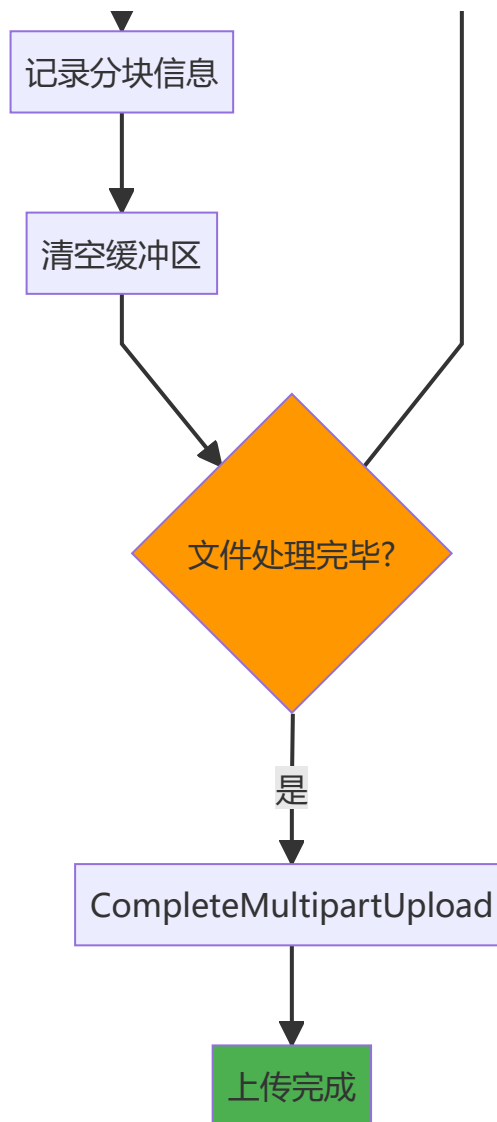
- 将 `vector<char>` 转换为 `string`
- 再创建 `stringstream` 供MinIO SDK使用
- 第三个参数 0 表示让SDK自动处理分块大小

5.3.3 大文件处理路径 (>= 5MB)

5.3.3.1 Multipart Upload流程







5.3.3.2 创建Multipart Upload

```
// 步骤1: 创建Multipart Upload
minio::s3::CreateMultipartUploadArgs createArgs;
createArgs.bucket = bucketName;
createArgs.object = objectName;

minio::s3::CreateMultipartUploadResponse createResp =
minio.CreateMultipartUpload(createArgs);
if (!createResp) {
    std::cerr << "创建Multipart Upload失败: " << createResp.Error().String() <<
std::endl;
    return 1;
}

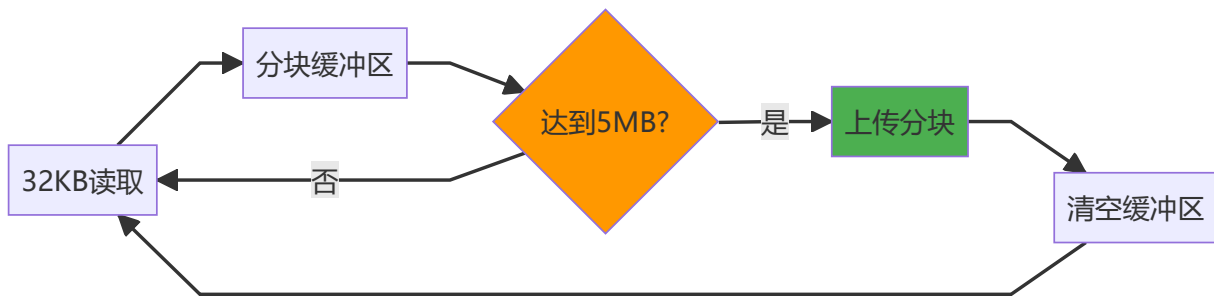
std::string uploadId = createResp.upload_id;
```

关键概念:

- `upload_id`: 用于标识整个Multipart Upload会话

- 后续所有分块上传都需要引用这个ID

5.3.3.3 分块数据累积和上传



```

std::list<minio::s3::Part> parts;           // 已完成分块列表
std::vector<char> partBuffer;               // 分块累积缓冲区
std::vector<char> readBuffer(CHUNK_SIZE); // 32KB读取缓冲区
int partNumber = 1;

while (file.read(readBuffer.data(), CHUNK_SIZE) || file.gcount() > 0) {
    size_t bytesRead = file.gcount();
    totalRead += bytesRead;

    // 将数据添加到分块缓冲区
    partBuffer.insert(partBuffer.end(), readBuffer.begin(), readBuffer.begin() +
bytesRead);

    // 当缓冲区达到5MB或文件读取完毕时，上传分块
    bool isLastPart = (totalRead >= totalSize);
    if (partBuffer.size() >= MIN_PART_SIZE || isLastPart) {
        // 执行分块上传逻辑
    }
}

```

缓冲区管理:

- `partBuffer`: 累积数据直到达到5MB
- `readBuffer`: 固定32KB的读取缓冲区
- `isLastPart`: 处理最后一个可能不足5MB的分块

3.3.4 分块上传实现

```
// 将数据从vector转换为string, 然后创建string_view
std::string partDataStr(partBuffer.begin(), partBuffer.end());
std::string_view partData(partDataStr);

minio::s3::UploadPartArgs uploadPartArgs;
uploadPartArgs.bucket = bucketName;
uploadPartArgs.object = objectName;
uploadPartArgs.upload_id = uploadId;
uploadPartArgs.part_number = partNumber;
uploadPartArgs.data = partData;

// 立即上传, 确保partDataStr在作用域内
minio::s3::UploadPartResponse uploadPartResp = minio.UploadPart(uploadPartArgs);
```

技术要点:

- `string_view`: MinIO SDK要求的数据格式
- 生命周期管理: 确保 `partDataStr` 在上传期间有效
- `part_number`: 分块编号, 从1开始

3.3.5 分块信息记录

```
// 记录完成的分块
minio::s3::Part part;
part.number = partNumber;
part.etag = uploadPartResp.etag;
parts.push_back(part);

// 清空缓冲区, 准备下一个分块
partBuffer.clear();
partNumber++;
```

数据结构:

- `Part.number`: 分块编号
- `Part.etag`: 分块的ETag标识
- `parts`: 使用 `std::list` 存储所有完成的分块

3.3.6 完成Multipart Upload

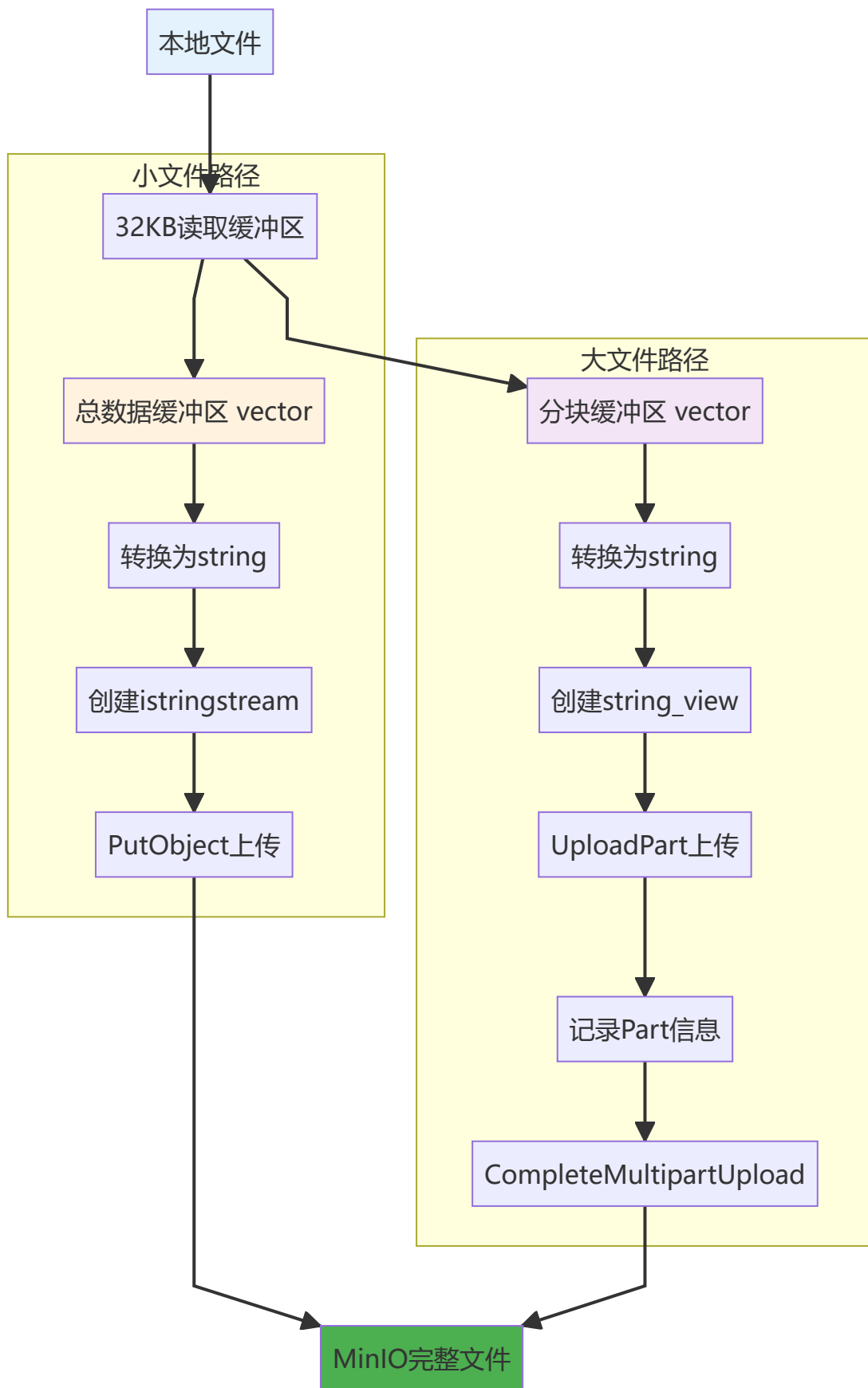
```
// 步骤3: 完成Multipart Upload
minio::s3::CompleteMultipartUploadArgs completeArgs;
completeArgs.bucket = bucketName;
completeArgs.object = objectName;
completeArgs.upload_id = uploadId;
completeArgs.parts = parts;

minio::s3::CompleteMultipartUploadResponse completeResp =
minio.CompleteMultipartUpload(completeArgs);
```

合并过程:

- MinIO服务器根据 `parts` 列表按顺序合并所有分块
- 最终形成一个完整的对象文件

5.3.4 数据流转换图



5.4 小结

5.4.1 技术点

1. **智能路径选择**: 根据文件大小自动选择最优上传策略
2. **内存友好**: 大文件处理时内存占用恒定
3. **Web场景模拟**: 真实模拟Web分块上传行为
4. **完整性保证**: 确保MinIO中存储完整文件

5.4.2 适用场景

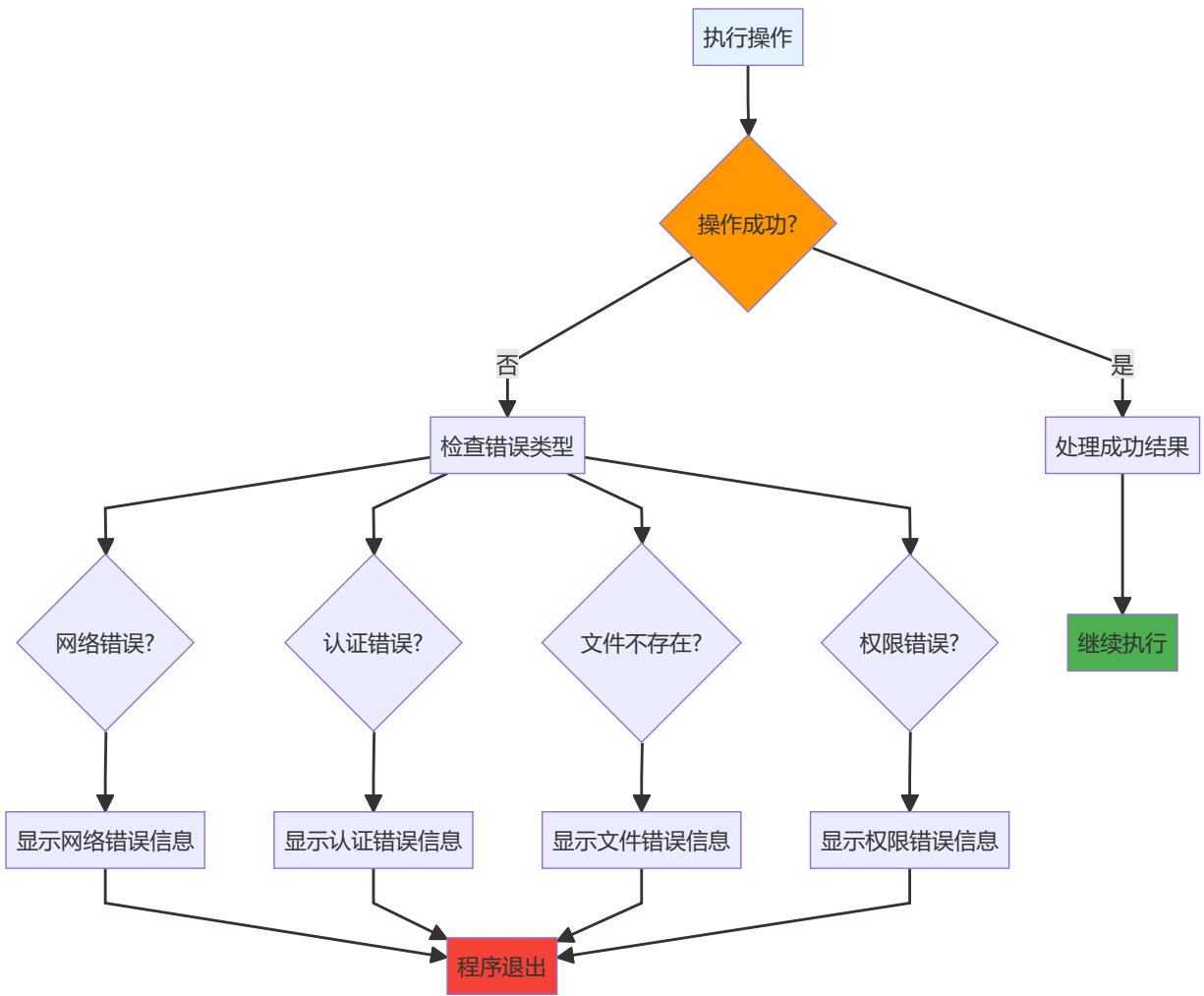
- **Web文件上传服务**: 后端接收分块数据并上传到对象存储
- **大文件处理**: 处理超大文件而不占用过多内存
- **流式数据处理**: 边接收边处理的数据管道
- **云存储代理**: 作为本地文件系统和云存储的桥梁

5.4.3 扩展建议

1. **断点续传**: 基于Multipart Upload实现断点续传
2. **进度回调**: 添加上传进度监控

6 错误处理机制

6.1 错误处理流程

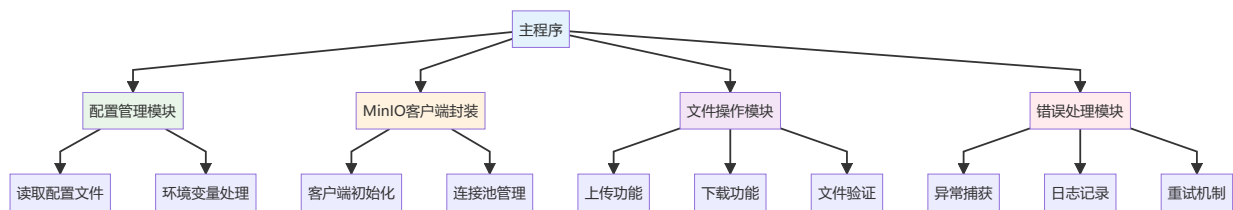


6.2 常见错误及解决方案

错误类型	可能原因	解决方案
连接失败	MinIO服务器未启动	启动MinIO服务器
认证失败	密钥错误	检查accessKey和secretKey
存储桶不存在	桶名错误或未创建	创建对应的存储桶
文件不存在	本地文件路径错误	检查文件路径
权限拒绝	用户权限不足	检查用户权限设置

7 最佳实践

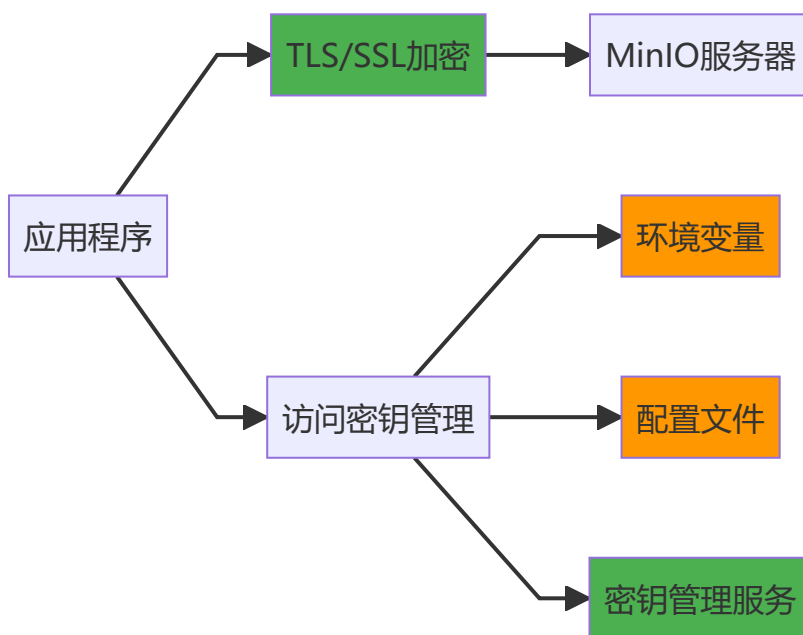
7.1 代码结构建议



7.2 性能优化建议

1. **连接复用**: 避免频繁创建客户端实例
2. **分块上传**: 对于大文件使用分块上传
3. **并发控制**: 合理控制并发上传/下载数量
4. **错误重试**: 实现指数退避重试机制
5. **资源清理**: 及时关闭文件流和网络连接

7.3 安全考虑



安全建议:

- 使用HTTPS连接
- 密钥不要硬编码在源代码中
- 定期轮换访问密钥
- 实施最小权限原则
- 启用访问日志和审计