XDELTALIB 完全参考手册

[一、关于我 3](#_Toc2670056)

[二、关于 XDELTALIB 3](#_Toc2670057)

[2.1 产生原因 3](#_Toc2670058)

[2.2 XDELTALIB 可以做什么 4](#_Toc2670059)

[2.2.1 数据同步 5](#_Toc2670060)

[2.2.2 文件多版本 5](#_Toc2670061)

[2.2.3 文件相似度检查 5](#_Toc2670062)

[2.2.4 XDELTALIB 不能做什么 5](#_Toc2670063)

[2.3 XDELTALIB 的代码协议 6](#_Toc2670064)

[2.4 关于 XDELTALIB logo 6](#_Toc2670065)

[三、技术解析 6](#_Toc2670066)

[3.1 术语 7](#_Toc2670067)

[3.2 计算原理 7](#_Toc2670068)

[3.2.1 单轮计算 7](#_Toc2670069)

[3.2.2 多轮计算 8](#_Toc2670070)

[3.2.3 就地生成 10](#_Toc2670071)

[四、编程模式 12](#_Toc2670072)

[4.1 操作对象 12](#_Toc2670073)

[4.2 黑盒（C++）方式 12](#_Toc2670074)

[4.2.1 客户端 13](#_Toc2670075)

[4.2.2 单轮计算：服务器端（目标文件端） 14](#_Toc2670076)

[4.2.3 多轮计算：服务器端（目标文件端） 14](#_Toc2670077)

[4.2.4 就地计算：服务器端（目标文件端） 15](#_Toc2670078)

[4.3 C API 模式 16](#_Toc2670079)

[4.3.1 数据结构 17](#_Toc2670080)

[4.3.2 接口声明 18](#_Toc2670081)

[4.3.3 单轮计算 19](#_Toc2670082)

[4.3.4 多轮计算 19](#_Toc2670083)

[4.3.5 就地生成 21](#_Toc2670084)

[4.3.6 混合模式 22](#_Toc2670085)

[4.3.7 回调模式 22](#_Toc2670086)

[五、未来的变化 25](#_Toc2670087)

[六、参考文档 25](#_Toc2670088)

[七、感谢 25](#_Toc2670089)

# 一、关于我

没有。

# 二、关于 XDELTALIB

## 2.1 产生原因

RSYNC 工具所使用的同步算法，简单而高效，并且在各种平台上都有很好的应用。但是由于自己处理数据工作的需要，想要将其中的算法集成到自己的项目中。虽然 RYSNC 的算法是公开的，但是自己再重写一遍有违程序员“懒”的美名。但是在网上找了一些相应的库，却没有发现特别合适。要么是实现的库没有比较好的接口化的实现，要么是夹杂在其他的项目中，没有办法直接使用。所以在重读了一下 RSYNC 算法后花了一个礼拜的时候写出了算法的核心。后面的开发都是在做外围支持。

虽然 RSYNC 的同步算法已经发明很久，而且当前的计算及网络带宽相对当时的情况来说实在是好太多了，RSYNC 算法是否还那么有价值么？这个问题其实不用去问题，因为硬件虽然进步了好很多，但是所要处理的数据也是海量地增长。同时，为了安全的需要数据是存在大量的冗余的。而且在云环境下，数据生产与存储根本针法控制，即规模巨大，也冗余度非常高。虽然现在硬件价格相对来说也不那么贵，但是能省个 10%，在一个亿里，也是一千万啊。也别把别人的钱不当钱花。其实，数据少点，节省的那可都是时间。就算不是为了节省存储空间，以目前的数据量来看，异地的云平台上同步数据，想想都有大把的应用空间。还有最近兴起的云盘等应用，经常在本地与云服务器上进行数据同步，如果可以，当然还是同步的数据越少越好。我在平时的工作中，就是不断在工作地、家里、外地不同的地方处理相同的文件。云盘给自己带来的好处，还是很大的。

总的来说，XDELTALIB是用 C++ 实现的差异数据提取库，其核心就是 RSYNC 的算法以及文件相似性计算算法。用于提取文件间差异数据，或者计算文件之间的相似性，并且可用于在两个端点之间进行差异化的文件同步，有一个压缩工具称为 xdelta，但是其所使用的算法原理跟本库完全不同。XDELTALIB库的特点有如下几个：

\* 用 C++ 写成，并提供 C++ 与 C 两种接口模式。

\* 支持多平台，在 Windows 与 Linux 中经过严格测试，也可以整合到 Unix 平台中。

\* 代码经过特别优化，差异算法及数据结构经过精心设计，增强了执行性能。

\* 支持 in-place 同步算法，可以应用到各种平台中，包括移动平台、服务器环境以及 PC 环境。

\* 支持可配置的 multi-round （多轮）同步算法，提高同步效率，同时提高了集成平台的可配置性。

\* 集成网络数据传输功能，减少了用户整合的工作量，加快整合进度。

\* 支持可配置的或者默认的线程数，充分利用多核优势，提高了执行性能。

\* 采用消费者与生产者模型提交与处理任务，充分利用并发优势。

\* 一库多用途，即可用于传统的文件数据同步，也可用于其他差异算法可应用的场景，如文件相似检查，以及大数据分析。

\* 良好的平台适应性。通过特别的设计，提供在各种存储平台的应用，如单设备环境，云存储环境，以及分存式存储环境。

\* 完备的文档。

## 2.2 XDELTALIB 可以做什么

虽然 RSYNC 算法由来已久，但是在云时代中，RSYNC 算法还有其巨大的优势。

### 2.2.1 数据同步

这是最经典的应用场景，其可用在如 RSYNC 工具所应用的场景中。特别是在现在这种云灾备，多点数据中心中，由于需要有大量的数据需要处理，而且相比来说，这些数据所具有的冗余度还是很高的。当然同步的形式不再只限于文件对文件的同步，如果平台够牛，可以进行平台与平台形式的同步。当然毫无疑问，这会消耗大量的计算资源与内存资源。但是这是另一个可以大力优化算法的地方。

### 2.2.2 文件多版本

为了能够恢复文件某个时刻的数据，现在大多平台都是在特定的时刻将文件重新复制一份。这当然方便，但是产生的数据冗余也是非常大的。相像一个文件只有1MB，存100个版本，则需要 100MB，想想看冗余度就高得不得了。特别是在一些云存储平台上。数据本身底层就已经有两个以上的复杂，所以这冗余度还得再乘上个冗余复本说。如果针对每次变化只存储变化的数据，能够节省下来的空间还是很可观的。

### 2.2.3 文件相似度检查

将计算出来的差异数据用下面的公式可以得到一个粗略的相似度的公式：

*公式1*

公式虽然很粗糙，但是却相对简单而实用。在进行云平台数据分析时，说不定有用。而且，有时越是简单直接的方法对大系统反而更有价值。

### 2.2.4 XDELTALIB 不能做什么

除了它能做的，其他的它都做不了。

## 2.3 XDELTALIB 的代码协议

我们采用 GPL v2.0 来发布我们的代码。你应该手头有一份这样的协议，如果没有，你可以写信到这里获取：Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA。

如果你需要在你的项目中集成本库，请确认你的项目符合 GPL 协议。如果你需要在商业产品中集成，请写信到：yeyouqun@163.com。

在本库中，大部份的代码是由作者编写，但是也有些代码是从公共的空间获取的，他们有自己的协议。作者尽量保持这些代码所使用的协议是与 GPL 协议兼容的（如数据摘要计算代码等），我们保留了原来代码的协议声明。他们中基本使用 BSD 2-Clause 协议方式授权代码，或者直接使用了 GPL 的授权方式。

## 2.4 关于 XDELTALIB logo

文件 xdelta.png 是 XDELTALIB 的专用图标，任何人不得在未经许可的情况下，使用在商业产品中，但是可以用于符合 GPL 协议的源代码或者产品，但是必须说明图标的来源。图标的所有人为项目的作者，即 yeyouqun@163.com。

图标如下：



图1： XDELTA 图标

# 三、技术解析

后面的说明集中于库的使用本身与相关的技术细节。

## 3.1 术语

为了后面讨论的方便，需要给一些常用的术语进行一下解释。

目标文件与源文件：如果有文件 A 与 B，需要将 A 差异同步到 B，或者要计算 A 与 B 的差异数据，我们这里称 B 为目标文件，A 为源文件。

在本库的 C++ 接口中，使用的同步模式为：

1. 如果要将某个源文件的内容同步到目的地，即在目的地上创建一个等待进程;

2. 源文件端将所要处理的任务（文件）依次提交到本地的处理线程中，多个线程会依次从提交的任务提取并处理完成。

3. 如果处理完成，则源文件端的进程退出，此时，目的地端的进程也会退出。

客户端：在上面的过程中，在源文件所在的系统中创建的同步进程为客户端，用来接收目标文件的哈希值，并对本地文件进行同步计算，然后将计算结果发送回去。

服务端：在上面的过程中，在目标文件所在的系统中创建的同步进程为服务端，计算目标文件的哈希值，等往同步结果，根据计算结果在目标系统中重建文件。

## 3.2 计算原理

详细的原理的分析，可以参考论文的原文件。里面有更详细的数据分析以及场景分析。这里的原理只是针对算法的概要进行总结，以方便进行全貌的理解。

### 3.2.1 单轮计算

单轮计算的原理如下：

图2：单轮计算示意图

在上面的图中，上部的矩形为目标文件，下部为源文件，红色虚线间隔则为分块，红色虚线框为计算窗口，红块表示差异数据的地方。其执行过程如下：

1. 按特定的固定长度计算目标文件的每个块的两个校验值，一个叫快速校验值，一个是慢速校验值，并传递给源文件。
2. 在源文件中按照相同的块大小，从当前位置（开始始为0）利用滚动哈希算法，[[1]](#footnote-1)计算快速校验值，并比较源文件中哪个块有相同的快速校验值，如果没有找到就到第3 步。找到了，再计算块的慢速校验值，如果相同，则表示块数据相同，则到第4 步。如果不相同，则跳到第3步。
3. 移动计算窗口一个字节，再重复 1，2 两个过程，直到整个源文件计算完毕。
4. 移动计算窗口一个块字节数，再重复 1，2 两个过程，直到整个源文件计算完毕。

更加详细的原理说明，请参考 *Efficient Algorithms for Sorting and Synchronization，Andrew Tridgell， A thesis submitted for the degree of Doctor of Philosophy at The Australian National University，February 1999。*

### 3.2.2 多轮计算

多轮计算的原理如下：

图3 ：多轮计算示意图：

在上面的图中，上部的矩形为目标文件，下部为源文件，红色虚线间隔则为分块，红块表示差异数据的地方，绿线表示更大的块分割区域。其执行过程如下：

首先，分别将目标文件与源文件整个文件作为一个洞，并对洞执行如下过程：

1. 按特定的固定长度计算目标文件的每个块的两个校验值，一个叫快速校验值，一个是慢速校验值，并传递给源文件。
2. 在源文件中按照相同的块大小，从当前位置（开始始为0）利用滚动哈希算法，[[2]](#footnote-2)计算快速校验值，并比较源文件中哪个块有相同的快速校验值，如果没有找到就到第3 步。找到了，再计算块的慢速校验值，如果相同，则表示块数据相同，则到第4 步。如果不相同，则跳到第3步。
3. 移动计算窗口一个字节，再重复 1，2 两个过程，直到整个源文件计算完毕。
4. 移动计算窗口一个块字节数，再重复 1，2 两个过程，直到整个源文件计算完毕。

当上面的第一轮计算完毕后，再将计算得到的相同的块的区域从两个文件洞中移除。然后再将目标文件剩下的洞区域以更小的块大小进行哈希计算，即重复步骤1到步骤4的过程。直到最达到最小的块大小。

示意图如下：

在第一次计算时，文件的数据假设如下：

图4：多轮计算第一轮

为简单起见，只假设针对第一轮块大小只有一个相同的块。当把相同块部分的区域移除后，如形成下面的的区域：

图5：多轮计算下一轮

这里需要将“洞3”与“同4”再用更小的块计算哈希，并供传输给源文件，同时对“洞1”与“洞2”进行差异计算，并将计算结果回传。

这个过程一直持续到最小块大小计算完成。

多轮计算是一个时间换空间的算法。如果文件差异很大，有可能导致性能下降非常严重。所以在使用多轮计算时，需要根据实际情况动态调整。

更加详细的原理说明，请参考 *J. Langford. Multiround Rsync. Technical Report Available at www.cs.cmu.edu/jcl/research/mrsync/-mrsync.ps, Dept. of Computer Science, Carnegie-Mellon University, 2001.*

### 3.2.3 就地生成

由于在同步重构文件时，需要新生成一个新文件，并将同步结果写入到这个临时的新文件中。当同步完成时，则将旧的文件删除，再将这个临时文件改名为原来的文件名。这个过程需要一些额外的空间，对于一些存储空间比较紧张的设备，则可以采用就地生成的文件重构模式。即不用生成新文件，而直接在原文件上进行数据移动或者写入操作来实现文件的重构。

对于就地生成文件，可能会导致同步结果的退化。但是根据测试结果，如果文件的内容基本是顺序移动进行变化，比如插入数据，删除数据比较多，这样用就地生成的方式不会产生严重的退化。如果文件内容会前后交换，则这可能比较容易导致同步的退化。

其生成原理如下：

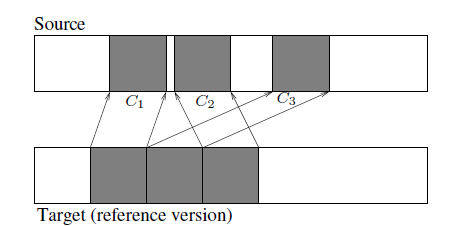


图6：就地生成原理图

1. 为了通过 Target 同步成源文件，则需要将 C1 对应的块向后移动，这个时候会导致覆盖目标文件中后面一个相同的块的数据。
2. 为了防止破坏 1 所说的相同的块数据，则需要在处理 C1 之前处理 C3 对应的块，依此类推，并将 C3 推入处理顺序队列的末尾，直到两个条件结束：
   1. 块不会再覆盖其他的块（数据相同的块）。
   2. 最后递归回到了自己，则本块被标志为不同块。
3. 再继续处理下一块，如果递归到已经处理的块，则返回。直到最后一块处理完成。
4. 将处理块按顺序从源文件中读取然后复制到块所指定的位置。

要注意的地方是，就地生成对于数据的执行顺序是很严格的。如果你想通过对内容的位置进行排序后再进行写入，以期得到顺序操作，进而增强执行效率，但极可能导致数据的错误，因此结果是错误的。

更加详细的原理说明，请参考 *In-Place Rsync: File Synchronization for Mobile and Wireless Devices，David Rasch and Randal Burns Department of Computer Science Johns Hopkins University {frasch,randalg}@cs.jhu.edu。*

# 四、编程模式

使用 XDELTALIB 可以有两种模式。

一种是直接使用 XDELTALIB 的 C++ 模式。这种模式使用者能够做的事情比较少，所有的通信与计算等都已经固定了。使用者只需要写一些自己的观察器以及相应的文件处理类型即可。如果要改变数据的使用等，需要自己重载相应的类，使用难度会较大（这让我再一次理解到要用 C++ 写一个库是非常难的，即在于用，也在于实现者的超强经验与理解力。而相比之下 C 语言的简单为 C 的生命力提供了很好的支撑。），即要非常理解了 C++ 的类实现，才能比较好地使用这种方式。

另一种方式是直接使用 C 接口。通过这种方式使用者只需要得到计算结果，而剩下的事情则由使用者自己处理。

## 4.1 操作对象

XDELTALIB 是以文件的视角来编写库的代码的。对于整个系统来说，只要是数据是以一连续的逻辑视图出现，其逻辑像一个文件一样的只是字节流的序列，那么就可以使用 XDELTALIB 来进行相应的差异计算。比如，可以用来进行卷级的同步（我没有亲自测试过这样的同步是否会带来什么样的效率，只是做一个例子来说明）。实际上，在很多的平台上，无论是对象存储还是其他的存储，最终实际都是表现为连续的字节流（即文件）。

## 4.2 黑盒（C++）方式

C++ 的方式，已经将传输压缩功能也网络传输功能集成到了实代码中，应用没有多少选择的余地。应用只需要写一些观察器，并且代码整合到系统中即可完成。意即，使用这样的接口，多限于非常有限的应用范围（只针对文件同步）。并且系统简单，开发过程很短的项目中。

如果项目需要有自己的节奏，或者要适配自己的应用场景，建议使用 C 接口的方式。

### 4.2.1 客户端

由于同步的执行方式（单轮或者多轮，或者是就地生成）是由服务器来决定的，客户端的职责在于添加要执行同步的文件任务。所以，由于客户端的代码都是一样的：

* 第一步，生成自己的观察器：

|  |
| --- |
| class source\_observer : public xdelta::xdelta\_observer  {  }; |

* 第二步，启动任务：

|  |
| --- |
| xdelta::f\_local\_creator localop (path); // 与平台相同的文件操作器，库提供了本地的操作器。  xdelta::file\_operator &fop = localop; // 如果你有不同的操作器，请从基类派生自己的文件操作器。  // 如在云平台中的文件操作器，与本地肯定是不同的。  client.run (fop, ob, (xdelta::uchar\_t\*)argc[3]); // 执行  traverse\_source (path, "", client);//遍历文件 |

而在遍历文件的时候，要将需要同步的文件加入到任务队列中：

|  |
| --- |
| client.add\_task (new xdelta::f\_local\_freader (pathname, fname), &mydeletor); |

* 第三步，等待返回：

|  |
| --- |
| client.wait (); // 任务执行完成后退出这个接口。 |

完整的例子，请参考 test/testclient.cpp 文件 。在客户端中，执行的过程中会根据系统中的 CPU 的核数生成多个线程，每个线程从同一步队列中取出文件，并与服务器的对应线程连接，并执行同步操作。所没有文件时，所有线程退出。

### 4.2.2 单轮计算：服务器端（目标文件端）

其分为两端，如果是本地同步，则只需要将目标地址换成本地 IP 地址即可。

* 第一步，写一个自己的观察器：

|  |
| --- |
| class my\_observer : public xdelta::xdelta\_observer  {  ......  } |

* 第二步，生成执行对象：

|  |
| --- |
| my\_observer mo;  f\_local\_creator localop (path); // 与平台相同的文件操作器，库提供了本地的操作器。  file\_operator &fop = localop;// 如果你有不同的操作器，请从基类派生自己的文件操作器。  // 如在云平台中的文件操作器，与本地肯定是不同的。  xdelta\_server server (compress);  // compress 表示传输是否经过压缩，压缩后会提高传输效率  xdelta::uint64\_t start = time (0);  server.run (fop, mo); // 开始运行，直到完成后才返回这个接口。  xdelta::uint64\_t end = time (0); |

* 第三步，等待结束：

针对自己的观察器收到的数据进行统计分析。

在库的实现内部，会根据当前系统的CPU 核数，动态地生成多个执行线程，每个线程处理一个文件。而在客户端，也会有相应的线程与服务线程对应。这些线程对之间通过 TCP 连接执行通信。通过这种方式来提高系统的执行效率。因为大多数情况下，计算会消耗多数时间资源。

更详细的例子代码，请对待 test/testserver.cpp 中的 test\_single\_round 方法。

### 4.2.3 多轮计算：服务器端（目标文件端）

其分为两端，如果是本地同步，则只需要将目标地址换成本地 IP 地址即可。

* 第一步，写一个自己的观察器：

|  |
| --- |
| class my\_observer : public xdelta::xdelta\_observer  {  ......  } |

* 第二步，生成执行对象：

|  |
| --- |
| my\_observer mo;  f\_local\_creator localop (path); // 与平台相同的文件操作器，库提供了本地的操作器。  file\_operator &fop = localop;// 如果你有不同的操作器，请从基类派生自己的文件操作器。  // 如在云平台中的文件操作器，与本地肯定是不同的。  xdelta\_server multi (compress);  // compress 表示传输是否经过压缩，压缩后会提高传输效率  multi.set\_multiround\_size (5 \* 1024 \* 1024);  // 设置启用多轮计算的文件大小。小于这个则不使用多轮计算。  xdelta::uint64\_t start = time (0);  server.run (fop, mo); // 开始运行，直到完成后才返回这个接口。  xdelta::uint64\_t end = time (0); |

* 第三步，等待结束：

针对自己的观察器收到的数据进行统计分析。

在库的实现内部，会根据当前系统的CPU 核数，动态地生成多个执行线程，每个线程处理一个文件。而在客户端，也会有相应的线程与服务线程对应。这些线程对之间通过 TCP 连接执行通信。通过这种方式来提高系统的执行效率。因为大多数情况下，计算会消耗多数时间资源。

更详细的例子代码，请对待 test/testserver.cpp 中的 test\_multiround 方法。

### 4.2.4 就地计算：服务器端（目标文件端）

其分为两端，如果是本地同步，则只需要将目标地址换成本地 IP 地址即可。

* 第一步，写一个自己的观察器：

|  |
| --- |
| class my\_observer : public xdelta::xdelta\_observer  {  ......  } |

* 第二步，生成执行对象：

|  |
| --- |
| my\_observer mo;  f\_local\_creator localop (path); // 与平台相同的文件操作器，库提供了本地的操作器。  file\_operator &fop = localop;// 如果你有不同的操作器，请从基类派生自己的文件操作器。  // 如在云平台中的文件操作器，与本地肯定是不同的。  xdelta\_server server (compress);  // compress 表示传输是否经过压缩，压缩后会提高传输效率  server.set\_inplace (); // 设置采用 inplace 来构造新文件。  xdelta::uint64\_t start = time (0);  server.run (fop, mo); // 开始运行，直到完成后才返回这个接口。  xdelta::uint64\_t end = time (0); |

* 第三步，等待结束：

针对自己的观察器收到的数据进行统计分析。

在库的实现内部，会根据当前系统的CPU 核数，动态地生成多个执行线程，每个线程处理一个文件。而在客户端，也会有相应的线程与服务线程对应。这些线程对之间通过 TCP 连接执行通信。通过这种方式来提高系统的执行效率。因为大多数情况下，计算会消耗多数时间资源。

更详细的例子代码，请对待 test/testserver.cpp 中的test\_inplace\_with\_single\_round 方法。

## 4.3 C API 模式

通常情况下，采用 C API 的模式来进行集成，具有最大的灵活性。XDELTALIB 只负责差异计算及输出。诸如数据传递，压缩等等附加功能，则是由具体的产品来做的。

其基本的操作步骤是：

1. 计算目标数据的哈希结果。

2. 将哈希结果输入到源文件中，对源文件进行差异计算。

3. 输出差异结果。

4. 根据差异结果，执行数据的迁移，或者从源文件中，或者从目标文件中读取数据，再写到重建的文件中（就地生成中，重新文件就是其自己）。

在这个过程中，由于在计算差异数据时，会读取一次源文件，而在差异计算结束时，可能会再次读取源文件中的某部分数据。如果在计算差异数据时就将差异数据输出，不是对执行效率有所提高么？为什么 XDELTALIB 没有采用这样的模式呢。原因如下：

1. 一般要实现这样的操作，需要通过回调的方式来实现。但是在回调时，就地生成方式有可能导致数据被破坏。如果只针对就地生成方式不使用回调方式，在接口上没有办法去实现隔离。这样调用都很容易就误用这个接口，而导致错误的结果。

2. 从多轮测试的结果来看，通过即时回调写差异数据的情况下，会导致差异数据的块数增加很多。而差异数据在最后一轮进行处理，效率会更高。

因此，综合上面的两个理由，XDELTALIB 没有采用即时回调输出差异数据的模式。

下面先说明一下相关的数据结构（这些数据结果有可能会有变化 ，我努力保持变化的同步更新）：

### 4.3.1 数据结构

下面是数据结构及其声明：

|  |
| --- |
| typedef struct file\_hole {  unsigned long long pos; // 洞在文件中的绝对位置。  unsigned long long len; // 洞的长度。  struct file\_hole \* next; // 相邻的下一个洞。  }fh\_t; |

上面是表示文件洞的结构，如果在单轮计算中，这个结构所表示的就是整个文件。

|  |
| --- |
| typedef struct hash\_item {  unsigned fast\_hash; // 快哈希值  unsigned char slow\_hash[DIGEST\_BYTES]; // 16 Bytes  unsigned long long t\_offset; //目标文件的基于0开始的偏移地址;  unsigned t\_index;  // 块索引, 基于 t\_offset。之所以采用这么复杂的结构，是为了与多轮计算使用相同的代码。  struct hash\_item \* next;  }hit\_t; |

上面的结构表示的是从目标文件中，按照指定块大小的计算得到的哈希描述结构体。

|  |
| --- |
| typedef struct xdelta\_item {  unsigned short type;  unsigned long long s\_offset;  // item 所指示的块在源文件中的位置，这个数据指示数据写入文件的位置。  unsigned long long t\_offset;  // type == DT\_DIFF，该值无意义，否则为 DT\_IDENT 时，指的是目标文件的起始位置。  // 可以从该处读取 blklen 长度的数据来生成同步的临时文件。  unsigned index;  // 相对于 t\_offset 的索引，索引大小为 blklen；  unsigned blklen; // 数据块的长度。  struct xdelta\_item \* next;  }xit\_t; |

上面的结构表示的是从源文件中，通过计算得到的差异数据的描述结构体。

更多的信息请参考头文件 capi.h。

### 4.3.2 接口声明

更多的信息请参考头文件 capi.h。

### 4.3.3 单轮计算

|  |
| --- |
| // initialize source file hole and target file hole as (0, source filesize) and (0, target filesize)  srchole = (0, source filesize);  tgthole = (0, target filesize);  blklen = origin\_len;  tmpfile = create\_a\_temp\_file ();  ***void \*inner\_data = start\_hash (blklen);***  ***pipehandle = xdelta\_run\_hash (tgthole, inner\_data);***  write\_file\_data\_in\_this\_hole\_to\_pipe (pipehandle);  ***hash\_result = xdelta\_get\_hashes\_free\_inner (inner\_data);***  ***inner\_data = xdelta\_start\_xdelta (hash\_result, blklen);***  ***xdelta\_free\_hashes (hash\_result);***  ***pipehandle = xdelta\_run\_xdeltas (srchole, inner\_data);***  write\_file\_data\_in\_this\_hole\_to\_pipe (pipehandle);  ***xdelta\_result = xdelta\_get\_xdeltas\_free\_inner (inner\_data);***  for (item in xdelta\_result) {  if (item.type == DT\_IDENT) {  data = copy\_from (tgtfile);  seek (tmpfile, block.s\_offset);  write (tmpfile, data, block.len);  }  else {  data = copy\_from (srcfile);  seek (tmpfile, block.s\_offset);  write (tmpfile, data, block.len);  }  }  ***xdelta\_free\_xdeltas (xdelta\_result);*** |

如果要查看详细的范例，请参考 test/testcapi.cpp 中的方法 test\_single\_round。注意上面的伪代码中红色的部分是 C api 接口。

### 4.3.4 多轮计算

|  |
| --- |
| // initialize source file hole and target file hole as (0, source filesize) and (0, target filesize)  srchole = (0, source filesize);  tgthole = (0, target filesize);  blklen = origin\_len;  tmpfile = create\_a\_temp\_file ();  for (blklen >= minimal len) {  ***void \*inner\_data = start\_hash (blklen);***  for (each hole in tgthole) {  ***pipehandle = xdelta\_run\_hash (hole, inner\_data);***  write\_file\_data\_in\_this\_hole\_to\_pipe (pipehandle);  }  ***hash\_result = xdelta\_get\_hashes\_free\_inner (inner\_data);***  ***inner\_data = xdelta\_start\_xdelta (hash\_result, blklen);***  ***xdelta\_free\_hashes (hash\_result);***  for (each hole in srchole) {  ***pipehandle = xdelta\_run\_xdeltas (hole, inner\_data);***  write\_file\_data\_in\_this\_hole\_to\_pipe (pipehandle);  }  ***xdelta\_result = xdelta\_get\_xdeltas\_free\_inner (inner\_data);***  for (item in xdelta\_result) {  if (item.type == DT\_IDENT) {  data = copy\_from (tgtfile);  seek (tmpfile, block.s\_offset);  write (tmpfile, data, block.len);  ***xdelta\_divide\_hole (tgthole, item.t\_offset, item.blklen);***  ***xdelta\_divide\_hole (srchole, item.s\_offset, item.blklen);***  }  }  ***xdelta\_free\_xdeltas (xdelta\_result);***  blklen = get\_next\_block\_length ();  }  for (item in xdelta\_result) {  if (item.type == DT\_DIFF) {  data = copy\_from (srcfile);  seek (tmpfile, block.s\_offset);  write (tmpfile, data, block.len);  }  } |

如果要查看详细的范例，请参考 test/testcapi.cpp 中的方法 test\_multiple\_round。注意上面的伪代码中红色的部分是 C api 接口。

下面有几个需要注意的点：

1. get\_next\_block\_length ()：对于这个方法，一般的处理可简简化地将块大小除以2，但是这么处理是否是最优的块不是很确定，没有足够的数据来支持下一个块大小的选择。这个可能需要根据应用自己的特点来处理。
2. 对于多轮是否需要持续下去，直到最小块大小，这个有两种情况。

a、如果第一转计算后基本没有相同的同，或者只有很少的块是相同的，则可能文件的差异性很大。再减少块大小执行差异计算，有可能结果也不是很乐观。所在在这种情况下，反倒是直接跳出，执行一转即可。这样相当于执行一次完全的复制。因此再多轮计算下去，并不能很好地利用相似数据来减少传输量，反而会导致计算性能的消耗，影响系统的吞吐量。

b、如果第一次计算时就已经非常相似了，比如按照“公式1”所计算得到的相似度已经达 80 以上，则再一次执行多轮计算，也只是消耗计算资源，影响系统的吞吐量。所以如何来中断多轮计算的过程，则可以根据应用自己的系统特点来选择。这个是应用自己可以优化的地方。

1. 关于多线程。上面的 API 接口都是多线程安全的。

### 4.3.5 就地生成

|  |
| --- |
| // initialize source file hole and target file hole as (0, source filesize) and (0, target filesize)  srchole = (0, source filesize);  tgthole = (0, target filesize);  blklen = origin\_len;  ***void \*inner\_data = start\_hash (blklen);***  ***pipehandle = xdelta\_run\_hash (tgthole, inner\_data);***  write\_file\_data\_in\_this\_hole\_to\_pipe (pipehandle);  ***hash\_result = xdelta\_get\_hashes\_free\_inner (inner\_data);***  ***inner\_data = xdelta\_start\_xdelta (hash\_result, blklen);***  ***xdelta\_free\_hashes (hash\_result);***  ***pipehandle = xdelta\_run\_xdeltas (blklen, srchole, inner\_data);***  write\_file\_data\_in\_this\_hole\_to\_pipe (pipehandle);  ***xdelta\_result = xdelta\_get\_xdeltas\_free\_inner (inner\_data);***  ***xdelta\_resolve\_inplace (&xdelta\_result);***  for (item in xdelta\_result) {  if (item.type == DT\_IDENT) {  data = copy\_from (tgtfile);  seek (tgtfile, block.s\_offset);  write (tgtfile, data, block.len);  }  else {  data = copy\_from (srcfile);  seek (tgtfile, block.s\_offset);  write (tgtfile, data, block.len);  }  }  ***xdelta\_free\_xdeltas (xdelta\_result);***  set\_file\_size (srcfile.get\_file\_size ()); |

如果要查看详细的范例，请参考 test/testcapi.cpp 文件中的方法 test\_single\_round\_inplace。注意上面的伪代码中红色的部分是 C api 接口。

### 4.3.6 混合模式

综合看上面的三个代码，他们的代码相似度是非常高的。应用完全可以将这些计算写在一个方法中，并通过相应的参数来控制最终采用什么样的计算模式。

### 4.3.7 回调模式

采用本库的前几个方法，你都只能得到差异数据块的描述信息，而不能得到真正的相同数据与差异数据。你需要再次从源文件中读取差异数据来写到目标文件中。这会导致对差异数据的多次读取。无论哪种情况都会使得性能有一定损失。为了解决这个问题，因此库中设计了一种回调模式来协调这个问题。但是一有点是需要注意的，这种模式只应用于单轮计算中。具体原因下面说明。

差异数据回调函数原型：

|  |
| --- |
| /\*\*  \* 参数说明：  \* @data 差异数据的缓存地址。  \* @datalen 差异数据的长度。  \* @offset 差异数据在源文件中的位置。  \* @priv 回调函数内部使用的数据。  \*/  typedef void(\*diff\_func\_t)(char \* data, unsigned datalen  , unsigned long long offset, void \* priv); |

函数用于在分析差异数据时，通过回调的方式处理差异数据。你在使用这个函数之前，请你务必仔细阅读下面的说明。

1. 这个函数应该只使用在单轮差异计算。这样的话，不用多次从源文件中读取差异数据，对 I/O 的提升是可预期地提高的。特别是数据量较大，较多的情况下。

2. 这个函数可以使用在多轮计算中，但是有可能会在处理差异数据时有重叠。即，在缩小差异块大小时会不断地处理第一个差异块中的二次、及多次差异数据结果。如果调用者将差异数据写入，则可能导致同一块数据的某个部分多次写入，导致不必要的写操作。当然，调用者可以不写入数据，而通过复制数据到一个新位置后，通过多次函数的调用后，将那些判定为相同的数据从最初的差异数据中排除出去，到最后一轮再写入余下的数据，也是可以的。但是调用者需要仔细处理这些数据。

**第一种处理方法：**

这里的处理，类似于文件洞的处理方法。即当第二次调用这个函数时，最初的数据范围中，应该只留下第二次调用中的范围，即 (offset, offset + datalen) 这个区间的数据。其余的数据则可以清除 ，余下的空间可以加收，或者保留不动，直到完成后再一次性清除（如果文件很大，有可能占用很大的内存空间，这个需要调用者视情况如何来处理）。其大意如下：

第一轮时（差异数据形成链表，并保存数据,X 表示差异数据）：

[XXXXXXXX]->[XXXXXXXXXXXXXXXXXXXXXXXX]->[XXXXXXXXXXXXXXXX]->...->|

第二轮时（将差异的数据之外的数据排除，形成最终的链表，X 表示差异数据，- 表示剔除的数据）：

[XXX--XXX]->[XX--XXXX----XXXXXXXXXXXX]->[XXXX-------XXXXX]->...->|，

经过第二轮后，差异数据链表成为如下（下一轮的差异数据不会超过任何之前轮的差异数据范围之外）：

[XXX]->[XXX]->[XX]->[XXXX]->[XXXXXXXXXXXX]->[XXXX]->[XXXXX]->...->|

通过这样的过程，直到最后一轮后，再将这剩下的差异数据写回。这些差异数据中带有其在源文件中的位置。因此在新文件中，只要 seek 到特定的位置，再写入对应长度的数据即可。

这种方法的优点时，在写出到文件或者存储时产生较少的 I/O，但是需要较大的内存，特别是要处理的单个数据单位的数据量（如单个文件）非常大时，也许是设备无法承受的在采用这种方法之间，要确保你有足够的内存来保存只够多的缓存数据。如果缓存不够，则可以考虑不缓存数据在结束时，再使用得到的差异数据链表从源文件中读取即可。只是增加了一点复杂度。

**第二种处理方法（较为简单）：**

就是在第一次接收到差异数据时，就直接 seek 到对应位置，将数据写入即可。后续的差异数据直接丢弃即可。

这种方法就是可能产生较大的 I/O，但是由于不用缓存，则处理大文件时有较大的余地。这实际上是退化成了单轮的方法了。）

**以上两种方法，都给库的使用造成了很大的使用复杂度，我们建议不要这么使用。如果你多轮计算中传入了这个参数，本库不保证性能。同时正确性由调用者来保证。**

3. 由于在就地生成的差异数据中，有可能会变动数据的位置，需要等处理完所有的相同块数据后才能写入差异数据块，否则可能会破坏目标文件的数据，导致数据错误。因此你需要对这些差异数据进行缓存才能保证正确。而缓存，有可能导致占用大量的内存资源，因此与就地生成的目标是相冲突的。如果不缓存，就需要对差异数据存入文件进行缓存等，而这也与就地生成的原始目标是相冲突的。**因此，在就地生成的场景下，我们最好不要使用差异数据回调处理函数。如果你使用了，则你要保证最终数据的正确性。**

具体的示例，请参考 test/testdiffcb.cpp 文件中的代码。

# 五、未来的变化

随着库的完善，有可能这个库的实现会有变化。我努力保持这个库与本文档的同步。比如在 C++ 的调用方式中有可能会有变化。但是目前暂时不动了。

# 六、参考文档

1. Efficient Algorithms for Sorting and Synchronization，Andrew Tridgell， A thesis submitted for the degree of Doctor of Philosophy at The Australian National University，February 1999。
2. In-Place Rsync: File Synchronization for Mobile and Wireless Devices，David Rasch and Randal Burns Department of Computer Science Johns Hopkins University {frasch,randalg}@cs.jhu.edu。
3. J. Langford. Multiround Rsync. Technical Report Available at www.cs.cmu.edu/jcl/research/mrsync/-mrsync.ps, Dept. of Computer Science, Carnegie-Mellon University, 2001.
4. Udi Manber.Finding Similar Files in a Large File System.1993
5. Moses S. Charikar.Similarity Estimation Techniques from Rounding Algorithms.
6. Jesse Kornblum.Identifying almost identical files using context triggered piecewise hashing.2006

# 七、感谢

感谢上面论文的作者，没有他们的工作，就不会有这个库的存在，就不会有那么多好的数据处理算法的存在。

1. 请参考：https://en.wikipedia.org/wiki/Rolling\_hash [↑](#footnote-ref-1)
2. 请参考：https://en.wikipedia.org/wiki/Rolling\_hash [↑](#footnote-ref-2)