|  |
| --- |
| [公司名称] |
| [文档标题] |
| [文档副标题] |

|  |
| --- |
| zuo Oliver  [日期] |

目录

[zM3vcf 工具说明 3](#_Toc161869359)

[1，编译生成 3](#_Toc161869360)

[2，命令行工具功能 3](#_Toc161869361)

[3，示例 4](#_Toc161869362)

[4，接口测试程序 5](#_Toc161869363)

[5，软件加速原理 5](#_Toc161869364)

[Vcflib库说明 7](#_Toc161869365)

[vcflib功能 7](#_Toc161869366)

[vcflib 文件结构 7](#_Toc161869367)

[vcflib 接口 7](#_Toc161869368)

[1．文件打开关闭接口 7](#_Toc161869369)

[2．文件读接口 9](#_Toc161869370)

[3．文件写接口 12](#_Toc161869371)

[4．其他接口 13](#_Toc161869372)

[5．vcf全格式接口 14](#_Toc161869373)

[vcflib 用户可调节的宏参数 16](#_Toc161869374)

[Tools说明 17](#_Toc161869375)

[1，编译生成 17](#_Toc161869376)

[2，工具说明 17](#_Toc161869377)

[附录1 Windows编译平台搭建 20](#_Toc161869378)

[Windows环境安装tdm64-gcc 20](#_Toc161869379)

[1.tdm64-gcc安装包 20](#_Toc161869380)

[2.安装tdm64-gcc 20](#_Toc161869381)

[Zlib库的编译与移植 23](#_Toc161869382)

[1.zlib安装包 23](#_Toc161869383)

[2.zlib编译移植 24](#_Toc161869384)

# zM3vcf 工具说明

## 1，编译生成

1）依赖库：gcc zlib pthread openmp

2）Linux平台下：

进入项目文件夹后，选择进入Linux版本目录

**cd zM3vcf**

编译 vcflib 库，并生成vcflib库文件到指定目录

**make vcflib**

编译并生成zM3vcf工具和m3vcf接口测试程序m3vcfTest

**make**

3）Windows平台下：

首先，需要安装tdm64-gcc编译器（下载网址<https://jmeubank.github.io/tdm-gcc/>，另在win\_supportPackage中有下载好的安装包tdm64-gcc-10.3.0-2.exe），在安装该编译器时，一定要在安装选项时勾选安装openMP，否则无法正常编译本项目（安装详细介绍见附录1）。

另外，需要下载并用tdm64-gcc编译zlib库（下载网址<https://www.zlib.net/>，win\_supportPackage中有下载好的源码包zlib-1.2.13.tar.gz），将编译成功的zconf.h zlib.h拷贝至tdm64-gcc安装目录中的include文件夹，将编译成功的libz.a拷贝至tdm64-gcc安装目录中的lib文件夹即可（安装详细介绍见附录1）。

进入cmd命令窗口，进入项目文件夹后，选择进入Windows版本目录

**cd zM3vcf\_win**

编译 vcflib 库，并生成vclib库文件到指定目录

**mingw32-make vcflib**

编译并生成zM3vcf.exe命令行工具和m3vcf接口测试程序m3vcfTest.exe

**mingw32-make**

## 2，命令行工具功能

1）Linux平台下：

compress: vcf->m3vcf

-b 压缩块的记录数[1000]

-o 输出文件名

-O 输出文件类型：M不压缩 m压缩

-t 用户指定的（读，写，压缩）最大pthread数量, 该值不得小于3，不指定时默认为8（1个读线程，1个写线程，6个压缩线程）。**注：该线程数不包含vcflib所使用的openMP线程数**

-m 用户指定的最大内存数量 GB（建议不指定）

convert：m3vcf->vcf

-o 输出文件名

-O 输出文件类型：M不压缩 m压缩

2）Windows平台下：

与Linux功能及参数相同。

## 3，示例

1）在项目文件夹中，用zM3vcf命令行工具将测试文件夹testFile中的vcf文件（该文件为截取[ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20110521/ALL.chr22.phase1\_release\_v3.20101123.snps\_indels\_svs.genotypes.vcf.gz文件的前20个Markers中的前10个samples](ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20110521/ALL.chr22.phase1_release_v3.20101123.snps_indels_svs.genotypes.vcf.gz文件的前10000个Markers)生成的）：

**ALL.chr22.20Markers.10Samples.vcf.gz**

压缩成testFile目录下的m3vcf文件

**ALL.chr22.20Markers.10Samples.m3vcf.gz**

Linux下执行如下命令：

**./zM3vcf compress testFile/ALL.chr22.20Markers.10Samples.vcf.gz -O m -o testFile/ALL.chr22.20Markers.10Samples.m3vcf.gz**

Windows下执行如下命令：

**zM3vcf.exe compress testFile\ALL.chr22.20Markers.10Samples.vcf.gz -O m -o testFile\ALL.chr22.20Markers.10Samples.m3vcf.gz**

2）在项目文件夹中，用zM3vcf命令行工具将测试文件夹testFile中上面生成的m3vcf文件（该文件为上面正确操作后由ALL.chr22.20Markers.10Samples.vcf.gz文件压缩生成的m3vcf文件）：

**ALL.chr22.20Markers.10Samples.m3vcf.gz**

还原到当前工作目录下的vcf文件：

**ALL.chr22.20Markers.10Samples.vcf.gz**

Linux下执行如下命令：

**./zM3vcf convert testFile/ALL.chr22.20Markers.10Samples.m3vcf.gz -O m -o ALL.chr22.20Markers.10Samples.vcf.gz**

Windows下执行如下命令：

**zM3vcf.exe convert testFile\ALL.chr22.20Markers.10Samples.m3vcf.gz -O m -o ALL.chr22.20Markers.10Samples.vcf.gz**

## 4，接口测试程序

项目文件夹中的m3vcfTest(Windows平台下是m3vcfTest.exe)测试程序，演示的是利用m3vcf功能库中的接口函数，将测试文件夹testFile中的vcf文件ALL.chr22.20Markers.10Samples.vcf.gz压缩成测试文件夹testFile中的m3vcf文件ALL.chr22.20Markers.10Samples.m3vcf.gz的整个流程， 和将生成的测试文件夹testFile中的m3vcf文件ALL.chr22.20Markers.10Samples.m3vcf.gz解压还原成测试文件夹testFile中的vcf文件ALL.chr22.20Markers.10Samples.IF.vcf.gz的整个流程。用户可以通过这个测试程序，学习如何使用接口函数的方式调用m3vcf功能库，在自己的代码中简洁方便的完成vcf文件与m3vcf文件之间的高速转换。

## 5，软件加速原理

1）原M3vcftools完成一次压缩工作的主要步骤为：

Read：从vcf文件中读取数据并解析至对应内存结构中

Compress：将vcf文件记录压缩转成m3vcf文件记录

Write：将生成的数据写入m3vcf文件中

由于原程序为单线程，故完成三次压缩工作在cpu中的一个核内流程如下：



2）zM3vcf，首先是将Read，Compress，Write三个步骤分别独立出来，放到不同的线程中完成。



故完成同样三次压缩工作是在cpu中的多个核内并行完成的，流程如下：



与上面图的纵向对比可以发现，同样完成三次压缩工作，原程序需要9个步骤的时间，而本程序只需要5个步骤的时间就可以完成，速度会有大幅的提高，而且压缩次数越多，效果显著。通过这种方法可以完成一部分提速。

3）通过实验获得，Read，Compress，Write三个步骤中，Compress占用时间最大，故采用多线程pthread的方法，将Compress步骤并行运行在cpu的多个核中。使Read，Compress，Write三个步骤的时间接近相等匹配。



通过这种方法可以进一步完成提速。

4）由于Read步骤中，可以细分为：

Read Data：从硬盘中将数据读入内存

Parsing：将内存中的原始数据解析到对应的结构体内存中

Parsing部分占据了整个Read步骤的绝大部分时间，故在Parsing部分采用了openMP的多线程形式进行加速。



软件中由vcflib部分完成这部分功能。通过这种方法，可以完成Read步骤的进一步提速。

# Vcflib库说明

## vcflib功能

该库主要功能是可以高速读取vcf文件进内存中。支持未压缩的vcf文件和zlib压缩过的gz格式文件。

## vcflib 文件结构

**Makefile：**该库的编译规则文件，通过make完成对该库的编译

**bin：**存放正确编译完毕后，生成的该库相关文件**（未编译前为空）**

vcflib.h:该库的头文件，其中有该库提供的所有函数接口的相关声明,及相关用户可调节的宏参数

libvcf.so:该库的动态库文件

libvcf.a: 该库的静态库文件

**inc：**存放该库的头文件

baseTool.h: 该库中用到的内部基本功能函数的声明

vcflib.h:该库的头文件，其中有该库提供的所有函数接口的相关声明,及相关用户可调节的宏参数

**obj：**存放该库在编译过程中产生的中间文件**（未编译前为空）**

**src：**存放该库的源码文件

baseTools.c:该库中用到的内部基本功能函数

vcfFile.c：该库中的主要功能接口函数

**tst：**存放该库的示例代码，包含一个simpleRead（最简读操作）的示例程序。

## vcflib 接口

### 1．文件打开关闭接口

**VCF\_STATUS vcfFileOpen(VCF\_FILE \*fp,const char \*fileName, FILE\_MODE fileMode, unsigned int parseItem);**

**功能：**

以读的方式打开一个存在的vcf文件，支持未压缩的文本文件格式，支持zlib压缩的gz格式。并开启vcflib日志

**参数：**

fp：结构体VCF\_FILE的指针。vcflib库中的文件handle，结构体定义如下：

typedef struct

{

union

{

FILE \* fp; //未压缩格式文件指针

gzFile gfp; //zlib压缩格式文件指针

}fp;

FILE\_MODE mode; //存储第3个参数fileMode的内容

//记录该文件的压缩格式

//FILE\_MODE\_NORMAL :未压缩

//FILE\_MODE\_GZ: zlib压缩gz格式

FILE\_HEAD head; //用于存储该vcf文件的文件头数据

//在未作读文件头操作之前，内容为空

int numSamples; //该vcf文件的sample个数

//在未作读文件头操作之前，该值为0

//读过文件头后，该值为文件的sample个数

unsigned int parsingItems; //用户设置解析marker中的字段选项bit

//P\_GT是解析GT字段(当前为默认解析)

//P\_DS是解析DS字段

}VCF\_FILE;

fileName：要打开的vcf文件路径及名称

fileMode：指定打开文件的压缩格式，FILE\_MODE\_NORMAL表示未压缩；FILE\_MODE\_GZ表示zlib压缩gz格式

parseItem：指定对文件进行读操作时，解析哪些字段。P\_DS|P\_GT表示读操作时将同时解析DS和GT字段；P\_GT表示读操作时将只解析GT字段

**返回值：**

VCF\_ERROR：打开失败

VCF\_OK：打开成功

**VCF\_STATUS vcfFileCreate(VCF\_FILE \*fp,const char \*fileName,FILE\_MODE fileMode);**

**功能：**

以写的方式创建一个vcf文件。若文件已经存在则清空其中内容。支持未压缩的文本文件格式，支持zlib压缩的gz格式。并开启vcflib日志

**参数：**

fp：结构体VCF\_FILE的指针；详解见上（**vcfFileOpen**）

fileName：要打开的vcf文件路径及名称

fileMode：指定打开文件的压缩格式，FILE\_MODE\_NORMAL表示未压缩；FILE\_MODE\_GZ表示zlib压缩gz格式

**返回值：**

VCF\_ERROR：创建/打开失败

VCF\_OK：创建/打开成功

**VCF\_STATUS vcfFileAppend(VCF\_FILE \*fp,const char \*fileName,FILE\_MODE fileMode);**

**功能：**

以写的方式创建一个vcf文件。若文件已经存在则将保留其中内容，并将文件写指针定位到文件尾。支持未压缩的文本文件格式，支持zlib压缩的gz格式。并开启vcflib日志

**参数：**

fp：结构体VCF\_FILE的指针；详解见上（**vcfFileOpen**）

fileName：要打开的vcf文件路径及名称

fileMode：指定打开文件的压缩格式，FILE\_MODE\_NORMAL表示未压缩；FILE\_MODE\_GZ表示zlib压缩gz格式

**返回值：**

VCF\_ERROR：追加/打开失败

VCF\_OK：追加/打开成功

**VCF\_STATUS vcfFileClose(VCF\_FILE \*fp);**

**功能：**

关闭已经打开的文件，清空并释放文件头所占用的内存空间。并关闭vcflib日志

**参数：**

fp：结构体VCF\_FILE的指针；详解见上（**vcfFileOpen**）

**返回值：**

VCF\_ERROR：关闭失败

VCF\_OK：关闭成功

### 2．文件读接口

**VCF\_STATUS vcfFileReadLine(VCF\_FILE \*fp,char \*lineStr,int lineSize);**

**功能：**

从打开的fp所指向的vcf文件中，读取当前行，将内容存入lineStr中。

**参数：**

fp：结构体VCF\_FILE的指针

lineStr：存储读取文件内容的内存指针

lineSize：lineStr所指向的内存长度

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**VCF\_STATUS vcfFileReadHead(VCF\_FILE \*fp);**

**功能：**

从打开的fp所指向的vcf文件中，读取vcf文件头部分，将内容存入fp->head所指向的结构体中。

**参数：**

fp：结构体VCF\_FILE的指针,其中的成员变量head为结构体FILE\_HEAD类型的变量。用于存储该vcf文件的文件头，结构体定义如下：

typedef struct

{

char \*\*metaInfoLines; //用于存储vcf文件头的meta information line

int numMetaInfoLines; //vcf文件头的meta information line行数

char \*headerLine; //用于存储vcf文件头的header line

}FILE\_HEAD;

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**VCF\_STATUS getNumMetaInfoLines(FILE\_HEAD \*fhp,int \*NumMetaInfoLines);**

**功能：**

从fhp所指向的vcf文件头中，获取meta information line行数

**参数：**

fhp：结构体FILE\_HEAD的指针。详解见上（**vcfFileReadHead**）

NumMetaInfoLines：所得到的fhp所指向的vcf文件头中meta information line行数

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**VCF\_STATUS getNumSamples(VCF\_FILE \*fp,int \*NumSamples);**

**功能：**

从打开的fp所指向的vcf文件中，获取该文件的sample个数

**参数：**

fp：结构体VCF\_FILE的指针

NumSamples：所获得的sample个数

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**char\* vcfFileParseDataLineInfo(char \*lineStr,DATA\_INFO \*dataInfo);**

**功能：**

将lineStr内的一行vcf文件的数据中的前九项，解析到dataInfo指向的DATA\_INFO结构体中。操作后lineStr中去掉前九项

**参数：**

lineStr：vcf文件的一行数据

dataInfo：指向的DATA\_INFO结构体指针。vcflib库中用于存储vcf文件中每行数据的前九项，结构体定义如下：

typedef struct

{

char \*chrom;

char \*pos;

char \*ID;

char \*ref;

char \*alt;

char \*qual;

char \*filter;

char \*info;

char \*format;

}DATA\_INFO;

**返回值：**

去掉前九项的vcf文件数据行

**VCF\_STATUS vcfFileParseDataLine(VCF\_FILE \*fp,char \*lineStr,DATA\_LINE \*dlp);**

**功能：**

将lineStr内的一行vcf文件的数据，解析到dlp指向的DATA\_LINE结构体中

**参数：**

fp：结构体VCF\_FILE的指针

lineStr：vcf文件的一行数据

dlp：指向的DATA\_LINE结构体指针。vcflib库中用于存储vcf文件中每行数据，结构体定义如下：

typedef struct

{

char \*rawDataLine; //该行数据的原串首地址，不是整字符串

//其中前九项的\t已经被替换成\0

DATA\_INFO dataInfo; //该行数据的前九项

char \*samplesRawString; //该行数据去掉前九项后的数据内容原串

char \*gtData; //该行数据GT部分数据

float \*dsData; //该行数据DS部分数据

int numSamples; //该行数据Sample的个数

}DATA\_LINE;

**返回值：**

VCF\_ERROR：解析失败

VCF\_OK：解析成功

**VCF\_STATUS vcfFileReadDataLine(VCF\_FILE \*fp,DATA\_LINE \*dlp);**

**功能：**

从打开的fp所指向的vcf文件中，读取当前数据行，将内容存入dlp指向的DATA\_LINE结构体中

**参数：**

fp：结构体VCF\_FILE的指针

dlp：指向的DATA\_LINE结构体指针。详解见上（**vcfFileParseDataLine**）

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**VCF\_STATUS vcfFileReadDataBlock(VCF\_FILE \*fp,DATA\_BLOCK \*dbp,int numLines);**

**功能：**

从打开的fp所指向的vcf文件中，读取numLines个数据行，将内容存入dbp指向的DATA\_BLOCK结构体中

**参数：**

fp：结构体VCF\_FILE的指针

dbp：指向的DATA\_BLOCK结构体指针。vcflib库中用于存储vcf文件中每块数据，结构体定义如下：

typedef struct

{

DATA\_LINE \*dataLines; //存储该数据块的多行数据的数组指针

int numDataLines; //该数据块数据行数

}DATA\_BLOCK;

numLines：从文件中一次性读取的行数

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**VCF\_STATUS vcfFileReadDataBlockOverlap1Line(VCF\_FILE \*fp,DATA\_BLOCK \*dbp,int numLines);**

**功能：**

从打开的fp所指向的vcf文件中，读取numLines个数据行，将内容存入dbp指向的DATA\_BLOCK结构体中。除第一次以外，后面每次读取的数据块的第一行内容都是上次读取的数据块中最后一行内容。即有一行overlap

**参数：**

fp：结构体VCF\_FILE的指针

dbp：指向的DATA\_BLOCK结构体指针。详解见上（**vcfFileReadDataBlock**）

numLines：从文件中一次性读取的行数

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

### 3．文件写接口

**VCF\_STATUS vcfFileWriteLine(VCF\_FILE \*fp,char \*lineStr);**

**功能：**

将lineStr内容，做为一行写入打开的fp所指向的vcf文件中

**参数：**

fp：结构体VCF\_FILE的指针

lineStr：存储写入文件内容的内存指针

**返回值：**

VCF\_ERROR：写入失败

VCF\_OK：写入成功

**VCF\_STATUS vcfFileWriteHead(VCF\_FILE \*fp,FILE\_HEAD \*fhp);**

**功能：**

将fhp所指向的结构体中的vcf文件头信息，写入打开的fp所指向的vcf文件中

**参数：**

fp：结构体VCF\_FILE的指针

fhp：结构体FILE\_HEAD的指针

**返回值：**

VCF\_ERROR：写入失败

VCF\_OK：写入成功

**VCF\_STATUS vcfFileAddMetaInfoLine(FILE\_HEAD \*fhp,int posIndex,char \*MetaInfoLine);**

**功能：**

将MetaInfoLine内的meta information line增加到fhp所指向的vcf文件头结构体中，并指定新增加的meta information line位置在meta information中的第posIndex行

**参数：**

fhp：结构体FILE\_HEAD的指针

posIndex：指定写入meta information中的第几行

MetaInfoLine：要写入的meta information内容

**返回值：**

VCF\_ERROR：写入失败

VCF\_OK：写入成功

**VCF\_STATUS vcfFileRemoveMetaInfoLine(FILE\_HEAD \*fhp,int posIndex);**

**功能：**

将fhp所指向的vcf文件头结构体中，第posIndex条meta information line删除

**参数：**

fhp：结构体FILE\_HEAD的指针

posIndex：要删除meta information中的第几行

**返回值：**

VCF\_ERROR：删除失败

VCF\_OK：删除成功

### 4．其他接口

**void clearFileHead(FILE\_HEAD \*fhp);**

**功能：**

清空fhp所指向的FILE\_HEAD结构体中的数据并释放其内存空间

**参数：**

fhp：结构体FILE\_HEAD的指针

**返回值：**

无

**void clearDataLine(DATA\_LINE \*dlp);**

**功能：**

清空dlp所指向的DATA\_LINE结构体中的数据并释放其内存空间（注：若调用vcfFileReadDataLine接口后，用相同dlp参数再次调用vcfFileReadDataLine接口之前，需要调用本接口清空并释放上一次DATA\_LINE结构体中的内容，否则会造成内存泄露）

**参数：**

dlp：结构体DATA\_LINE的指针

**返回值：**

无

**void clearDataBlock(DATA\_BLOCK \*dbp);**

**功能：**

清空dbp所指向的DATA\_BLOCK结构体中的数据并释放其内存空间

**参数：**

dbp：结构体DATA\_BLOCK的指针

**返回值：**

无

**VCF\_STATUS vcfPopSubString(char \*\*lineStr,char \*subStr);**

**功能：**

从lineStr所指向的字符串中，截出用空格、制表符‘\t’、换行符‘\n’所分隔的第一段子串，存入subStr中。操作后lineStr中去掉截出的子串

**参数：**

lineStr：待截断的长字符串

subStr: 截出来的子串

**返回值：**

VCF\_ERROR：截取失败；或待截长字符串为空（长字符串全部截完）

VCF\_OK：截取成功

**void printDataLine(DATA\_LINE \*dlp);**

**功能：**

打印显示dlp指向的DATA\_LINE结构体中的所有成员变量的值，用于显示dlp中存储内容。

**参数：**

dlp：结构体DATA\_LINE的指针

**返回值：**

无。

### 5．vcf全格式接口

**VCF\_STATUS vcfFileParseDataLine\_allFormat(VCF\_FILE \*fp,char \*lineStr,DATA\_LINE\_ALL\_FORMAT \*** **dlafp);**

**功能：**

将lineStr内的一行vcf文件的数据，解析到dlafp指向的DATA\_LINE\_ALL\_FORMAT结构体中

**参数：**

fp：结构体VCF\_FILE的指针

lineStr：vcf文件的一行数据

dlafp：指向的DATA\_LINE\_ALL\_FORMAT结构体指针。vcflib库中用于存储vcf文件中支持全格式的每行数据，结构体定义如下：

typedef struct

{

char \*rawDataLine; //该行数据的原串首地址，不是整字符串

//其中前九项的\t已经被替换成\0

DATA\_INFO dataInfo; //该行数据的前九项

int numFormats; //该行数据format项的个数

DATA\_FORMAT\_STR \*dataFormatStr;//DATA\_FORMAT\_STR结构体指针

//根据numFormats分配空间

//存储该行数据所有format数据内容

int numSamples; //该行数据Sample的个数

}DATA\_LINE;

typedef struct

{

char \*format; //format名称

char \*\*dataStr; //每个sample该format的字符串形式的具体值

}DATA\_FORMAT\_STR;

**返回值：**

VCF\_ERROR：解析失败

VCF\_OK：解析成功

**VCF\_STATUS vcfFileReadDataLine\_allFormat(VCF\_FILE \*fp,DATA\_LINE\_ALL\_FORMAT \*dlafp);**

**功能：**

从打开的fp所指向的vcf文件中，读取当前数据行，将内容存入dlafp指向的DATA\_LINE\_ALL\_FORMAT结构体中

**参数：**

fp：结构体VCF\_FILE的指针

dlafp：指向的DATA\_LINE\_ALL\_FORMAT结构体指针。详解见上（**vcfFileParseDataLine\_allFormat**）

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**VCF\_STATUS vcfFileReadDataBlock\_allFormat(VCF\_FILE \*fp,DATA\_BLOCK\_ALL\_FORMAT \*dbafp,int numLines);**

**功能：**

从打开的fp所指向的vcf文件中，读取numLines个数据行，将内容存入dbafp指向的DATA\_BLOCK\_ALL\_FORMAT结构体中

**参数：**

fp：结构体VCF\_FILE的指针

dbafp：指向的DATA\_BLOCK\_ALL\_FORMAT结构体指针。vcflib库中用于存储vcf文件中每块数据，结构体定义如下：

typedef struct

{

DATA\_LINE\_ALL\_FORMAT \*dataLines; //存储该数据块的多行数据的数组指针

int numDataLines; //该数据块数据行数

}DATA\_BLOCK\_ALL\_FORMAT;

numLines：从文件中一次性读取的行数

**返回值：**

VCF\_ERROR：读取失败

VCF\_OK：读取成功

**void clearDataLine\_allFormat (DATA\_LINE\_ALL\_FORMAT \*dlafp);**

**功能：**

清空dlafp所指向的DATA\_LINE\_ALL\_FORMAT结构体中的数据并释放其内存空间（注：若调用vcfFileReadDataLine\_allFormat接口后，用相同dlafp参数再次调用vcfFileReadDataLine\_allFormat接口之前，需要调用本接口清空并释放上一次DATA\_LINE\_ALL\_FORMAT结构体中的内容，否则会造成内存泄露）

**参数：**

dlafp：结构体DATA\_LINE\_ALL\_FORMAT的指针

**返回值：**

无

**void clearDataBlock\_allFormat (DATA\_BLOCK\_ALL\_FORMAT \*dbafp);**

**功能：**

清空dbafp所指向的DATA\_BLOCK\_ALL\_FORMAT结构体中的数据并释放其内存空间

**参数：**

dbafp：结构体DATA\_BLOCK\_ALL\_FORMAT的指针

**返回值：**

无

**void printDataLine\_allFormat (DATA\_LINE\_ALL\_FORMAT \*dlafp);**

**功能：**

打印显示dlafp指向的DATA\_LINE\_ALL\_FORMAT结构体中的所有成员变量的值，用于显示dlafp中存储内容。

**参数：**

dlafp：结构体DATA\_LINE\_ALL\_FORMAT的指针

**返回值：**

无。

## vcflib 用户可调节的宏参数

在inc/vcflib.h文件中，有一些宏定义，用户可以根据不同的需求进行动态调整。

**BT\_MAX\_LINE\_SIZE：**当前该库支持的读取vcf文件中，一行中最大的字符数量。当前值为6M。若因需要处理的vcf文件一行字符数特别大而产生了错误，可以将其值改大后重新编译即可。

**OPEN\_MP\_THREAD\_NUM：**在进行读取vcf文件中的数据解析时，开启的openMP的线程数。当前值为10。用户可以提高该值来提高整个库读取vcf文件的速度，但整体速度受硬盘I/O限制和CPU核数限制。

# Tools说明

zM3vcf文件夹中配套提供了一些关于在测试该项目速度和效率的时候，使用的一些小工具程序。

## 1，编译生成

编译方法如下：

1）Linux平台下：

进入项目文件夹后，选择进入Linux版本目录

**cd zM3vcf**

编译 vcflib 库，并生成vcflib库文件到指定目录

**make vcflib**

进入小工具文件夹

**cd tools**

编译并生成小工具

**make**

2）Windows平台下：

进入cmd命令窗口，进入项目文件夹后，选择进入Windows版本目录

**cd zM3vcf\_win**

编译 vcflib 库，并生成vcflib库文件到指定目录

**mingw32-make vcflib**

进入小工具文件夹

**cd tools**

编译并生成小工具

**mingw32-make**

## 2，工具说明

1）pVcf

其源码为tools文件夹中的copyCreateVcf.c。其功能是用真实的vcf文件填充创建一个原vcf文件n倍sample的新的vcf测试文件。即用真实vcf的每一行扩充n倍（每行扩充顺序随机打乱）合成目标vcf的一行。需要vcf真实文件。

其用法为：

**pVcf sourceVCFFile sampleTimes targetVCFFile**

其中：

sourceVCFFile为原vcf文件的路径

sampleTimes为原vcf文件扩充多少倍的倍数

targetVCFFile为生成的新vcf文件的路径

示例，用该工具将testFile文件夹中的ALL.chr22.20Markers.10Samples.vcf.gz文件，sample数扩充2倍，在testFile文件夹中生成新文件ALL.chr22.20Markers.20Samples.vcf.gz，命令如下（当前工作目录在tools文件夹中）：

**./pVcf ../testFile/ ALL.chr22.20Markers.10Samples.vcf.gz 2 ../ ALL.chr22.20Markers.20Samples.vcf.gz**

（windows用法参考上述Linux操作）

2）gVMH

其源码为tools文件夹中的getVcfMH.c。其功能是打印指定vcf文件中的marker数和sample数，并测试vcf文件中每一个marker是否符合sample数

其用法为：

**gVMH VCFFile**

其中：

VCFFile为要查看的vcf文件的路径

示例，用该工具将testFile文件夹中的ALL.chr22.20Markers.10Samples.vcf.gz文件，中的sample个数和marker个数打印出来，并且检查是否每行marker的sample值都一致，命令如下（当前工作目录在tools文件夹中）：

**./gVMH ../testFile/ ALL.chr22.20Markers.10Samples.vcf.gz**

（windows用法参考上述Linux操作）

3）vCM

其源码为tools文件夹中的vcfCutMarkers.c。其功能是从指定大文件中截取前n个markers、和每个marker中的前m个samples的内容生成一个新的小vcf文件。

其用法为：

**vCM sourceVCFFile targetVCFFile markersNumber samplesNumber**

其中：

sourceVCFFile为作为数据源的大vcf文件的路径

targetVCFFile为生成的小的vcf文件的路径

markersNumber为从大文件中截取的marker个数，当其值为0时，保留所有marker。

samplesNumber为从大文件中截取的sample个数，当其值为0时，保留所有sample。

示例，用该工具将testFile文件夹中的ALL.chr22.20Markers.10Samples.vcf.gz文件，截取其中的10个markers和5个samples，在testFile中生成ALL.chr22.10Markers.5Samples.vcf.gz命令如下（当前工作目录在tools文件夹中）：

**./vCM ../testFile/ ALL.chr22.20Markers.10Samples.vcf.gz ../testFile/ALL.chr22.10Markers.5Samples.vcf.gz 10 5**

（windows用法参考上述Linux操作）

4）readVcfTest

其源码为tools文件夹中的readVcfTest.c。其功能是测试vcflib的读取vcf文件的速度及其占用的资源情况，只读，不做任何操作

5）readvcf.c

用于测试htslib读取vcf文件的速度及其占用的资源情况，只读，不做任何操作

6）cmWatcher.sh

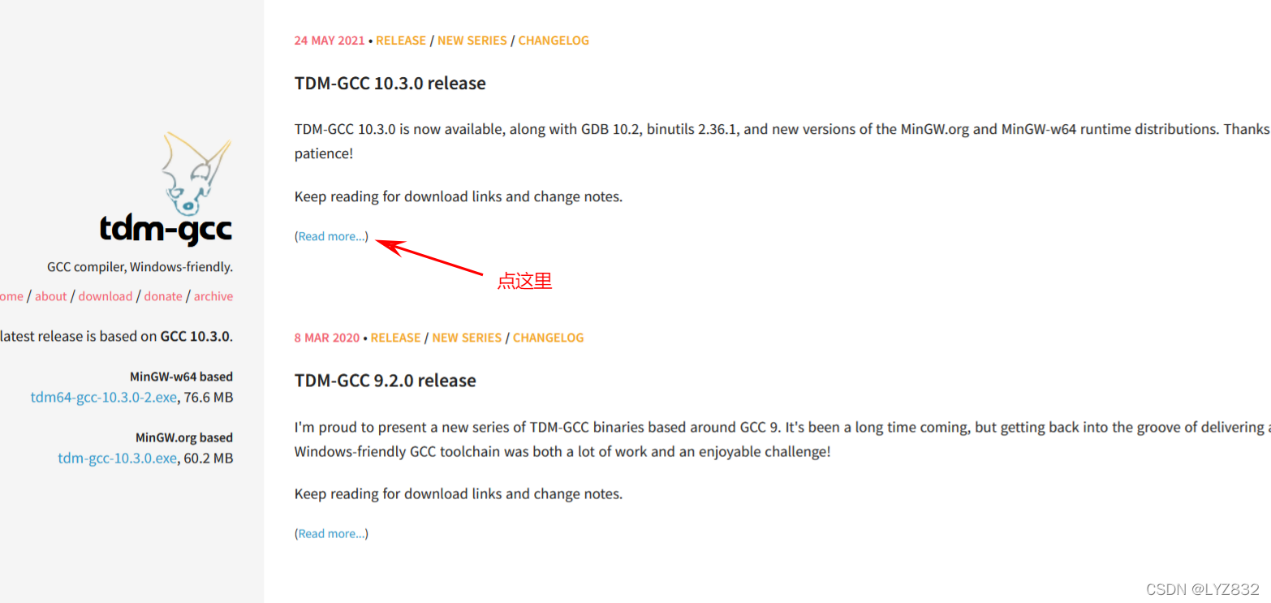
用于监控指定进程的cpu占用率峰值，内存占用率峰值，并输出到指定文本文档中。

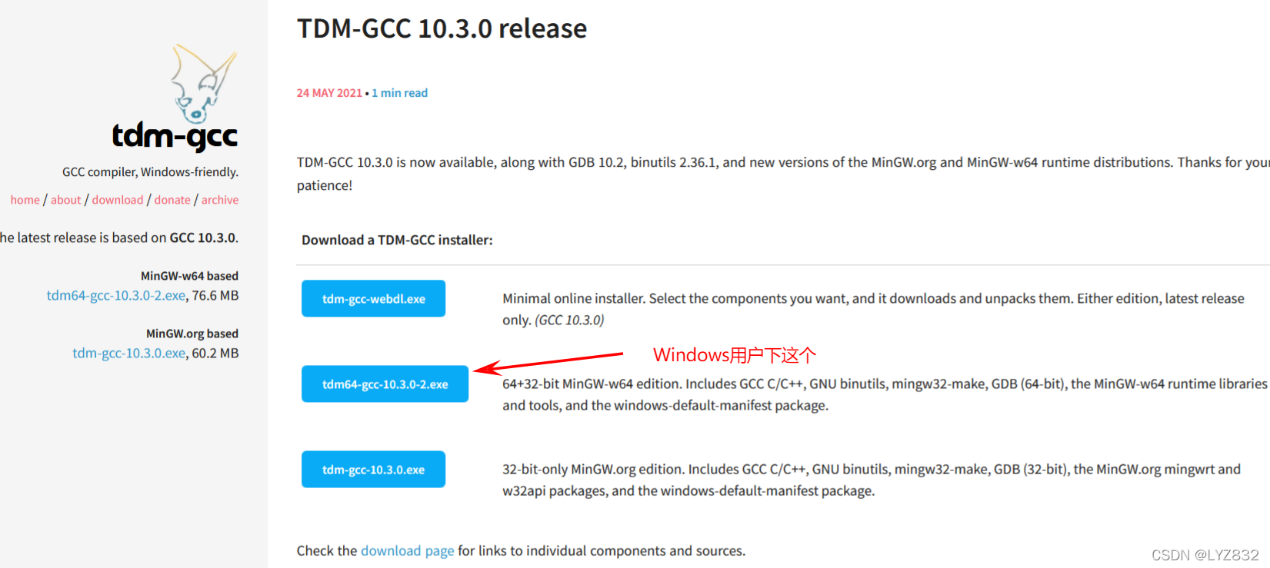
# 附录1 Windows编译平台搭建

## Windows环境安装tdm64-gcc

### 1.tdm64-gcc安装包

进入官网（<https://jmeubank.github.io/tdm-gcc/>）直接下载

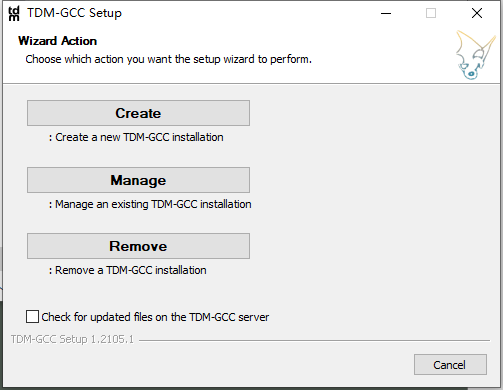




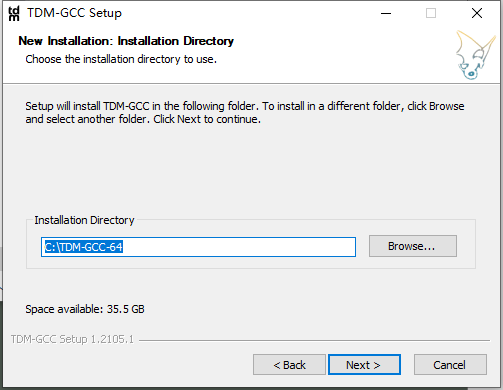
或在我提供的win\_supportPackage中有下载好的安装包tdm64-gcc-10.3.0-2.exe。

### 2.安装tdm64-gcc

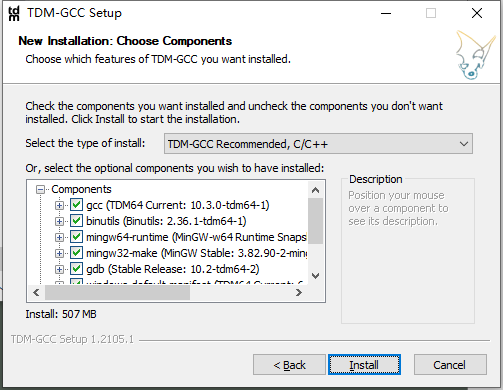
下图为弹出的安装向导：



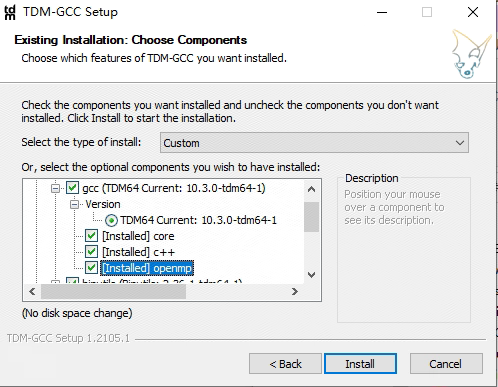
取消“check for updated files on the TDM-GCC server”选项，然后点击Create。



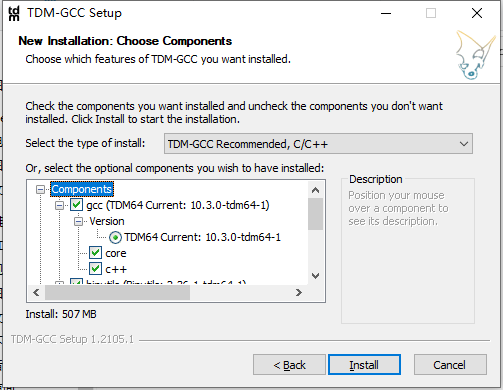
选择一个安装路径，不要有中文路径和特殊符号，然后点击Next。



所有选项全部选中。这里一定注意第一项“gcc”中是否有openmp，一定选中该项。如下图：



如果在第一项中无openmp，如下图：



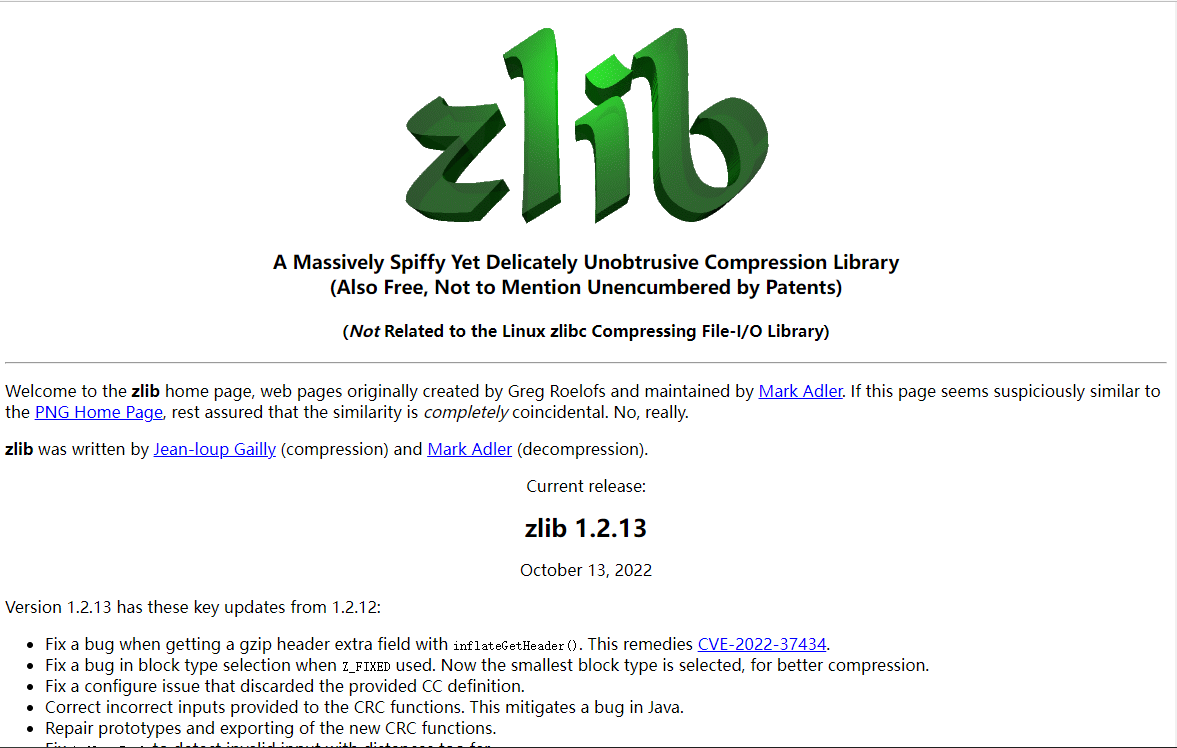
则安装完后，要重新双击安装包进入安装向导，对安装完的tdm-gcc进行Manage，将openmp添加上，否则无法正常编译vcflib。

最后点击Install完成安装即可。

## Zlib库的编译与移植

### 1.zlib安装包

进入官网（<https://www.zlib.net/>）直接下载



或在我提供的win\_supportPackage中有下载好的tar包zlib-1.2.13.tar.gz。

### 2.zlib编译移植

在Windows下，进入cmd命令行。

将zlib的安装包解压到任意目录，我这里是将zlib包放在d:\tmp下，并在本地进行解压，输入如下命令：

**d：**

**cd tmp**

**tar -xzvf zlib-1.2.13.tar.gz**

进入解压好的zlib文件夹，将zlib中的win32\Makefile.gcc拷贝到zlib文件夹中，输入如下命令：

**cd zlib-1.2.13**

**copy win32\Makefile.gcc makefile.gcc**

编译zlib，输入如下命令：

**Mingw32-make -f makefile.gcc**

完成编译后，将编译成功的zconf.h zlib.h拷贝至tdm64-gcc安装目录中的include文件夹；将编译成功的libz.a拷贝至tdm64-gcc安装目录中的lib文件夹即可。