目录

序言

本翻译稿只为方便大家查阅手册使用，最好对照原英文手册阅读。稿中难免有很多不准确之处，应以英文为准，见仁见智。

只对手册作了部分翻译，陈旧的、高深的以及 C++ 相关的内容没有翻译，读者自己把握。

译者
2005 年 6 月 30 日

第 一 章 GMP 及其安装

1.1 介绍

GNU MP 是用 C 语言写成的一个便携式库，它可以进行整数、有理数和浮点数的任意精度算术，其目标是为所有需要不能由基本 C 类型直接支持的多精度类型的应用提供可能最快的算术。

许多应用只需要几百 bit 的精度，但是有些应用需要上千甚至上百万 bit 精度。通过根据操作数的规模选择算法，通过仔细的保持开销最小化，GMP 被设计成为各种应用提供最好的性能。

通过把整个字作为基本算术类型，通过使用精益求精的算法，通过为大部分的内部循环针对不同 CPU 包含认真优化的汇编代码，通过对速度 (与简洁性和优雅性不同) 的强调，GMP 实现了较快的速度。

如果希望获得关于 GMP 的最新消息，可以访问

http : //swox.com/gmp/

可以从该网站上获得 GMP 的最新源码和安装包 gmp-4.1.4.tar.bz2，或者从 GNU 官方 ftp 网址

ftp : //ftp.gnu.org/gnu/gmp

上获得，当然使用离你最近的镜像站点是最好的选择。本文针对 GMP 的 4.1.4 版，主要参考 gmp-4.1.4 用户手册。

1.2 GMP 在 UNIX 类系统下的安装

GMP 有一个基于 autoconf/automake/libtool 的配置系统，在 UNIX 类系统环境下基本编译可以通过

```
./configure
make
```

来完成，这时 autoconf 使用默认配置选项。自测试通过

```
make check
```

完成，你可以通过

```
make install
```

来安装 GMP(默认安装在'/usr/local/' 目录下)。GMP 的安装配置选项是可以被修改的，假设 gmp-4.1.4.tar.bz2 解压缩于 ~/gmp-4.1.4/，进入该目录键入

```
./configure --help
```

可以看到 GMP 的所有安装配置选项信息，下面对其中重要的几个作一些简单介绍。

“--prefix”和“--exec-prefix”

“--prefix”指定 GMP 安装目录，在 &prefix 下建立安装目录树，主要是将库文件 libgmp.a 等安装在 &prefix/lib/ 下，而将头文件 gmp.h 等安装在 &prefix/include/ 下，默认选项是‘--prefix=/usr/local/’。如果希望将库文件安装在不同的目录下，可以设置“--exec-prefix”，这时需要保证头文件在 &prefix/include/ 以及库文件在 &exec-prefix/lib/ 都是可编译链接的。

“--disable-shared”和“--disable-static”

在默认情形下动态和静态链接库会同时建立 (如果可能)，但是可以其中之一可以不建立。动态链接库可以实现更小的可执行文件体积并允许在不同的处理器之间共享代码，但是在有些 CPU 上动态链接库可能有一些慢因为需要额外的函数调用开销。

“--enable-cxx”

GMP 的 C++ 支持可以通过‘--enable-cxx’来实现，这时需要一个 C++ 编译器。C++ 支持有库文件 libgmpxx.a 和头文件 gmpxx.h 组成，在 GMP 中采用了分离的库文件 libgmpxx.a 而不是在库文件 libgmp.a 中包含 C++ 对象，这是为了保证动态链接 C 程序时不会因为要依赖 C++ 标准库而变得庞大，并且避免链接普通 C 程序时需要使用 C++ 编译器。libgmpxx.a 依赖于 libgmp.a，不同版本的 libgmp.a 是不能支持 libgmpxx.a 的，而且，libgmpxx.a 只能配合编译它的 C++ 编译器使用，因为在不同的编译器中名字处理和运行时库通常是不兼容的。

“--enable-mpbsd”

只有当‘--enable-mpbsd’被使用时，用于 Berkley MP 函数兼容的库文件 libmp.a 和头文件 mp.h 被建立和安装。

“--enable-mpfr”

MPFR 包含很多快速的浮点运算函数，现在也被集成在 GMP 的安装包中，只有当‘--enable-mpfr’被使用时，可选的 MPFR 函数库文件 libmpfr.a 和头文件 mpfr.h,mpf2mpfr.h,mpfrxx.h 被建立和安装。

GMP 安装还有其他很多配置选项，这里就不做介绍了，实际上 autoconf 会自动收集 CPU, 编译器等信息，所以这些不需要我们手动设置。下面是一个推荐配置:

```
./configure  --enable-mpbsd --enable-mpfr --enable-cxx
             --disable-shared
```

1.3 GMP 在 Windows 系统下的安装

GMP 包是在 UNIX 类系统上开发出来的，不幸的是 UNIX 类系统操作的不直观性使得很多人对它望而却步，而宁愿选择需要花钱的 Windows 操作系统。为了在 Windows 操作系统上使用 GMP 包，我们需要在 Windows 操作系统上模拟 UNIX 环境，有很多方法来实现它，其中 MinGW 和 Cygwin 是两个不错的选择。MinGW 直接应用 Windows 系统的 C 运行时库 msvcrt.dll 进行 I/O 处理，配合 MinSys，MinGW 可以在 Windows 操作系统上模拟简单的 UNIX 环境，它也可以配合 Cygwin 使用。Cygwin 定义了自己的动态链接库 cygwin1.dll，在 Cygwin 内部实现的应用一般都要通过这个动态链接库来运行，但是如果选用 MinGW 作为内部的编译环境，Cygwin 像 MinGW 那样实现应用。Cygwin 是一个比较完备的环境，它提供许多 UNIX 应用程序，使用它时你会感觉像在真正的 UNIX 类操作系统上工作一样。可以从网址

http : //www.mingw.org/
http : //www.cygwin.com/

分别得到 MinGW 和 Cygwin 的有关信息和安装方法。

MinGW 和 MinSys 的二进制安装包可以直接从其官方网站上获得，在得到了它们的二进制安装包后，应该首先安装 MinGW 再安装 MinSys，一般将 MinGW 安装于目录为 C:/MinSys/MinGW/，而 MinSys 安装于目录 C:/MinSys/，这样将 MinSys 文件夹拷贝到其他运行 Windows 的机器 C 盘根目录下，不需要任何其他的设置，MinSys 仍然是可以直接使用的。当然也可以选择任意的其他安装目录，在 MinSys 安装的最后会提示输入 MinGW 的安装目录。

下载 Cygwin 安装包一般需要先从其官方网站上获得 setup.exe 程序，运行 setup.exe，它会要求你选择下载到本地文件夹还是在线安装或者从本地文件夹安装，选定下载到本地文件夹后，它会提示网络连接方式，接下来就是选择镜像网站了，注意不要同时选择两个网址，你也可以输入列表中没有的镜像网站，国内有很多大学和机构都作了 Cygwin 的镜像，例如

ftp : //ftp.ctex.org/pub/cygwin/
ftp : //ftp.tsinghua.edu/pub/cygwin/

在顺利登陆镜像站点之后，会有一个应用程序树供你选择，除了默认的应用程序之外，GCC,vi,libtool 等也是必须的，如果网速和空间允许的话建议完全下载，但是安装的时候没有必要完全安装，那样需要约 1G 的空间，一般根据自己需要选择应用程序。如果想省去编译 GMP 的麻烦，甚至可以在这些应用程序中找到 GMP，但是由于机器的不同，直接安装的 GMP 可能不如在本机编译安装的 GMP 速度快。

有了 MinGW+MinSys 或者 Cygwin 之后，可以像上一节那样编译安装 GMP，但是默认情形下只建立静态库，如果希望建立动态库 DLL，需要使用

‘`--disable-static --enable-shared`’

选项，注意静态库和动态库是不能同时建立的。在建立 DLL 时‘`--enable-cxx`’是不可用的，因为目前 libtool 还不支持 C++ DLL。

习惯了 VC 的高手可能希望将 GMP 用于 VC 编程，这似乎不是很好办到，有两种方法也许可以，首先是使用源码，GMP 的源码就在压缩包 gmp-4.1.4.tar.bz2 中，具体怎么用 VC 高手也许有办法。其次我们可以用 MinGW 编译 GMP 建立动态库 libgmp.dll，但是它不会产生.lib 和.exp 文件，这需要通过下面的命令得到:

```
lib /machine:IX86 /def:.libs/libgmp-3.dll-def
cp libgmp-3.lib /my/inst/dir/lib
cp .libs/libgmp-3.dll-exp /my/inst/dir/lib/libgmp-3.exp
```

其中/my/inst/dir 是.lib 和.exp 文件安装目录。正如前面所说，MinGW 依赖于 C 运行时库 msvcrt.dll 处理 I/O，因此在使用 libgmp.dll 进行 VC 编程时需要用“cl /MD”进行编译。

第 二 章 GMP 基础

2.1 头文件与库文件

应用 GMP 所需的所有声明都集中在头文件‘gmp.h’中，它被设计为对 C 和 C++ 编译器都可用。

```
#include <gmp.h>
```

但是需要注意的是只有同时包含了头文件‘stdio.h’，原型包括 FILE* 参数的所有 GMP 函数才可用。

```
#include <stdio.h>
#include <gmp.h>
```

类似地，对原型包含 `va_list` 参数的函数如 `gmp_vprintf`，头文件 `<stdarg.h>`(或者 `<varargs.h>`) 是必须的；而对原型包含 `struct obstack` 参数的函数，头文件 `<obstack.h>` 也是必须的，例如当 `gmp_obstack_printf` 可用时。

所有使用 GMP 的程序都必须与“libgmp”库链接，在典型的 UNIX 类系统中，这通常用参数‘-lgmp’来完成，例如

```
gcc myprogram.c -lgmp
```

GMP C++ 函数存在于另外的库“libgmpxx”中，这个库在 C++ 支持开启时会建立 (参见上一章)。

```
g++ mycxxprog.cc -lgmpxx -lgmp
```

如果 GMP 不是安装在标准位置，你需要用‘-I’和‘-L’参数来指定头文件和库文件目录，例如

```
gcc myprogram.c -I/myincldir/gmp.h -L/mylibdir/libgmp.a
```

2.2 术语与类型

在 GMP 库的定义中，整数通常指多精度整数，它的 C 数据类型表示为 `mpz_t`，下面的例子显示怎样声明这样的整数：

```
mpz_t sum;
struct foo { mpz_t x, y; };
mpz_t vec[20];
```

有理数指的是多精度的分数，它的 C 类型表示为 `mpq_t`，例如，

```
mpq_t quotient;
```

浮点数或者简称实数，实际上是任意精度的有限定精度指数小数部分的数，这种对象的 C 类型表示为 `mpf_t`。

limb 指的是刚好能在机器字范围内的多精度数的一部分 (选择 *limb* 这个词是因为人体的一部分类似于一位，也许更多，包含几位)。通常一个 *limb* 表示 32 或 64bit，它的 C 类型表示为 `mp_limb_t`。

2.3 函数类

GMP 库中有六类函数：

1. 名字以 mpz_ 开头的有符号整数算术函数，相关的数据类型是 mpz_t，这一类中有大约 150 个函数。
2. 名字以 mpq_ 开头的有理数算术函数，相关的数据类型是 mpq_t，这一类中有大约 40 个函数，但是整数函数可以分别应用于对分子和分母的算术。
3. 名字以 mpf_ 开头的有符号整数算术函数，相关的数据类型是 mpf_t，这一类中有大约 60 个函数。
4. 与 Berkeley MP 相兼容的函数，例如 itom、madd 和 mult，相关的数据类型是 MINT。
5. 作用于自然数的快速低级函数，它们被用于前述的各类函数，对于时间要求极高的用户编程，也可以直接调用它们。这些函数以 mpn_ 开头，相关的数据类型是 mp_limb_t，这一类中有大约 30 个函数 (用起来有难度)。
6. 杂类函数，包括建立用户内存分配的函数和生成随机数的函数。

2.4 变量约定

GMP 函数一般先写输出变量后写输入变量，这个概念类似于赋值运算符。BSD MP 兼容函数是特例，它们都是后写输出的。

GMP 允许我们在一次调用中使用相同的输入和输出变量，例如作整数乘法的主函数 mpz_mul，可以计算 x 的平方，然后把输出放回到 x 中，

```
mpz_mul (x, x, x);
```

在对一个 GMP 变量赋值之前，需要通过调用一个特定的初始化函数对它进行初始化，当用完了一个 GMP 变量时，需要把它清除掉，你要调用一个实现这个目的的特定函数，具体用哪一个函数依赖于变量的类型，细节见于关于整数函数、有理数函数和实数函数的章节。

一个变量只能初始化一次，或者在下一次初始化之前对它进行清除。当一个变量初始化了以后，它可以被赋值任意多的次数。

为了实现更高的效率，应该尽量减少初始化和清除。通常在函数开始时进行初始化而在其结束时进行清除：

```
void
foo (void)
{
    mpz_t n;
    int i;
    mpz_init (n);
    for (i = 1; i < 100; i++)
    {
        mpz_mul (n, ...);
        mpz_fdiv_q (n, ...);
        ...
    }
    mpz_clear (n);
}
```

2.5 参数约定

当一个 GMP 变量作为函数参数时，通过引用进行调用是比较有效的，这个意思也就是说，如果函数函数储存了一个值那么它就可以在调用中改变它的原值。只作为输入的参数可以设计为 const 参数，这样在试图修改它的值时就可以引发编译器报错或者警告。

正如库函数所作的那样，当一个函数要返回 GMP 结果时，它应该指定其赋予的参数。还是像库函数那样，通过指定多于一个参数可以返回多个值。虽然本质上不是希望得到的，但是 mpz_t 等类型返回的往往是一个指针，而不是一个对象。

这里是一个接受 mpz_t 参数，进行计算，然后将它返回到指定参数的例子：

```
void
foo(mpz_t result, const mpz_t param, unsigned long n)
{
    unsigned long i;
    mpz_mul_ui (result, param, n);
    for (i = 1; i < n; i++)
    mpz_add_ui (result, result, i*7);
}
int main(void)
{
    mpz_t r, n;
    mpz_init (r);
    mpz_init_set_str (n, "123456", 0);
    foo (r, n, 20L);
    gmp_printf ("%Zd\n", r);
    return 0;
}
```

即使主函数为 parm 和 result 传递相同的值，正如库函数那样，函数 foo 也可以工作。但是有时候要使它正常运行可能会有些难于处理，所以在应用中最好不要为支持它而困扰。

有趣的是，GMP 类型 mpz_t 等被实现为特定结构的一元队列。这就是为什么声明一个变量 GMP 需要空间来建立对象，但是应用它作为参数时传递的是指向对象的指针。要注意的是，如果希望与未来的 GMP 版本相兼容，每个 mpz_t 等类型的变量所占用的实际空间，最好只被内部函数使用，而不应该直接被代码所使用。

2.6 内存管理

像 mpz_t 这类的 GMP 类型占用内存是很小的，它只包含由分配数据的长度和指针组成一对数。一旦 GMP 变量被初始化，GMP 承担所有的空间分配任务，每当变量没有足够的空间是 GMP 会分配额外的空间。

mpz_t 和 mpq_t 类型变量一定不会减少它所分配了的空间。通常这是最好的措施，因为这避免了频繁的重分配。那些需要在特定的点把内存返回到堆上的应用，可以使用函数 mpz_relloc2，或者清除不需要再使用的变量。

mpf_t 类型变量，在当前的实现中，使用固定数量的空间，这由选定的精度决定并在初始化时被分配，因此它们的长度是不变的。

所有的内存分配默认通过 malloc 和 friends 来完成，但是这也可以改变，详见第 13 章 [用户内存分配]。栈上的临时内存也可以被使用 (通过 alloc)，但是这可以在编译 GMP 时被改变 (见第一章)。

2.7 重入

GMP 是可重入和线程安全的，除了下面的例外：

- 如果编译 GMP 时用‘--enable-alloca=malloc-notreentrant’进行配置 (或者当 alloca 不可用时，用‘--enable-alloca=notreentrant’进行配置)，那么自然地 GMP 是不可重入的。
- mpf_set_default_prec 和 mpf_init 把选定的精度作为全局变量。可以使用 mpf_init2 作为替代，而在 C++ 接口中可以用复杂的精度来代替 mpf_class 构造函数。
- mpz_random 和其他的一些老的随机数函数使用了一个全局的随机状态，因而是不可重入的。新的随机数函数接受一个 gmp_randstate_t 参数作为替代。
- gmp_randinit(obsolete) 通过一个全局变量返回一个错误标志，这是非线程安全的。在应用中建议用 gmp_randinit_lc_2exp 作为替代。
- mp_set_memory_functions 使用全局变量储存选定的内存分配函数。
- 若内存分配函数通过 mp_set_memory_functions 来设置 (或者默认的 malloc 和 friends) 则是不可重入的，那么 GMP 也是不可重入的。
- 如果标准的 I/O 函数如 fwrite 是不可重入的，那么使用它们的 GMP 函数也是不可重入的。
- 两个线程同时读取同一个 GMP 变量是安全的，但是当一个在读而另一个在写时，它是不安全的，两个线程同时写也是不安全的。两个线程同时从相同的 gmp_randstate_t 产生随机数也是不安全的，因为这会牵涉到这个变量的更新。

2.8 有用的宏和常量

`const int mp_bits_per_limb` [全局常量]

每个 limb 的 bit 数。

`__GNU_MP_VERSION` [宏]
`__GNU_MP_VERSION_MINOR` [宏]
`__GNU_MP_VERSION_PATCHLEVEL` [宏]

三个整数分别表示主次 GMP 版本以及补丁级别。对 GMP i.j，这些数字分别是 i,j 和 0，对 GMP i.j.k，这些数字分别是 i,j 和 k。

`const char * const gmp_version` [全局常量]

GMP 版本数字，“i.j”和“i.j.k”这种形式，本版是“4.1.4”。

2.9 与其它版本的兼容

本版 GMP 是与所有 4.x 和 3.x 版本向上二进制兼容的，在源码级上与所有的 2.x 版本向上兼容，除了下面的例外：

- mpn_gcd 相对于 GMP3.0 发生了源参数的交换，与它有联系的其他函数也发生了改变。
- mpf_get_prec 计算精度与在 GMP3.0 和 GMP3.0.1 中有所不同，但是 3.1 版回复到 2.x 版的风格。

在 GMP 1 和 GMP 2 之间有很多的兼容性问题，当然把 GMP 1 的应用使用于 GMP 4 也会有这些问题。细节请参考 GMP 2 手册。

Berkeley MP 兼容库与标准的“libmp”是源码和二进制完全兼容的。

2.10 示例程序

在‘demos’子目录下有一些使用 GMP 的例子程序，这些程序不会编译和安装，但是有相应的“Makefile”文件来管理它们，例如，

```
make pexpr
./pexpr 68^975+10
```

下面的程序被提供：

- ‘pexpr’是一个表达式求值器，这个程序被应用于 GMP 的官方网页上。
- ‘calc’子目录下有一个相似但更简单的使用 `lex` 和 `yacc` 的求值器。
- ‘expr’子目录下还有另一个表达式求值器，并且设计了一个库以更好的应用于 C 程序中，更多的信息见‘demos/expr/READM’。
- ‘factorize’是一个 `Pollard-Rho` 整数分解程序。
- ‘isprime’是函数 `mpz_probab_prime_p` 的命令行接口程序。
- ‘primes’使用筛法计算和列举一定间隔内的素数。
- ‘qcn’是使用 `mpz_kronecker_ui` 估计平方数的一个例子。
- ‘perl’子目录是一个完整的 GMP 与 perl 接口，更多的信息见于‘demos/perl/INSTALL’。

2.11 效率

小操作数

在小操作数上，函数调用辅助操作和内存分配的时间可能比实际计算的时间还要显得重要。这在通用可变精度库中是不可避免的，尽管 GMP 试图设计成对大小操作数都是高效的。

静态链接

在一些 CPU 中，特别是在 X86CPU 中，静态库‘libgmp.a’可以用来获得最快的速度，因为 PIC 代码在共享‘libgmp.so’时每次函数调用和全局数据寻址可能需要很少的辅助操作。对很多程序来说这是无关紧要的，但是对于长计算来说却可以大大的获益。

初始化和清除

应该尽量的避免过多的初始化和清除，因为这是很耗时的，特别是与其它的快速操作如加法等相比之下。
一个语言解释器可能希望保存一个自由的初始化变量列表或栈，以供后续使用。它应该像带有一个垃圾收集器的解释器一样，也可以解释一些东西。

内存重分配

一个 `mpz_t` 或 `mpq_t` 类型的变量往往需要保存其后续增加的值，这就可能使得其内存需要分别重分配，依赖于不同的 C 标准库，这将是很慢的甚至可能产生内存碎片。假如应用可以估计最后的长度，那么就可以调用 `mpz_init2` 或者 `mpz_realloc2` 来从头开始分配所需空间。
如果 `mpz_init2` 或 `mpz_realloc2` 分配的空间太少，那也无需太在意，因为所有的函数如果必要的话会作更多的重分配。但是估计得严重的超出也可能会浪费内存。

2exp 函数

推荐在适当的时候应调用像 mpz_mul_2exp 这类的函数，通用函数如 mpz_mul 不会试图去辨别 2 的方幂或者其他特殊形式的数，因为这样的输入是极稀少的，每次都去测试将是非常浪费的。

ui 和 si 函数

针对小整数的 ui 和 si 函数是为了方便而存在的，可应用于适当的情况下。但是如果举例来说 mpz_t 包含了一个 unsigned long 范围内的数值时，没有必要把它从中摘取出来然后再调用 ui 函数，直接使用常规的 mpz 函数就可以了。

本地操作

mpz_abs，mpq_abs，mpf_abs，mpz_neg，mpq_neg，mpf_neg 用作本地操作是很快的，例如 mpz_abs(x,x)，因为在当前的实现中只有 x 的符号位需要改变。在合适的编译器 (如 GCC) 中这也是内联的。

mpz_add_ui，mpz_sub_ui，mpf_add_ui，mpf_sub_ui 也可以从本地操作中获益，例如 mpz_add_ui(x,x,y)，因为通常 x 只有一或二 limb 需要改变。类似的适用于 y 较小的情况下作全精度的 mpz_add 等。如果 y 较大时，放置于 cache 区可能会有帮助，但是也不过如此。

当前 mpz_mul 情况正好相反，分别的位置可能稍微好一些。除非 y 只是一 limb，调用 mpz_mul(x,x,y) 往往在形成结果之前需要临时存储 x 的一个拷贝。通常拷贝时间相对于乘法时间来说是很少的，所以没必要对它进行特别的关注。

mpz_set，mpq_set，mpq_set_num，mpf_set 等不会去辨别到自己的一个拷贝，所以像 mpz_set(x,x) 这样的调用是很浪费的，当然这不可能是故意这样写的，但是可能在两个指向同一对象的指针之间出现，因此避免它的一个测试是合理的：

```
if (x != y)
  mpz_set (x, y);
```

注意，任何时候本地操作引入额外的 mpz_set 调用都是不值得的，如果一个返回值要赋予某个特定的变量，你只需要直接赋给它，让 GMP 去处理数据的移动吧。

整除性测试

mpz_divisible_ui_p 和 mpz_congruent_ui_p 是测试一个 mpz_t 是否被一个小整数整除的最好函数，它应用了一个比 mpz_tdiv_ui 更快的算法，但是它不会提供关于余数的任何信息，只是判断特是否为 0(或者某个特定的值)。

但是测试被多个小整数的整除性时，最好是求模这些小整数的积的余数，这样可以节省多精度操作，例如测试一个数是否能被 23,29 或 31 中的任意一个整除，最好是先获得模 $23 \times 29 \times 31 = 20677$ 的余数，然后再测试它。

像同时给出商和余数的 mpz_tdiv_q_ui 一样的除法函数通常要比像 mpz_tdiv_ui 这样只给出余数的函数要慢一些。如果商不是很需要的话，可能最好就是只获得余数，然后到需要的时候再计算商 (如果余数是 0 的话可以使用 mpz_divexact_ui)。

有理数算术

mpq 函数作用于分子和分母没有公因子的 mpq_t 值上。有必要的话公因子会被检测并约去。通常，消除公因子是最好的措施，因为这样可以为后续的操作减小规模。

但是对于已知操作数的因式分解一些信息的应用，可以避免标准化时所需的一些 GCD 计算，或者取而代之以除法。例如被一个素数乘时，测试其分母的因子

就可以了，而不需要完全的 GCD 计算。又或是构造大乘积时可能已知几乎没有任何可以约去的，因此标准化可以留到最后来做。

`mpq_numref` 和 `mpq_denref` 宏为利用 `mpq` 以外的函数对分子和分母作一些操作提供了入口。

有理数的标准形式允许 `mpq_t` 与整数的混合类型加减法直接通过求分母的倍数来完成，这有时候比用 `mpq_add` 来的快一些，例如，

```
/* mpq increment */
mpz_add (mpq_numref(q), mpq_numref(q), mpq_denref(q));

/* mpq += unsigned long */
mpz_addmul_ui (mpq_numref(q), mpq_denref(q), 123UL);

/* mpq -= mpz */
mpz_submul (mpq_numref(q), mpq_denref(q), z);
```

整数序列

像 `mpz_fac_ui`, `mpz_fib_ui`, `mpz_bin_uiui` 这类函数被设计来计算单独的值。如果所需要的是一定范围内的数时，最好的办法就是调用它来达到一定的初始点，然后再反复的从那开始。

文本输入输出

在文本形式中十六进制和八进制被推荐使用。2 的方幂基可以比其他像 10 这样的基更有效的进行转换。对于大整数来说通常不会有特别的兴趣去看它的具体位，因此用什么基是无关紧要的。

也许可以期待有一天我们日常使用的就是八进制，正如瑞典的 King Charles XII 和其后的改革家所提出的那样。

2.12 其他编译链接相关内容 (略)

这部分涉及到 debug,profile,autoconf,emacs 等编译链接相关内容，这些内容需要对 GNU 软件的一些了解，这里略过去了，有兴趣的可以参考 [1]。

第 三 章　整数函数

这一章描述进行整数算术的 GMP 函数，这些函数以前缀 mpz_ 开头。
GMP 整数存储在 mpz_t 类型对象中。

3.1　初始化函数

整数算术函数假设所有的整数对象都是被初始化了的，这可以通过调用函数 mpz_init 来完成，例如，

```
{
    mpz_t integ;
    mpz_init (integeg);
    ...
    mpz_add (integ, ...);
    ...
    mpz_sub (integ, ...);
    /* Unless the program is about to exit, do ... */
    mpz_clear (integ);
}
```

可以看到，如果一个对象被初始化，你可以在其中存放新值任意多次。

`void mpz_init (mpz_t integer)` [函数]

初始化整数，并设其值为 0。

`void mpz_init2 (mpz_t integer, unsigned long n)` [函数]

初始化整数，开辟 n bits 空间，并设其值为 0。
n 只是初始空间，为了后续值的存储，有必要的话，integer 可以用通常的方法自动增长。如果预先知道一个最大长度值 mpz_init2 可以避免这样的内存重分配。

`void mpz_clear (mpz_t integer)` [函数]

释放 integer 所占据的空间，如果你已经用完了它们，应该对所有的 mpz_t 调用这个函数。

`void mpz_realloc2 (mpz_t integer, unsigned long n)` [函数]

把分配给 integer 的空间改为 n bits，如果能放下的话保持 integer 的值不变，否则设为 0。
这个函数可以增加变量的空间以避免频繁的自动重分配，或者减小它以回收堆空间。

```
void mpz_array_init  (mpz_t integer_array[],
                      size_t array_size, mp_size_t fixed_num_bits )
```
[函数]

这是一个特殊类型的初始化函数，Fixed_num_bits bits 的固定空间被分配给 integer_array 的 array_size 个整数中的每一个。
与 mpz_init 不同的是，分配了的空间是不会自动增长的，取而代之的是应用必须保证它足以存放任何值。下面的空间需求被应用于不同的函数，

- mpz_abs, mpz_neg, mpz_set, mpz_set_si, mpz_set_ui 需要它们存储值的空间。
- mpz_add, mpz_add_ui, mpz_sub, mpz_sub_ui 需要两个操作数中的大者加上额外的一 mp_bits_per_limb 的空间。
- mpz_mul, mpz_mul_ui, mpz_mul_ui 需要两个操作数 bits 之和的空间，但是每个操作数 bits 必须上升到 mp_bits_per_limb 的倍数 bits。
- mpz_swap 可用于两个数组变量之间，但是不可用于数组变量和普通变量之间。

对于其他函数，或者在有所怀疑的情况下，建议对常规的 mpz_init 变量进行计算，然后通过 mpz_set 把结果拷贝到数组变量中。
mpz_array_init 可以减少那些需要大整数数组的算法的内存消耗，因为它避免了小内存块的分配和重分配。没有函数可以释放由该函数分配的存储，注意不要调用 mpz_clear!

void * _mpz_realloc (mpz_t integer, mp_size_t new_alloc) [函数]

把 integer 的空间变成 new_alloc limbs，如果能放下的话保持 integer 的值不变，否则设为 0。返回值对应用是无用的，可以忽略。
mpz_realloc2 是达到这样的分配效果的更好办法，除了新长度以 limb 作为单位之外，mpz_realloc2 和 _mpz_realloc 没什么两样。

3.2 赋值函数

这些函数为已经初始化的整数赋新值。

void mpz_set (mpz_t rop, mpz_t op) [函数]
void mpz_set_ui (mpz_t rop, unsigned long int op) [函数]
void mpz_set_si (mpz_t rop, signed long int op) [函数]
void mpz_set_d (mpz_t rop, double op) [函数]
void mpz_set_q (mpz_t rop, mpq_t op) [函数]
void mpz_set_f (mpz_t rop, mpf_t op) [函数]

用 op 设置 rop 的值。
mpz_set_d, mpz_set_q, mpz_set_f 舍去 op 的小数部分使之为整数。

int mpz_set_str (mpz_t rop, char *str, int base) [函数]

用 str 设置 rop 的值，其中 str 是基为 base 的零结束的 C 字符串。空格允许出现在字符串中，它只会被简单的忽略掉。基可以是从 2 到 36 的整数值。如果基是 0，那么基由字符串的前几个字符决定：如果前两个字符是‘0x’或‘0X’，那么就认为是十六进制，而‘0’开头则认为是八进制，否则就认为是十进制。
如果整个字符串是基为 base 的有效数，则函数返回 0，否则返回 -1。

void mpz_swap (mpz_t rop1, mpz_t rop2) [函数]

有效的交换 rop1 和 rop2 的值。

3.3 初始化赋值组合函数

为了方便，GMP 提供了一个并行系列的初始化赋值函数，它们初始化输出并立即把值存于其中。这些函数的名字有 mpz_init_set... 这种形式，下面是应用一个这类函数的例子，

```
{
    mpz_t pie;
    mpz_init_set_str (pie,
        "3141592653589793238462643383279502884", 10);
    ...
    mpz_sub (pie, ...);
    ...
    mpz_clear (pie);
}
```

一旦整数被任意一个 mpz_init_set... 函数初始化，它可以用作普通整数函数的源或目的操作数。不要将初始化赋值函数应用于已初始化的变量！

```
void mpz_init_set (mpz_t rop, mpz_t op)                          [函数]
void mpz_init_set_ui (mpz_t rop, unsigned long int op)           [函数]
void mpz_init_set_si (mpz_t rop, signed long int op)             [函数]
void mpz_init_set_d (mpz_t rop, double op)                       [函数]
```

用 limb 空间对 rop 进行初始化，并用 op 设置其初值。

```
int mpz_init_set_str (mpz_t rop, char *str, int base)            [函数]
```

初始化 rop 并像 mpz_set_str 那样设置其值。如果字符串是正确的基 base 整数，那么函数返回 0，而当错误发生时返回 -1。即使有错误发生，rop 也会被初始化。(也就是说，必须对它调用 mpz_clear。)

3.4 转换函数

本节描述从 GMP 整数到标准 C 类型的转换函数。

```
unsigned long int mpz_get_ui (mpz_t op)                          [函数]
```

作为 unsigned long，返回 op 的值。
如果 op 太大，不能用 unsigned long 表示，那么只取其能用 unsigned long 表示的最低位，op 的符号会被忽略，而只用它的绝对值。

```
signed long int mpz_get_si (mpz_t op)                            [函数]
```

若 op 能用 signed long int 表示，则返回 op 的值，否则返回 op 的最低部分，符号与 op 相同。
如果 op 太大不能用 signed long int 表示，返回的结果可能不是很有用。要判断值是否在 signed long int 范围内，使用函数 mpz_fits_slong_p。

```
double mpz_get_d (mpz_t op)                                      [函数]
```

将 op 转换成双精度浮点数。

```
double mpz_get_d_2exp (signed long int *exp, mpz_t op)           [函数]
```

寻找 d 和 exp 使得 $d \times 2^{exp}$ 较好的近似 op，其中 $0.5 \leq |d| < 1$。

`char * mpz_get_str (char *str, int base, mpz_t op)` [函数]

将 op 转换成 base 进制表示下的字符串，其中 base 是从 2 到 36 的整数。
如果 str 为 NULL，则结果字符串用当前的内存分配函数分配内存，所需内存块大小为 strlen(str)+1 字节，刚好够存储字符串以及 0 结束符。如果 str 非 NULL，它应该指向足够存储结果的内存块，即 mpz_sizeinbase (op, base) + 2，额外的 2 字节存储可能的负符号和 0 结束符。
最后返回结果字符串的指针，指向分配的内存块，或者给定的 str。

`mp_limb_t mpz_getlimbn (mpz_t op, mp_size_t n)` [函数]

返回 op 的第 n limb，符号会被忽略，只考虑绝对值，最低 limb 是第 0limb。
用 mpz_size 可以知道 op 由多少 limb 组成，如果 n 在 0 到 mpz_size(op)-1 范围之外，则 mpz_getlimbn 返回 0。

3.5 算术函数

`void mpz_add (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]
`void mpz_add_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` [函数]

置 rop 为 op1+op2。

`void mpz_sub (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]
`void mpz_sub_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` [函数]
`void mpz_ui_sub (mpz_t rop, unsigned long int op1, mpz_t op2)` [函数]

置 rop 为 op1-op2。

`void mpz_mul (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]
`void mpz_mul_si (mpz_t rop, mpz_t op1, long int op2)` [函数]
`void mpz_mul_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` [函数]

置 rop 为 op1×op2。

`void mpz_addmul (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]
`void mpz_addmul_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` [函数]

置 rop 为 rop+op1×op2。

`void mpz_submul (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]
`void mpz_submul_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` [函数]

置 rop 为 rop-op1×op2。

`void mpz_mul_2exp (mpz_t rop, mpz_t op1, unsigned long int op2)` [函数]

置 rop 为 $op1 \times 2^{op2}$，这个操作也就是左移 $op2$ 比特。

`void mpz_neg (mpz_t rop, mpz_t op)` [函数]

置 rop 为 -op。

`void mpz_abs (mpz_t rop, mpz_t op)` [函数]

置 rop 为 op 的绝对值。

3.6 除法函数

如果除数为 0，那么除法是没有意义的，向除法或者模函数 (包括模指数函数 mpz_powm 和 mpz_powm_ui) 传递一个 0 除数，将会引起被 0 除的异常。这就使得程序需要像处理普通 C int 算术一样处理这些函数中的算术异常。

```
void mpz_cdiv_q (mpz_t q, mpz_t n, mpz_t d)                                    [函数]
void mpz_cdiv_r (mpz_t r, mpz_t n, mpz_t d)                                    [函数]
void mpz_cdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)                          [函数]
unsigned long int mpz_cdiv_q_ui (mpz_t q,  mpz_t n,                            [函数]
                                 unsigned long int d)
unsigned long int mpz_cdiv_r_ui (mpz_t r,  mpz_t n,                            [函数]
                                 unsigned long int d)
unsigned long int mpz_cdiv_qr_ui (mpz_t q,  mpz_t r, mpz_t n,                  [函数]
                                  unsigned long int d)
unsigned long int mpz_cdiv_ui (mpz_t n, unsigned long int d)                   [函数]
void mpz_cdiv_q_2exp (mpz_t q, mpz_t n, unsigned long int b)                   [函数]
void mpz_cdiv_r_2exp (mpz_t r, mpz_t n, unsigned long int b)                   [函数]
void mpz_fdiv_q (mpz_t q, mpz_t n, mpz_t d)                                    [函数]
void mpz_fdiv_r (mpz_t r, mpz_t n, mpz_t d)                                    [函数]
void mpz_fdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)                          [函数]
unsigned long int mpz_fdiv_q_ui (mpz_t q,  mpz_t n,                            [函数]
                                 unsigned long int d)
unsigned long int mpz_fdiv_r_ui (mpz_t r,  mpz_t n,                            [函数]
                                 unsigned long int d)
unsigned long int mpz_fdiv_qr_ui (mpz_t q,  mpz_t r, mpz_t n,                  [函数]
                                  unsigned long int d)
unsigned long int mpz_fdiv_ui (mpz_t n, unsigned long int d)                   [函数]
void mpz_fdiv_q_2exp (mpz_t q, mpz_t n, unsigned long int b)                   [函数]
void mpz_fdiv_r_2exp (mpz_t r, mpz_t n, unsigned long int b)                   [函数]
void mpz_tdiv_q (mpz_t q, mpz_t n, mpz_t d)                                    [函数]
void mpz_tdiv_r (mpz_t r, mpz_t n, mpz_t d)                                    [函数]
void mpz_tdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)                          [函数]
unsigned long int mpz_tdiv_q_ui (mpz_t q,  mpz_t n,                            [函数]
                                 unsigned long int d)
unsigned long int mpz_tdiv_r_ui (mpz_t r,  mpz_t n,                            [函数]
                                 unsigned long int d)
unsigned long int mpz_tdiv_qr_ui (mpz_t q,  mpz_t r, mpz_t n,                  [函数]
                                  unsigned long int d)
unsigned long int mpz_tdiv_ui (mpz_t n, unsigned long int d)                   [函数]
void mpz_tdiv_q_2exp (mpz_t q, mpz_t n, unsigned long int b)                   [函数]
void mpz_tdiv_r_2exp (mpz_t r, mpz_t n, unsigned long int b)                   [函数]
```

用 d 除 n，得到商 q 与/或余数 r。对于 2exp 函数，$d = 2^b$。根据应用不同，有三种舍入方式，

- cdiv 中 q 向上趋近于 $+\infty$ 舍入，r 有与 d 相反的符号，c 表示“ceil”。
- fdiv 中 q 向下趋近于 $-\infty$ 舍入，r 有与 d 相同的符号，f 表示“floor”。
- tdiv 中 q 趋近于 0 舍入，r 有与 d 相同的符号，t 表示“truncate”。

在所有情况下 q 和 r 都满足 $n = qd + r$，其中 r 还满足 $0 \le |r| < |d|$。

`q` 函数只计算商 q，`r` 函数只计算余数 r，`qr` 函数同时计算商 q 和余数 r，需要注意的是，对于 `qr` 函数不可以为 q 和 r 传递相同的变量，否则结果是不可预料的。

`ui` 函数的返回值是余数，实际上所有的 `div_ui` 函数都返回余数。因为 `tdiv` 和 `cdiv` 的余数可能是负的，所以这些函数的返回值是余数的绝对值。

`2exp` 函数做的是右移或者位屏蔽，但是当然也像其他函数一样进行舍入。对于正的 n `mpz_fdiv_q_2exp` 和 `mpz_tdiv_q_2exp` 都是简单位的右移。对于负的 n，`mpz_fdiv_q_2exp` 像位逻辑函数一样把 n 看作它的 2 的补码，然后进行算术右移，而 `mpz_tdiv_q_2exp` 把 n 看作符号和数值。

`void mpz_mod (mpz_t r, mpz_t n, mpz_t d)` [函数]
`unsigned long int mpz_mod_ui (mpz_t r, mpz_t n, unsigned long int d)` [函数]

置 r 为 $n \mod d$。除数的符号被忽略，结果总是非负的。

`mpz_mod_ui` 等价于上面的 `mpz_fdiv_r_ui`，在置 r 的同时返回余数。如果只想得到返回值可见上面的 `mpz_fdiv_ui`。

`void mpz_divexact (mpz_t q, mpz_t n, mpz_t d)` [函数]
`void mpz_divexact_ui (mpz_t q, mpz_t n, unsigned long d)` [函数]

置 q 为 n/d。只有预先知道 d 是整除 n 的，这些函数才返回正确的结果。

这里使用的程序比其他的除法函数的快得多，当确知整除发生时它是最好的选择，例如化简有理数为最小项时。

`int mpz_divisible_p (mpz_t n, mpz_t d)` [函数]
`int mpz_divisible_ui_p (mpz_t n, unsigned long int d)` [函数]
`int mpz_divisible_2exp_p (mpz_t n, unsigned long int b)` [函数]

如果 n 被 d 整除或者在 `mpz_divisible_2exp_p` 中被 2^d 整除，那么返回非 0 值。

`int mpz_congruent_p (mpz_t n, mpz_t c, mpz_t d)` [函数]
`int mpz_congruent_ui_p(mpz_t n,unsigned long int c,unsigned long int d)` [函数]
`int mpz_congruent_2exp_p (mpz_t n, mpz_t c, unsigned long int b)` [函数]

如果 n 模 d 同余 c 或者在 `mpz_congruent_2exp_p` 中模 2^d 同余 c，那么返回非 0 值。

3.7 指数函数

`void mpz_powm (mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)` [函数]
`void mpz_powm_ui(mpz_t rop,mpz_t base,unsigned long int exp,mpz_t mod)` [函数]

置 `rop` 为 $base^{exp} \mod mod$。

如果逆 $base^{-1} \mod mod$ 存在，那么负的 exp 可以被支持。若逆不存在，而产生被 0 除错误。

`void mpz_pow_ui (mpz_t rop, mpz_t base, unsigned long int exp)` [函数]
`void mpz_ui_pow_ui (mpz_t rop, unsigned long int base, unsigned long int exp)` [函数]

置 `rop` 为 $base^{exp}$，0^0 返回 1。

3.8 求根开方函数

`int mpz_root (mpz_t rop, mpz_t op, unsigned long int n)` [函数]

置 `rop` 为 $\lfloor \sqrt[n]{op} \rfloor$，即截取 *op* 的 *n* 次根整数部分。如果计算是整的，也就是说 *op* 恰是 *rop* 的 *n* 次方，那么返回非 0 值。

`void mpz_sqrt (mpz_t rop, mpz_t op)` [函数]

置 `rop` 为 $\lfloor \sqrt{op} \rfloor$，即截取 *op* 的 2 次根整数部分。

`void mpz_sqrtrem (mpz_t rop1, mpz_t rop2, mpz_t op)` [函数]

像 `mpz_sqrt` 一样，置 `rop1` 为 $\lfloor \sqrt{op} \rfloor$，置 *rop*2 为余数 $(op - rop1^2)$，如果 *op* 是完全平方数，那么 *rop*2 置为 0。

如果 *rop*1 和 *rop*2 传递的是相同的变量，那么结果是无意义的。

`int mpz_perfect_power_p (mpz_t op)` [函数]

如果 *op* 是完全幂数，也就是说存在整数 a, b，其中 $b > 1$，使得 $op = a^b$，那么返回非 0 值。

在这个定义之下，0 和 1 都是完全幂数。负值的 *op* 也可以被接受，但只能是奇的完全幂数。

`int mpz_perfect_square_p (mpz_t op)` [函数]

如果 *op* 是完全平方数，也就是说 *op* 的二次根是整数，那么返回非 0 值。在这个定义之下，0 和 1 都是完全平方数。

3.9 数论函数

`int mpz_probab_prime_p (mpz_t n, int reps)` [函数]

判断 *n* 是否为素数，若 *n* 确定是素数则返回 2，如果 *n* 是概率素数 (不能完全确定) 那么返回 1，或者如果 *n* 确定是合数那么返回 0。

这个函数首先做一些试除，然后再作 Miller-Rabin 概率素性判别。*reps* 控制这样的判别做多少次，5 到 10 是较合理的数值，更多次的判别可以减小合数被返回为概率素数的可能。

调用 Miller-Rabin 判别和其他相似的判别组成的复合判别可能更为合理。判别失败的数可以知道是合数，但是通过判别的数可以是素数也可能是合数。只有极少数的合数可以通过判别，因此通过判别的数一般都被认为是素数。

`void mpz_nextprime (mpz_t rop, mpz_t op)` [函数]

置 *rop* 为比 *op* 大的下一个素数。

这个函数使用一个概率算法来判别素数。对于实际的使用来说它是足够的，合数通过判别的可能性极小。

`void mpz_gcd (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]

置 *rop* 为 *op*1 和 *op*2 的最大公因子。结果总是正的，即使一个或多个输入是负数也是如此。

`unsigned long int mpz_gcd_ui(mpz_t rop,mpz_t op1,unsigned long int op2)` [函数]

计算 *op*1 和 *op*2 的最大公因子，如果 *rop* 非 NULL，那么将结果存于其中。

若结果小到能存于一个 unsinged long int 中，则返回之。否则，返回 0，结果等于 *op*1。注意，如果 *op*2 非 0，结果总是能存于一个 unsinged long int 中的。

`void mpz_gcdext (mpz_t g, mpz_t s, mpz_t t, mpz_t a, mpz_t b)` [函数]

置 g 为 a 和 b 的最大公因子，同时计算 s 和 t 满足 $as + bt = g$，其中 g 总是正的，即使 a 或 b 为负的也是如此。

如果 t 为 NULL，那么不计算其值。

`void mpz_lcm (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]
`void mpz_lcm_ui (mpz_t rop, mpz_t op1, unsigned long op2)` [函数]

置 *rop* 为 *op*1 和 *op*2 的最小公倍数，不管 *op*1 和 *op*2 的符号如何，*rop* 总是正的，若 *op*1 或 *op*2 为 0 则返回 *rop* 为 0。

`int mpz_invert (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]

计算 *op*1 模 *op*2 的逆，并将结果置于 *rop*。如果逆存在则返回非 0 值，且 *rop* 满足 $0 \le rop < op2$。如果逆不存在，那么返回 0，此时 *rop* 是无意义的。

`int mpz_jacobi (mpz_t a, mpz_t b)` [函数]

计算 Jacobi 符号 $(\frac{a}{b})$。只有当 b 为奇数时此函数才有定义。

`int mpz_legendre (mpz_t a, mpz_t p)` [函数]

计算 Legendre 符号 $(\frac{a}{p})$。只有当 p 为正奇素数时才有定义，对于这样的 p 它等价于 Jacobi 符号。

`int mpz_kronecker (mpz_t a, mpz_t b)` [函数]
`int mpz_kronecker_si (mpz_t a, long b)` [函数]
`int mpz_kronecker_ui (mpz_t a, unsigned long b)` [函数]
`int mpz_si_kronecker (long a, mpz_t b)` [函数]
`int mpz_ui_kronecker (unsigned long a, mpz_t b)` [函数]

利用 Kronecker 扩展计算 Jacobi 符号，Kronecker 扩展是：若 a 为奇数则 $(\frac{2}{a}) = (\frac{a}{2})$，而若 a 为偶数则 $(\frac{a}{2}) = 0$。

当 b 为奇数时，Jacobi 符号与 Kronecker 符号是等价的，所以 mpz_kronecker_ui 等也可以用于复合精度 Jacobi 符号计算。

更多的信息可见于任意的数论课本，或者参看例程‘demos/qcn.c’，其中用到了 mpz_kronecker_ui。

`unsigned long int mpz_remove (mpz_t rop, mpz_t op, mpz_t f)` [函数]

约去 *op* 的所有 f 因子，并将结果储存于 *rop*。返回的结果是多少个 f 因子被约去。

`void mpz_fac_ui (mpz_t rop, unsigned long int op)` [函数]

置 *rop* 为 *op*!，*op* 的阶乘。

`void mpz_bin_ui (mpz_t rop, mpz_t n, unsigned long int k)` [函数]
`void mpz_bin_uiui (mpz_t rop, unsigned long int n, unsigned long int k)` [函数]

计算二项式系数 $\binom{n}{k}$，并将结果储存于 *rop* 中。利用等式 $\binom{-n}{k} = (-1)^k \binom{n+k-1}{k}$，mpz_bin_ui 可支持负值的 n。

`void mpz_fib_ui (mpz_t fn, unsigned long int n)` [函数]
`void mpz_fib2_ui (mpz_t fn, mpz_t fnsub1, unsigned long int n)` [函数]

`mpz_fib_ui` 置 fn 为第 n 个 Fibonacci 数 F_n。`mpz_fib2_ui` 置 fn 为 F_n，并置 $fnsub1$ 为 F_{n-1}。

这两个函数被设计为计算单独的 Fibonacci 数。如果希望得到一系列的 Fibonacci 数，那么最好先应用 `mpz_fib2_ui`，然后再利用递推公式 $F_{n+1} = F_n + F_{n-1}$ 或相似的公式进行迭代。

`void mpz_lucnum_ui (mpz_t ln, unsigned long int n)` [函数]
`void mpz_lucnum2_ui (mpz_t ln, mpz_t lnsub1, unsigned long int n)` [函数]

`mpz_lucnum_ui` 置 ln 为第 n 个 Lucas 数 L_n。`mpz_lucnum2_ui` 置 ln 为 L_n，并置 $lnsub1$ 为 L_{n-1}。

这两个函数被设计为计算单独的 Lucas 数。如果希望得到一系列的 Lucas 数，那么最好先应用 `mpz_lucnum2_ui`，然后再利用递推公式 $L_{n+1} = L_n + L_{n-1}$ 或相似的公式进行迭代。

Fibonacci 数与 Lucas 数是相关的序列，所以 `mpz_fib2_ui` 和 `mpz_lucnum2_ui` 不必同时调用，具体的从 Fibonacci 数到 Lucas 数的公式可参考数论课本。

3.10 比较函数

`int mpz_cmp (mpz_t op1, mpz_t op2)` [函数]
`int mpz_cmp_d (mpz_t op1, double op2)` [函数]
`int mpz_cmp_si (mpz_t op1, signed long int op2)` [宏]
`int mpz_cmp_ui (mpz_t op1, unsigned long int op2)` [宏]

比较 $op1$ 和 $op2$。若 $op1 > op2$ 则返回正值，若 $op1 = op2$ 则返回 0，若 $op1 < op2$ 则返回负值。

需要注意的是 `mpz_cmp_ui` 和 `mpz_cmp_si` 是宏，它们可能对参数估值一次以上。

`int mpz_cmpabs (mpz_t op1, mpz_t op2)` [函数]
`int mpz_cmpabs_d (mpz_t op1, double op2)` [函数]
`int mpz_cmpabs_ui (mpz_t op1, unsigned long int op2)` [函数]

比较 $op1$ 和 $op2$ 的绝对值。若 $|op1| > |op2|$ 则返回正值，若 $|op1| = |op2|$ 则返回 0，若 $|op1| < |op2|$ 则返回负值。

`int mpz_sgn (mpz_t op)` [函数]

若 op 为正则返回 +1，0 则返回 0，负则返回 -1。

这个函数实际上用宏来实现，它要对它的参数估值多次。

3.11 逻辑和位操作函数

这些函数像使用二的补码算术一样进行操作 (即使实际实现的是有符号数)。最低有效比特是第 0 位。

`void mpz_and (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]

置 rop 为 $op1$ 位与 $op2$。

`void mpz_ior (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]

置 *rop* 为 *op*1 位同或 *op*2。

`void mpz_xor (mpz_t rop, mpz_t op1, mpz_t op2)` [函数]

置 *rop* 为 *op*1 位异或 *op*2。

`void mpz_com (mpz_t rop, mpz_t op)` [函数]

置 *rop* 为 *op* 的 1 的补码。

`unsigned long int mpz_popcount (mpz_t op)` [函数]

如果 $op \geq 0$，返回 *op* 的重量，也就是 *op* 的二进制表示中 1 的个数。如果 $op < 0$，那么 1 的个数是无限的，返回 `unsigned long` 可能的最大值 `ULONG_MAX`。

`unsigned long int mpz_hamdist (mpz_t op1, mpz_t op2)` [函数]

如果 *op*1 和 *op*2 都 ≥ 0 或者都 < 0，返回这两个操作数之间的海明距离，也就是它们之间不同 bit 位的个数。如果其中一个 ≥ 0 而另一个 < 0，那么不同 bit 位的个数是无限的，返回 `unsigned long` 可能的最大值 `ULONG_MAX`。

`unsigned long int mpz_scan0 (mpz_t op, unsigned long int starting_bit)` [函数]
`unsigned long int mpz_scan1 (mpz_t op, unsigned long int starting_bit)` [函数]

从 `starting_bit` 到更高位对 *op* 进行搜索，直到第一个 0 或 1(分别) 被找到。返回找到位的索引。

如果 `starting_bit` 就有所寻找的位，则返回 `starting_bit`。

如果所要 bit 没有找到，那么返回 `ULONG_MAX`。这种情况往往发生在当 `mpz_scan0` 通过了正数的顶端，或者当 `mpz_scan1` 通过了负数的顶端时。

`void mpz_setbit (mpz_t rop, unsigned long int bit_index)` [函数]

置 *rop* 的 `bit_index` 位为 1。

`void mpz_clrbit (mpz_t rop, unsigned long int bit_index)` [函数]

置 *rop* 的 `bit_index` 位为 0。

`int mpz_tstbit (mpz_t op, unsigned long int bit_index)` [函数]

测试 *op* 的 `bit_index` 位，据此返回 0 或 1。

3.12 输入输出函数

处理来自标准输入输出流 `stdio` 的输入和输出的函数。传递 `NULL` 参数给这些函数中的任意一个都可能使得它们分别从 `stdin` 和 `stdout` 读和写。
使用这些函数时最好在‘gmp.h’之前定义‘stdio.h’，这样可允许‘gmp.h’为这些函数定义原型。

`size_t mpz_out_str (FILE *stream, int base, mpz_t op)` [函数]

将 *op* 作为基为 *base* 的数字串输出于标准输入输出流 `stream` 中，其中基可以从 2 到 36。

返回所写的字节数，如果有错误发生，那么返回 0。

`size_t mpz_inp_str (mpz_t rop, FILE *stream, int base)` [函数]

从标准输入输出流 stream 中，输入可能以空格开头的基为 *base* 的字符串，并将可读整数存于 *rop* 中，其中基可以从 2 到 36。如果基是 0，那么基由字符串的前几个字符决定：如果前两个字符是‘0x’或‘0X’，那么就认为是十六进制，而‘0’开头则认为是八进制，否则就认为是十进制。

返回所读的字节数，如果有错误发生，那么返回 0。

`size_t mpz_out_raw (FILE *stream, mpz_t op)` [函数]

以原始数据二进制存储格式，将 *op* 输出于标准输入输出流 stream 中。整数以便携式结构书写，也就是 4 字节的长度信息加上若干字节的 limbs。长度和 limbs 都以递减序书写 (也就是大结尾)。

这里的输出可以被 mpz_inp_raw 读取。

返回所写的字节数，如果有错误发生，那么返回 0。

从 GMP1 开始，除了因为为了 32 位和 64 位机之间的兼容性而作的必要修改之外，这个输出可以被 mpz_inp_raw 读取。

`size_t mpz_inp_raw (mpz_t rop, FILE *stream)` [函数]

以 mpz_out_raw 所写的格式那样，从标准输入输出流 stream 中进行输入，并将结果置于 *rop* 中。返回所读的字节数，如果有错误发生，那么返回 0。

从 GMP1 开始，这个程序也可以从 mpz_out_raw 的输出中进行读取，除了因为为了 32 位和 64 位机之间的兼容性而作的必要修改之外。

3.13 随机数函数

GMP 的随机数函数由两类函数组成，一类是老的依赖全局状态的随机数函数，另一类是新的接收可读可变的状态参数的随机数函数。

`void mpz_urandomb(mpz_t rop,gmp_randstate_t state,unsigned long int n)` [函数]

产生一个均匀分布在 0 到 2^n-1(包含) 范围内的随机整数。

在激活这个函数之前必须先调用某个 gmp_randinit 函数对变量 *state* 进行初始化。

`void mpz_urandomm (mpz_t rop, gmp_randstate_t state, mpz_t n)` [函数]

产生一个均匀分布在 0 到 $n-1$(包含) 范围内的随机整数。

在激活这个函数之前必须先调用某个 gmp_randinit 函数对变量 *state* 进行初始化。

`void mpz_rrandomb(mpz_t rop,gmp_randstate_t state,unsigned long int n)` [函数]

产生一个随机整数，其二进制表示由 01 长串组成。这种随机数对测试函数和算法很有用，因为这类随机数被证明更容易引发边界值 bug。这个随机数将在 0 到 $n-1$(包含) 范围内。

在激活这个函数之前必须先调用某个 gmp_randinit 函数对变量 *state* 进行初始化。

`void mpz_random (mpz_t rop, mp_size_t max_size)` [函数]

产生一个至多 max_sizelimbs 的随机整数。所产生的随机数不满足任意特别的随机性要求。若 max_size 为负数，则产生负的随机数。

这个函数很陈旧，建议用 mpz_urandomb 或者 mpz_urandomm 代替。

`void mpz_random2 (mpz_t rop, mp_size_t max_size)` [函数]

产生一个至多 max_sizelimbs 的随机整数，其二进制表示由 01 长串组成。这种随机数对测试函数和算法很有用，因为这类随机数被证明更容易引发边界值 bug。若 max_size 为负数，则产生负的随机数。

这个函数很陈旧，建议用 mpz_rrandomb 代替。

3.14 整数引入和导出

通过下面的函数 mpz_t 变量可以转换到或转换自任意字组成的二进数据。

```
void mpz_import (mpz_t rop,  size_t count, int order, int size,
                             int endian, size_t nails, const void *op)
```
[函数]

用 *op* 中的字数组，对 *rop* 进行赋值。

参数用以指定数据格式。*count* 个字被读取，每个字 *size* 字节。*order* 可以是 1 表示最高字优先，或者 -1 表示最低字优先。在每个字中，*endian* 可以是 1 表示最高字节优先，-1 表示最低字节优先，或者 0 表示使用主机 CPU 的结尾顺序。每个字的最高 *nail* 比特会被忽略，若 *nail* 是 0，则可使用全字。

符号不会从数据中取出，*rop* 只会是一个正数。应用可以自己处理符号，例如使用函数 mpz_neg。

对于 *op* 没有任何存储约束，任何地址都可以。

这里是转换一个 unsigned long 数组数据的例子，最高元素优先，在每个值中使用主机的字节顺序。

```
unsigned long a[20];
mpz_t z;
mpz_import (z, 20, 1, sizeof(a[0]), 0, 0, a);
```

这个例子假定使用给定类型全 sizeof 字节数据，这通常都是正确的，特别是众所周知对 unsigned long 在各种情形下这肯定是正确的。但是需要注意的是，在 Cray 向量系统中，short 和 int 往往都是存储于 8 字节 (通过 sizeof 可知) 中，但是用到的只是 32 和 46 比特，*nail* 量可以通过这个计算出来，例如传递 8*sizeof(int)-INT_BIT 与之。

```
void * mpz_export (void *rop,  size_t *countp, int order, int size,
                               int endian, size t nails, mpz t op)
```
[函数]

用 *op* 的数据填充 *rop*。

参数控制数据产生的格式。每个字 *size* 字节，*order* 可以是 1 表示最高字优先，或者 -1 表示最低字优先。在每个字中，*endian* 可以是 1 表示最高字节优先，-1 表示最低字节优先，或者 0 表示使用主机 CPU 的结尾顺序。每个字的最高 *nail* 比特会被忽略，若 *nail* 是 0，则可使用全字。

产生的字数写入 *countp 中，或者 *countp 也可以是 NULL，此时丢弃字数。*rop* 必须有足够的空间以存放数据，或者如果 *rop* 为 NULL，那么将使用当前的内存分配函数分配必要长度的结果数组，在两种情形下，返回值为 *rop* 或者分配的内存块，它只能作为目的操作数使用。

如果 *op* 非 0，那么产生的最高字也是非 0 的。如果 *op* 为 0，那么返回的字数为 0，并且不写 *rop*。在这种情况下，如果 *rop* 为 NULL，那么不分配内存，只是返回 NULL。

op 的符号会被忽略，只有其绝对值会被导出。应用可以通过 mpz_sgn 获得符号，并自由处理它。

对于 *rop* 没有任何存储约束，任何地址都可以。

3.15 杂类函数

`int mpz_fits_ulong_p (mpz_t op)` [函数]
`int mpz_fits_slong_p (mpz_t op)` [函数]
`int mpz_fits_uint_p (mpz_t op)` [函数]
`int mpz_fits_sint_p (mpz_t op)` [函数]
`int mpz_fits_ushort_p (mpz_t op)` [函数]
`int mpz_fits_sshort_p (mpz_t op)` [函数]

当且仅当 *op* 分别在 unsigned long int, signed long int, unsigned int, signed int, unsigned short int 或 signed short int 范围内，函数返回非 0，否则返回 0。

`int mpz_odd_p (mpz_t op)` [宏]
`int mpz_even_p (mpz_t op)` [宏]

分别确定 *op* 为奇数或偶数。若是，返回非 0，否则返回 0。这两个宏对参数估值一次以上。

`size_t mpz_size (mpz_t op)` [函数]

以 `limb` 为单位，返回 *op* 的长度。若 *op* 为 0，则返回值亦为 0。

`size_t mpz_sizeinbase (mpz_t op, int base)` [函数]

返回 *op* 在给定的基 *base* 下的位数，其中 *base* 可以从 2 到 36。*op* 的符号会被忽略，只有绝对值会被使用。结果可能是正好的，也可能比实际值大 1。当 *base* 为 2 的方幂时，结果总是正好的。当 *op* 为 0 时，结果总是 1。

这个函数可以用来确定将 *op* 转换成字符串所需要的空间。正确的空间量通常应该比 `mpz_sizeinbas` 的返回值大 2，额外的一个字节表示可能的负号，另一个表示 0 结束符。

可以注意到，`mpz_sizeinbase(op,2)` 可以用来确定 *op* 的最高 1 比特，计算从 1 开始。(不像位操作函数那样从 0 开始)

当应用要为自己分配空间时，所需长度可以通过下面这样的计算来确定。因为 `mpz_sizeinbase` 总是至少返回 1，这里 *count* 也将至少为 1，这就避免了碰到 `malloc(0)` 问题的可能性，尽管如果 `z` 为 0，那么根本不需要分配 (或写) 空间。

```
numb = 8*size - nail;
count = (mpz_sizeinbase (z, 2) + numb-1) / numb;
p = malloc (count * size);
```

第 四 章 有理数函数

这一章介绍执行有理数算术的 GMP 函数，这些函数都有前缀 mpq_。
有理数存储于 mpq_t 类型对象中。
所有的有理数函数都假设操作数具有规范形，并且规范化其结果。规范形的意思是分子和分母没有公因子，并且分母是正的。0 具有唯一的表示 0/1。
纯赋值函数不会规范化赋值变量，在对这些变量进行算术运算之前，对它们进行规范化，这是用户自己的责任。

`void mpq_canonicalize (mpq_t op)` [函数]

消去 *op* 分子和分母的公因子，并使分母为正。

4.1 初始化和赋值函数

`void mpq_init (mpq_t dest_rational)` [函数]

用 0/1 对 dest_rational 进行初始化，每个变量应该只被初始化一次，或者在下次初始化之前作清除 (使用函数 mpq_clear)。

`void mpq_clear (mpq_t rational_number)` [函数]

释放 rational_number 所占的空间。应该确保在使用完一个 mpq_t 变量之后都调用这个函数以清除之。

`void mpq_set (mpq_t rop, mpq_t op)` [函数]
`void mpq_set_z (mpq_t rop, mpz_t op)` [函数]

用 *op* 对 *rop* 赋值。

`void mpq_set_ui (mpq_t rop, unsigned long int op1, unsigned long int op2)` [函数]
`void mpq_set_si (mpq_t rop, signed long int op1, unsigned long int op2)` [函数]

置 *rop* 为 *op*1/*op*2。需要注意的是，如果 *op*1 与 *op*2 具有公因子，那么在对 *rop* 进行任意运算之前，应该用 mpq_canonicalize 对 *rop* 进行处理。

`int mpq_set_str (mpq_t rop, char *str, int base)` [函数]

用给定基 *base* 下的 0 结束字符串对 *rop* 进行赋值。

字符串可以是整数如‘41’或分数‘41/152’，分数必须是规范形的，否则必须调用 mpq_canonicalize 对其进行处理。

分子和可能的分母像在函数 mpz_set_str 中一样进行解析。在字符串中允许有空格，它只是简单的被忽略。基 *base* 可以从 2 到 36，或者为 0，此时首字符作为参考：0x 表示十六进制，0 表示八进制，其他则是十进制。注意对分子和分母这个判断是分别来做的，例如 0xEF/100 表示 239/100，而 0xEF/0x100 表示 239/256。

若整个字符串都是有效数字则返回 0，否则返回 -1。

`void mpq_swap (mpq_t rop1, mpq_t rop2)` [函数]

有效的交换 *rop*1 和 *rop*2 的值。

4.2 转换函数

`double mpq_get_d (mpq_t op)` [函数]

将 *op* 转换为双精度浮点数。

`void mpq_set_d (mpq_t rop, double op)` [函数]
`void mpq_set_f (mpq_t rop, mpf_t op)` [函数]

置 *rop* 为 *op* 的值，不需要舍入。

`char * mpq_get_str (char *str, int base, mpq_t op)` [函数]

转换 *op* 成基为 *base* 的数字串。基可以在 2 到 36 范围内变化。字符串将是‘num/den’的形式，或者若分母为 1，则只有分子‘num’。

若 *str* 为 NULL，则结果用当前的内存分配函数进行内存分配。所分配的内存块将是 strlen(str)+1 字节，这正好够存储字符串及 0 结束符。

若 *str* 不是 NULL，则它应该指向足够存储结果的内存块，那应该是

```
mpz_sizeinbase (mpq_numref(op), base)
+ mpz_sizeinbase (mpq_denref(op), base) + 3
```

三个额外的字节分别是可能的负号，可能的斜线，以及 0 结束符。

返回值是指向结果字符串的指针，可能是分配的内存块，或者是给定的 *str*。

4.3 算术运算函数

`void mpq_add (mpq_t sum, mpq_t addend1, mpq_t addend2)` [函数]

置 *sum* 为 $addend1 + addend2$。

`void mpq_sub (mpq_t difference, mpq_t minuend, mpq_t subtrahend)` [函数]

置 $diffenrence$ 为 $minuend - subtrahend$。

`void mpq_mul (mpq_t product, mpq_t multiplier, mpq_t multiplicand)` [函数]

置 *product* 为 $multiplier \times multiplicand$。

`void mpq_mul_2exp (mpq_t rop, mpq_t op1, unsigned long int op2)` [函数]

置 *rop* 为 $op1 \times 2^{op2}$。

`void mpq_div (mpq_t quotient, mpq_t dividend, mpq_t divisor)` [函数]

置 *quotient* 为 $dividend/divisor$。

`void mpq_div_2exp (mpq_t rop, mpq_t op1, unsigned long int op2)` [函数]

置 *rop* 为 $op1/2^{op2}$。

`void mpq_neg (mpq_t negated_operand, mpq_t operand)` [函数]

置 negated_operand 为 $-operand$。

`void mpq_abs (mpq_t rop, mpq_t op)` [函数]

置 *rop* 为 *op* 的绝对值。

`void mpq_inv (mpq_t inverted_number, mpq_t number)` [函数]

置 inverted_number 为 $1/number$，如果分母为 0，那么程序将产生被 0 除错误。

4.4 比较函数

`int mpq_cmp (mpq_t op1, mpq_t op2)` [函数]

比较 $op1$ 和 $op2$，若 $op1 > op2$ 则返回正值，若 $op1 = op2$ 则返回 0，若 $op1 < op2$ 则返回 -1。

要确定两个有理数是否相等，`mpq_equal` 比 `mpq_cmp` 更快。

`int mpq_cmp_ui(mpq_t op1, unsigned long int num2, unsigned long int den2)` [宏]
`int mpq_cmp_si (mpq_t op1, long int num2, unsigned long int den2)` [宏]

比较 $op1$ 和 $num2/den2$，若 $op1 > num2/den2$ 则返回正值，若 $op1 = num2/den2$ 则返回 0，若 $op1 < num2/den2$ 则返回 -1。

$num2$ 和 $den2$ 允许有公因子。

这两个函数用宏来实现，它们需要对参数估值一次以上。

`int mpq_sgn (mpq_t op)` [宏]

若 $op > 0$ 则返回正值，若 $op = 0$ 则返回 0，若 $op < 0$ 则返回 -1。

这个函数用宏来实现，它需要对参数估值一次以上。

`int mpq_equal (mpq_t op1, mpq_t op2)` [函数]

若 $op1$ 与 $op2$ 相等则返回非 0，否则返回 0。尽管 `mpq_cmp` 也可用于相同目的，但是这个函数要快得多。

4.5 应用整数函数于有理数

`mpq` 函数集很小。实际上很少有函数既处理输入又处理输出，下面的函数提供了直接操作一个 `mpq_t` 的分子和分母的途径。
需要注意的是，如果对分子与/或分母的赋值可以提取出如本章开头所描述的规范型，那么在对 `mpq_t` 使用 `mpq` 函数之前必须对它调用函数 `mpq_canonicalize` 以规范化之。

`mpz_t mpq_numref (mpq_t op)` [宏]
`mpz_t mpq_denref (mpq_t op)` [宏]

分别返回对 op 的分子和分母的引用。`mpz` 函数可以作用于这两个宏的返回值。

`void mpq_get_num (mpz_t numerator, mpq_t rational)` [函数]
`void mpq_get_den (mpz_t denominator, mpq_t rational)` [函数]
`void mpq_set_num (mpq_t rational, mpz_t numerator)` [函数]
`void mpq_set_den (mpq_t rational, mpz_t denominator)` [函数]

取或者置有理数的分子和分母。这些函数等价于对某个适当的 `mpq_numref` 或 `mpq_denref` 调用 `mpz_set` 函数。建议直接使用 `mpq_numref` 和 `mpq_denref` 而不要调用这些函数。

4.6 输入输出函数

使用这些函数之前最好同时包含‘stdio.h’和‘gmp.h’，因为这样可以允许‘gmp.h’定义这些函数的原型。
对这些函数中的 *stream* 参数传递 `NULL` 指针，可能使得它们分别从 `stdin` 读和对 `stdout` 写。

`size_t mpq_out_str (FILE *stream, int base, mpq_t op)` [函数]

将 *op* 作为基 *base* 的数字串输出到标准输出流 *stream* 中，其中基 *base* 可从 2 到 36。输出是‘num/den’的形式，或者当分母为 1 时，只有‘num’。

返回所写的字节数，或者若错误发生则返回 0。

`size_t mpq_inp_str (mpq_t rop, FILE *stream, int base)` [函数]

从 *stream* 读取数字串并把它们转换成有理数置入 *rop* 中。所有的空位符被读取后立即丢弃。返回所读取的字符数 (含空格)，或者若错误发生则返回 0。

输入可以是分数‘17/63’或者简单的整数‘123’。读取停止于第一个非此类型的输入，或者字符串中不允许出现的空位符。如果输入不是规范型，那么在使用其它函数之前必须调用 `mpq_canonicalize`。

基 *base* 可以从 2 到 36，或者为 0，此时首字符作为参考：0x 表示十六进制，0 表示八进制，其他则是十进制。注意对分子和分母这个判断是分别来做的，例如 0x10/11 表示 16/11，而 0x10/0x11 表示 16/17。

第 五 章 浮点函数

GMP 浮点数存储于 mpf_t 对象中，所有的浮点函数有 mpf_ 前缀。

每个浮点数的尾数有一个用户可选的精度，它只受可用内存的限制。每个变量都有自身的精度，任何时候都可以对它进行增减。

每个浮点数的指数都是固定精度的，在大多数系统中是一个机器字。在当前的实现中是一个 limb 的量，例如对于 32 比特系统，这就意味着大约在 $2^{-68719476768}$ 到 $2^{68719476768}$ 范围内，而对于 64 比特系统这将是更大的。但是需要注意的是，mpf_get_str 只能返回 mp_exp_t 范围内的精度，并且目前 mpf_set_str 不能接受超过 long 范围的指数。

每个变量为实际上使用的尾数保存一个长度。这就意味着如果一个浮点数实际上只用少量的位来表示，那么即使选定的精度很高，在计算中它也只使用少量的位。

所有的计算都要对目的变量精度进行操作。每个函数都定义为用‘无限精度’进行计算，而后再切断为目的精度。但是需要完成的工作只是在这种定义下需要确定什么样的结果。

为变量选定的精度只是一个最小值，GMP 可将它加大一些以使之能够更有效的进行计算。当前这就意味着，舍入为整数个 limb，或者有时要更多的部分 limb，这依赖于尾数的高 limb。但是应用不应该关心这些细节。

尾数以二进制的形式存储，可以想像用二进制来描述精度，结果就使得像 0.1 这样十进制分数无法准确的表示，同样普通 IEEE 双精度浮点数也是如此。这就使得对于涉及钱和其它需要精确十进制浮点数值的计算，它都是很不胜任的。(合适的近似整数，又或是有理数，也许是更好的选择)

mpf 函数和变量对‘无限’和‘不是一个数’没有特别的概念，应用必须注意不要溢出指数，否则结果是不可预料的。在未来的版本中这将有所改变。

需要注意的是 mpf 函数没有试图作为 IEEE P754 算术的平滑扩展。实际上，一台计算机上的结果可能与另一台有不同字长的计算机的结果是不同的。

5.1 初始化函数

`void mpf_set_default_prec (unsigned long int prec)` [函数]

将默认精度置为**至少**$prec$ 比特，所有后续调用 mpf_init 的变量都使用这个精度，但是前面已初始化过的变量不受影响。

`unsigned long int mpf_get_default_prec (void)` [函数]

返回实际使用的默认精度。

一个 mpf_t 对象在对它存储数值之前必须进行初始化，mpf_init 和 mpf_init2 函数用以实现这一目的。

`void mpf_init (mpf_t x)` [函数]

初始化 x 为 0，一般地，在两次初始化之间，变量应该只初始化一次，而至少用 mpf_clear 清除一次。除非默认精度已有 mpf_set_default_prec 确定，x 的精度不予定义。

`void mpf_init2 (mpf_t x, unsigned long int prec)` [函数]

初始化 x 为 0，并将默认精度置为**至少**$prec$ 比特，一般地，在两次初始化之间，变量应该只初始化一次，而至少用 mpf_clear 清除一次。

void mpf_clear (mpf_t x) [函数]

释放 x 所占的空间。对所有 mpf_t 变量，应该确保在使用完它们了之后，调用该函数以清除之。

这里是一个如何初始化浮点数的例子：

```
{
    mpf_t x, y;
    mpf_init (x); /* use default precision */
    mpf_init2 (y, 256); /* precision at least 256 bits */
    ...
    /* Unless the program is about to exit, do ... */
    mpf_clear (x);
    mpf_clear (y);
}
```

下面三个函数对于在计算中改变精度很有用。典型的应用是像 Newton-Raphson 迭代算法那样逐步的调整精度，以使计算精度接近匹配于数值实际精确部分的值。

unsigned long int mpf_get_prec (mpf_t op) [函数]

以比特为单位，返回 op 当前的精度。

void mpf_set_prec (mpf_t rop, unsigned long int prec) [函数]

置 rop 的精度为**至少**$prec$ 比特，rop 的数值将按新的精度进行舍入。

这个函数需要对 realloc 的调用，所有不应该用于紧密地循环中。

void mpf_set_prec_raw (mpf_t rop, unsigned long int prec) [函数]

置 rop 的精度为**至少**$prec$ 比特，不改变内存分配。

$prec$ 必须不超过 rop 内存分配的精度，这是 rop 初始化的精度，或者最近的 mpf_set_prec 赋予的精度。

rop 的值不会被改变，实际上如果它有比 $prec$ 更高的精度，它将保持更高精度不变。如果对 rop 置入新值，那么将使用新的精度。

在调用 mpf_clear 或完全 mpf_set_prec 之前，必须调用另一次 mpf_set_prec_raw 以恢复 rop 到原始内存分配的精度。如果不这样做的话，可能导致不可预料的结果。

在 mpf_set_prec_raw 之前可以用 mpf_get_prec 得到原始内存分配精度。在 mpf_set_prec_raw 之后，它可以反映所置的 $prec$ 值。

mpf_set_prec_raw 是在计算中使用不同精度的 mpf_t 变量的有效方法，可以在一个迭代中逐步的增加精度，或者只是在计算中为不同的目的选用不同的精度。

5.2 赋值函数

这些函数为已初始化的浮点数赋新值。

void mpf_set (mpf_t rop, mpf_t op) [函数]

void mpf_set_ui (mpf_t rop, unsigned long int op) [函数]

`void mpf_set_si (mpf_t rop, signed long int op)` [函数]
`void mpf_set_d (mpf_t rop, double op)` [函数]
`void mpf_set_z (mpf_t rop, mpz_t op)` [函数]
`void mpf_set_q (mpf_t rop, mpq_t op)` [函数]

用 *op* 对 *rop* 赋值。

`int mpf_set_str (mpf_t rop, char *str, int base)` [函数]

用字符串 *str* 对 *rop* 赋值。字符串应该是“`M@N`”形式，或者如果基为 10 或更小，那么也可以是“`MeN`”形式，其中 `M` 是尾数，而 `N` 是指数。尾数总是在指定基下表示的，而指数可能是在指定基下表示，也可能当基 *base* 为负数时，它采用的则是十进制。期望的十进制小数点已经从当前地点取出，而放置于系统提供的 `localeconv` 中。

基 *base* 可以从 2 到 36 或者从 -36 到 -2，负的基使得指数以十进制表示。

与相应的 `mpz` 函数不同，如果基 *base* 为 0，基不由首字符来推断，这样就避免了‘0.23’这类数被看作是八进制。

空格允许出现在字符串中，它只是被简单的忽略掉。[实际上并不完全这样，空格只在字符串的开始以及尾数中被忽略，而在其他地方不会被忽略，例如在负号之后或指数之中]

如果字符串是基为 *base* 的有效数，那么函数返回 0，否则返回 -1。

`void mpf_swap (mpf_t rop1, mpf_t rop2)` [函数]

有效的交换 *rop*1 和 *rop*2，其数值以及精度都被交换。

5.3 初始化赋值组合函数

方便起见，GMP 提供了并行初始化 - 赋值函数，这些函数在初始化输出后立即在其中存储相应的值。这些函数一般有 `mpf_init_set...` 的形式。
浮点数用 `mpf_init_set...` 函数初始化之后，可以用作普通浮点函数的源操作数。不要对已初始化的变量使用 `mpf_init_set...` 函数。

`void mpf_init_set (mpf_t rop, mpf_t op)` [函数]
`void mpf_init_set_ui (mpf_t rop, unsigned long int op)` [函数]
`void mpf_init_set_si (mpf_t rop, signed long int op)` [函数]
`void mpf_init_set_d (mpf_t rop, double op)` [函数]

初始化 *rop* 并置其值为 *op*。

rop 的精度将会置为当前的默认精度，例如由 `mpf_set_default_prec` 设定的精度。

`int mpf_init_set_str (mpf_t rop, char *str, int base)` [函数]

初始化 *rop* 并用 *str* 对其赋值。赋值操作细节可参看前面的 `mpf_set_str` 函数。

需要注意的是即使有错误发生 *rop* 也会被初始化。(也就是说，必须用 `mpf_clear` 来清除之。)

rop 的精度将会置为当前的默认精度，例如由 `mpf_set_default_prec` 设定的精度。

5.4 转换函数

`double mpf_get_d (mpf_t op)` [函数]

将 *op* 转换成双精度浮点数。

`double mpf_get_d_2exp (signed long int *exp, mpf_t op)` [函数]

寻找 d 和 exp 使得 $d \times 2^{exp}$ 很好的逼近 *op*，其中 $0.5 \leq |d| < 1$。这个函数类似于标准 C 函数 `frexp`。

`long mpf_get_si (mpf_t op)` [函数]
`unsigned long mpf_get_ui (mpf_t op)` [函数]

将 *op* 转换成 `long` 或 `unsinged long`，舍去所有分数部分。如果对于返回类型来说，*op* 值太大，那么结果是不确定的。

也可参考 `mpf_fits_slong_p` 和 `mpf_fits_ulong_p`。

```
char * mpf_get_str (char *str,  mp_exp_t *expptr, int base,
                                size_t n_digits, mpf_t op)
```
[函数]

将 *op* 转换成基为 *base* 的数字串，其中基 *base* 可从 2 到 36。产生 n_digits 位数字，尾部的 0 不被返回。不会产生超过足以精确表示 *op* 的数字位。如果 n_digits 为 0，那么将会产生精确表示 *op* 的最大数字位。

如果 *str* 为 `NULL`，那么返回的字符串用当前的内存分配函数进行内存分配。内存块的长度将是 `strlen(str)+1` 字节，这刚好足够表示字符串以及 0 结束符。

如果 *str* 不是 `NULL`，那么它必须指向长度为 $n_digits + 2$ 的内存块，以容纳尾数、可能的负号以及 0 结束符。当 n_digits 为 0 时，要取得所有的有效数字，应用无法知道所需的空间，因此在这种情况下 *str* 必须是 `NULL`。

产生的字符串是一个分数，它隐藏了直到第一数字左边的一个小数点。适当的指数将返回到 `expptr` 中。例如分数 3.1416 将会返回字符串“31416”以及指数 1。

当 *op* 为 0 时，返回的字符串将是空串，返回指数为 0。

指向返回字符串的指针将被返回，这个指针或者为分配的内存块或者为给定的 *str*。

5.5 算术函数

`void mpf_add (mpf_t rop, mpf_t op1, mpf_t op2)` [函数]
`void mpf_add_ui (mpf_t rop, mpf_t op1, unsigned long int op2)` [函数]

置 *rop* 为 $op1 + op2$。

`void mpf_sub (mpf_t rop, mpf_t op1, mpf_t op2)` [函数]
`void mpf_ui_sub (mpf_t rop, unsigned long int op1, mpf_t op2)` [函数]
`void mpf_sub_ui (mpf_t rop, mpf_t op1, unsigned long int op2)` [函数]

置 *rop* 为 $op1 - op2$。

`void mpf_mul (mpf_t rop, mpf_t op1, mpf_t op2)` [函数]
`void mpf_mul_ui (mpf_t rop, mpf_t op1, unsigned long int op2)` [函数]

置 *rop* 为 $op1 \times op2$。

除数为 0 的除法是没有意义的，对除法函数的除数传递 0 将会产生被 0 除错误，这就允许用户使用处理其他算术异常一样的方法处理这些函数中异常。

`void mpf_div (mpf_t rop, mpf_t op1, mpf_t op2)` [函数]
`void mpf_ui_div (mpf_t rop, unsigned long int op1, mpf_t op2)` [函数]
`void mpf_div_ui (mpf_t rop, mpf_t op1, unsigned long int op2)` [函数]

置 rop 为 $op1/op2$。

`void mpf_sqrt (mpf_t rop, mpf_t op)` [函数]
`void mpf_sqrt_ui (mpf_t rop, unsigned long int op)` [函数]

置 rop 为 $\sqrt{op}$。

`void mpf_pow_ui (mpf_t rop, mpf_t op1, unsigned long int op2)` [函数]

置 rop 为 $op1^{op2}$。

`void mpf_neg (mpf_t rop, mpf_t op)` [函数]

置 rop 为 $-op$。

`void mpf_abs (mpf_t rop, mpf_t op)` [函数]

置 rop 为 op 的绝对值。

`void mpf_mul_2exp (mpf_t rop, mpf_t op1, unsigned long int op2)` [函数]

置 rop 为 $op \times 2^{op2}$。

`void mpf_div_2exp (mpf_t rop, mpf_t op1, unsigned long int op2)` [函数]

置 rop 为 $op/2^{op2}$。

5.6 比较函数

`int mpf_cmp (mpf_t op1, mpf_t op2)` [函数]
`int mpf_cmp_d (mpf_t op1, double op2)` [函数]
`int mpf_cmp_ui (mpf_t op1, unsigned long int op2)` [函数]
`int mpf_cmp_si (mpf_t op1, signed long int op2)` [函数]

比较 $op1$ 和 $op2$，若 $op1 > op2$ 则返回正值，若 $op1 = op2$ 则返回 0，若 $op1 < op2$ 则返回 -1。

`int mpf_eq (mpf_t op1, mpf_t op2, unsigned long int op3)` [函数]

如果 $op1$ 与 $op2$ 的前 $op3$ 比特相同则返回非 0，否则返回 0。或者说测试 $op1$ 也 $op2$ 是否近似相等。

警告：当前进行的只是全 `limbs` 的比较，且采用精确比较的风格。在未来像 1000 和 0111 这样的数值可能被认为有 3 比特是相同的 (在它们的差很小的基础上)。

`void mpf_reldiff (mpf_t rop, mpf_t op1, mpf_t op2)` [函数]

比较 $op1$ 和 $op2$ 之间的相对差，把结果也就是 $|op1 - op2|/op1$ 储存于 rop。

`int mpf_sgn (mpf t op)` [宏]

若 $op > 0$ 则返回正值，若 $op = 0$ 则返回 0，若 $op < 0$ 则返回 -1。

这个函数用宏来实现，它需要对参数估值一次以上。

5.7 输入输出函数

包括处理来自标准输入输出流的输入和输出的函数。对这些函数中的 *stream* 参数传递 NULL 指针，可能使得它们分别从 stdin 读和对 stdout 写。
使用这些函数之前最好同时包含‘stdio.h’和‘gmp.h’，因为这样可以允许‘gmp.h’定义这些函数的原型。

`size_t mpf_out_str (FILE *stream, int base, size_t n_digits, mpf_t op)` [函数]

把 *op* 作为数字串打印到 *stream*。返回所写的字节数，若错误发生则返回 0。

尾数具有‘0.’这样的前缀，而且在给定基下进行表示，其中基可以从 2 到 36。然后会打印一个指数，用‘e’来分开。或者若基大于 10，则用‘@’来分开。指数总是用十进制表示。十进制小数点已经从当前地点取出，而放置于系统提供的 localeconv 中。

产生 n_digits 位数字，而不会产生超过足以精确表示 *op* 的数字位。如果 n_digits 为 0，那么将会产生精确表示 *op* 的最大数字位。

`size_t mpf_inp_str (mpf_t rop, FILE *stream, int base)` [函数]

从 *stream* 中读取基为 *base* 的字符串，将读到的实数置于 *rop* 中。字符串应该是“M@N”形式，或者如果基为 10 或更小，那么也可以是“MeN”形式，其中 M 是尾数，而 N 是指数。尾数总是在指定基下表示的，而指数可能是在指定基下表示，也可能当基 *base* 为负数时，它采用的则是十进制。期望的十进制小数点已经从当前地点取出，而放置于系统提供的 localeconv 中。

基 *base* 可以从 2 到 36 或者从 -36 到 -2，负的基使得指数以十进制表示。

与相应的 mpz 函数不同，如果基 *base* 为 0，基不由首字符来推断，这样就避免了‘0.23’这类数被看作是八进制。

返回所读的字节数，若错误发生则返回 0。

5.8 杂类函数

`void mpf_ceil (mpf_t rop, mpf_t op)` [函数]
`void mpf_floor (mpf_t rop, mpf_t op)` [函数]
`void mpf_trunc (mpf_t rop, mpf_t op)` [函数]

op 舍入为一整数，然后置入 *rop* 中。mpf_ceil 向下一个更大的整数舍入，mpf_floor 向下一个更小的整数舍入，mpf_trunc 向 0 舍入。

`int mpf_integer_p (mpf_t op)` [函数]

若 *op* 为整数则返回非 0。

`int mpf_fits_ulong_p (mpf_t op)` [函数]
`int mpf_fits_slong_p (mpf_t op)` [函数]
`int mpf_fits_uint_p (mpf_t op)` [函数]
`int mpf_fits_sint_p (mpf_t op)` [函数]
`int mpf_fits_ushort_p (mpf_t op)` [函数]
`int mpf_fits_sshort_p (mpf_t op)` [函数]

若 *op* 舍入为一个整数时在相应的 C 数据类型范围内则返回非 0。

```
void mpf_urandomb (mpf_t rop,  gmp_randstate_t state,
                               unsigned long int nbits)
```
[函数]

产生一个均匀分布的随机浮点数置于 *rop* 中，使得 $0 \leq rop < 1$，并且尾数有 *nbits* 个有效数字比特。

在调用这些函数之前，必须调用某个 `gmp_randinit` 对值 *state* 进行初始化。

`void mpf_random2 (mpf_t rop, mp_size_t max_size, mp_exp_t exp)` [函数]

产生至少 `max_size` 个 `limbs` 的 01 二进制长串的随机浮点数。浮点数的指数在 $-exp$ 到 exp 范围内 (单位 `limbs`)。这个函数对测试函数和算法很有用，因为这类随机数被证明更容易引发边界值 bug。当 `max_size` 为负时，可以产生负值的随机数。

第 六 章 低级函数

本章描述低级 GMP 函数，它们被用来实现高级 GMP 函数，但是也可以被用于苛求时间的用户代码。
这些函数都有前缀 mpn_。
mpn 设计时追求的是越快越好，而不提供连续调用的接口。不同的函数可能有些相似的接口，但是有一些变数使得它们难以使用。这些函数尽量少的做一些与实际的多精度计算无关的操作，这样就可以不必在不是所有调用者需要的东西上花费时间。
源操作数由指向最低有效 limb 的指针和 limb 量组成，目的操作数只是一个指针。保证目的操作数有足够的空间来存放返回值，这是调用者的责任。
通过这种指定操作数的方式，可以在一个参数的附属区域进行计算以及存储结果于目的操作数的附属区域。
所有函数共同要求的是每个源区域至少需要一 limb 的空间。没有长度参数可以为 0，除非特别约定，本地操作允许源与目的是相同的，但是它们不能只是部分交叠的。
mpn 函数是实现 mpz_、mpq_、mpf_ 函数的基础。
下面这个例子对开始于 $s1p$ 和 $s2p$ 的整数相加，并将和写入 $destp$ 中，其中参数都有 $n$limbs 空间。

```
cy = mpn_add_n (destp, s1p, s2p, n)
```

在这里用到的符号是，源操作数由花括号中的指向最低有效 limb 的指针和 limb 量来指定。例如 $\{s1p, s1n\}$。

```
mp_limb_t mpn_add_n (mp_limb_t *rp,  const mp_limb_t *s1p, const
                                     mp_limb_t *s2p, mp_size_t n)
```
[函数]

$\{s1p, n\}$ 与 $\{s2p, n\}$ 相加，将结果的 n 个最低有效 limb 写入 rp，返回进位，0 或 1。

这是加法的低级函数。它是做加法的最好函数，因为它是针对大多数 CPU 用汇编代码写的。对于加上自身 (也就是说，$s1p$ 与 $s2p$ 相同)，应该使用 mpn_lshift 左移一位以获得最优的速度。

```
mp_limb_t mpn_add_1 (mp_limb_t *rp,  const mp_limb_t *s1p, mp_size_t n,
                                     mp_limb_t s2limb)
```
[函数]

$\{s1p, n\}$ 与 $s2limb$ 相加，将结果的 n 个最低有效 limb 写入 rp，返回进位，0 或 1。

```
mp_limb_t mpn_add (mp_limb_t *rp,  const mp_limb_t *s1p, mp_size_t s1n,
                                   const mp_limb_t *s2p, mp_size_t s2n)
```
[函数]

$\{s1p, n\}$ 与 $\{s2p, n\}$ 相加，将结果的 $s1n$ 个最低有效 limb 写入 rp，返回进位，0 或 1。

这个函数要求 $s1n \geq s2n$。

```
mp_limb_t mpn_sub_n (mp_limb_t *rp,  const mp_limb_t *s1p, const
                                     mp_limb_t *s2p, mp_size_t n)
```
[函数]

从 $\{s1p, n\}$ 减去 $\{s2p, n\}$，将结果的 n 个最低有效 limb 写入 rp，返回借位，0 或 1。

这是减法的低级函数。它是做减法的最好函数，因为它是针对大多数 CPU 用汇编代码写的。

```
mp_limb_t mpn_sub_1 (mp_limb_t *rp, const mp_limb_t *s1p,  mp_size_t n,
                                                          mp_limb_t s2limb)
```
[函数]

从 $\{s1p, n\}$ 中减去 $s2limb$，将结果的 n 个最低有效 limb 写入 rp，返回借位，0 或 1。

```
mp_limb_t mpn_sub (mp_limb_t *rp,     const mp_limb_t *s1p, mp_size_t s1n,
const mp_limb_t *s2p, mp_size_t s2n)
```
[函数]

从 $\{s1p, n\}$ 中减去 $\{s2p, n\}$，将结果的 $s1n$ 个最低有效 limb 写入 rp，返回借位，0 或 1。

这个函数要求 $s1n \geq s2n$。

```
void mpn_mul_n (mp_limb_t *rp,  const mp_limb_t *s1p,
                                const mp_limb_t *s2p, mp_size_t n)
```
[函数]

$\{s1p, n\}$ 与 $\{s2p, n\}$ 相乘，将结果的 $2 * n$ 个最低有效 limb 写入 rp。

即使乘积的最高有效 limb 为 0，目的操作数也必须有 $2 * n$limbs 的空间。目的与两个源之间不允许有交叠。

```
mp_limb_t mpn_mul_1 (mp_limb_t *rp,  const mp_limb t *s1p,
                                     mp_size_t n, mp_limb_t s2limb)
```
[函数]

$\{s1p, n\}$ 与 $s2limb$ 相乘，将结果的 n 个最低有效 limb 写入 rp。返回乘积的最高有效 limb，如果 $rp \leq s1p$，那么 $\{s1p, n\}$ 与 $\{rp, n\}$ 允许有交叠。

这是一个低级 GMP 函数，与 GMP 中的其他操作一样，它是用来做通常乘法的内置模块，针对大多数 CPU 用汇编实现。

如果 $s2limb$ 是 2 的方幂，那么不用调用这个函数，用 mpn_lshift 左移等于 $s2limb$ 对数的一个量来代替之，这样可以获得最优的速度。

```
mp_limb_t mpn_addmul_1 (mp_limb_t *rp,  const mp_limb_t *s1p,
                                        mp_size_t n, mp_limb_t s2limb)
```
[函数]

$\{s1p, n\}$ 与 $s2limb$ 相乘，并将乘积的 n 个最低有效 limb 与 $\{rp, n\}$ 相加，将结果写入 rp。返回乘积的最高有效 limb 加上加法的进位。

这是一个低级 GMP 函数，与 GMP 中的其他操作一样，它是用来做通常乘法的内置模块，针对大多数 CPU 用汇编实现。

```
mp_limb_t mpn_submul_1 (mp_limb_t *rp,  const mp_limb_t *s1p,
                                        mp_size_t n, mp_limb_t s2limb)
```
[函数]

$\{s1p, n\}$ 与 $s2limb$ 相乘，并将乘积的 n 个最低有效 limb 从 $\{rp, n\}$ 中减去，将结果写入 rp。返回乘积的最高有效 limb 减去减法的借位。

这是一个低级 GMP 函数，与 GMP 中的其他操作一样，它是用来做通常乘法的内置模块，针对大多数 CPU 用汇编实现。

```
mp_limb_t mpn_mul (mp_limb_t *rp,  const mp_limb_t *s1p, mp_size_t s1n,
                                   const mp_limb_t *s2p, mp_size_t s2n)
```
[函数]

$\{s1p, s1n\}$ 与 $\{s2p, s2n\}$ 相乘，将结果写入 rp。返回结果的最高有效 limb。

目的操作数必须有 $s1n + s2n$limbs 的空间，即使结果可能要少一 limb 也需如此。

这个函数要求 $s1n \geq s2n$，目的操作数必须与两个输入操作数不同。

`void mpn_tdiv_qr (mp_limb_t *qp, mp_limb_t *rp, mp_size_t qxn, const mp_limb t *np, mp_size_t nn, const mp_limb_t *dp, mp_size_t dn)` [函数]

用 $\{dp, dn\}$ 除 $\{np, nn\}$，将商置入 $\{qp, nn - dn + 1\}$，余数置入 $\{rp, dn\}$，商向 0 舍入。

参数之间不允许有交叠，要求必须有 $nn \geq dn$，dp 的最高有效 limb 必须非 0，qxn 必须为 0。

`mp_limb_t mpn_divrem (mp_limb_t *r1p, mp_size_t qxn, mp_limb_t *rs2p, mp_size_t rs2n, const mp_limb_t *s3p, mp_size_t s3n)` [函数]

[这个函数过时了，为了更好性能请用 mpn_tdiv_qr 作为代替。]

用 $\{s3p, s3n\}$ 除 $\{rs2p, rs2n\}$，除了最高有效 limb 返回之外，商写入 $r1p$。余数代替被除数存储于 $rs2p$，其长度将是 $s3n$(也就是说，与除数相同)。

要求必须有 $rs2n \geq s3n$，除数的最高有效比特位必须为 1。

如果不必返回商，可传递 $rs2p + s3n$ 到 $r1p$。除了这个特殊情况，参数之间不允许出现交叠。

返回商的最高有效 limb，0 或 1。

$r1p$ 需要的空间为 $rs2n - s3n + qxn$limbs。

`mp_limb_t mpn_divrem_1 (mp_limb_t *r1p, mp_size_t qxn, mp_limb_t *s2p, mp_size_t s2n, mp_limb_t s3limb)` [函数]

`mp_limb_t mpn_divmod_1 (mp_limb_t *r1p, mp_limb_t *s2p, mp_size_t s2n, mp_limb_t s3limb )` [宏]

用 $s3limb$ 除 $\{s2p, s2n\}$，将商写入 $r1p$，返回余数。

整数商被写入 $\{r1p + qxn, s2n\}$，另外可能产生 qxnlimb 并写入 $\{r1p, qxn\}$。qxn 和 $s2n$ 都可以为 0。在大部分应用中，qxn 往往为 0。

$r1p$ 和 $s2p$ 可以相同或者完全分离，不允许部分交叠。

`mp_limb_t mpn_divmod (mp_limb_t *r1p, mp_limb_t *rs2p, mp_size_t rs2n, const mp_limb_t *s3p, mp_size_t s3n)`
[函数]

[这个函数是过时的，请用 mpn_tdiv_qr 代替之。]

`mp_limb_t mpn_divexact_by3 (mp_limb_t *rp, mp_limb_t *sp, mp_size_t n)` [宏]

`mp_limb_t mpn_divexact_by3c (mp_limb_t *rp, mp_limb_t *sp, mp_size_t n, mp_limb_t carry)` [函数]

用 3 除 $\{sp, n\}$，期望它是整除的，把结果写入 $\{rp, n\}$。若可被 3 整除，则返回 0，结果是商。若不整除，则返回非 0，结果是没什么用的。

mpn_divexact_by3c 可以有一个初始进位参数，它可以是前一次调用的返回值，所以大型计算可以从低到高分成一块一块的。mpn_divexact_by3 只是使用 0 进位参数调用 mpn_divexact_by3c 的一个宏。

这个程序使用了乘以逆的技巧，所以在快乘法慢除法的 CPU 上要比 mpn_divrem_1 来的快。

源 a、结果 q、长度 n、初始进位 i 以及返回值 c 应该满足 $cb^n + a - i = 3q$，其中 $b = 2^{\texttt{mp\_bits\_per\_limb}}$。返回值总是 0、1 或 2，初始进位也必须是 0、1 或 2。当 $c = 0$ 显然 $q = (a - i)/3$。当 $c \neq 0$，则余数 $(a - i) \mod 3$ 由 $3 - c$ 给定，因为总有 $b \equiv 1 \mod 3$(当前总是如此，因为选用的 mp_bits_per_limb 总是偶数)。

mp_limb_t mpn_mod_1 (mp_limb_t *s1p, mp_size_t s1n, mp_limb_t s2limb) [函数]

用 $s2limb$ 除 $\{s1p, s1n\}$，返回余数，$s1n$ 可以为 0。

mp_limb_t mpn_bdivmod (mp_limb_t *rp, mp_limb_t *s1p, mp_size_t s1n, const mp_limb_t *s2p, mp_size_t s2n, unsigned long int d) [函数]

这个函数将 $q = \{s1p, s1n\}/\{s2p, s2n\} \mod 2^d$ 的低 $\lfloor d/\texttt{mp_bits_per_limb} \rfloor$limbs 置入 rp，返回 q 的高 $d \mod$ mp_bits_per_limb 比特。

$\{s1p, s1n\} - q * \{s2p, s2n\} \mod 2^{s1n*\texttt{mp\_bits\_per\_limb}}$ 置入 $s1p$。因为这个差的低 $\lfloor d/\texttt{mp_bits_per_limb} \rfloor$limbs 为 0，所以如果 $rp \leq s1p$，那么这个差可能与 $s1p$ 的低 limbs 相交叠。

这个函数要求 $s1n *$ mp_bits_per_limb $\geq D$ 且 $\{s2p, s2n\}$ 为奇数。

mp_limb_t mpn_lshift (mp_limb_t *rp, const mp_limb_t *sp, mp_size_t n, unsigned int count) [函数]

$\{sp, n\}$ 左移 $count$ 比特，将结果写入 $\{rp, n\}$。左移出去的比特作为返回值的最低有效 $count$ 比特被返回 (返回值的其他部分为 0)。

$count$ 必须在从 1 到 mp_bits_per_limb-1 范围之内，如果 $rp \geq sp$，则 $\{rp, n\}$ 与 $\{sp, n\}$ 允许交叠。

这个函数在大多数 CPU 上用汇编实现。

mp_limb_t mpn_rshift (mp_limb_t *rp, const mp_limb_t *sp, mp_size_t n, unsigned int count) [函数]

$\{sp, n\}$ 左移 $count$ 比特，将结果写入 $\{rp, n\}$。右移出去的比特作为返回值的最高有效 $count$ 比特被返回 (返回值的其他部分为 0)。

$count$ 必须在从 1 到 mp_bits_per_limb-1 范围之内，如果 $rp \leq sp$，则 $\{rp, n\}$ 与 $\{sp, n\}$ 允许交叠。

这个函数在大多数 CPU 上用汇编实现。

int mpn_cmp (const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n) [函数]

比较 $\{s1p, s1n\}$ 与 $\{s2p, s2n\}$，若 $s1 > s2$ 则返回正值，若相等则返回 0，若 $s1 < s2$ 则返回负值。

mp_size_t mpn_gcd (mp_limb_t *rp, mp_limb_t *s1p, mp_size_t s1n, mp_limb_t *s2p, mp_size_t s2n) [函数]

置 $\{rp, retval\}$ 为 $\{s1p, s1n\}$ 与 $\{s2p, s2n\}$ 的最大公因子。结果可能有 $s2n$ limbs，返回值是实际结果的长度。两个源操作数都被清除。

$\{s1p, s1n\}$ 必须有与 $\{s2p, s2n\}$ 一样多的比特。$\{s2p, s2n\}$ 必须是奇数。两个操作数都必须有非 0 的最高有效 limb。$\{s1p, s1n\}$ 与 $\{s2p, s2n\}$ 不允许发生交叠。

`mp_limb_t mpn_gcd_1 (const mp_limb_t *s1p, mp_size_t s1n, mp_limb_t s2limb)` [函数]

返回 $\{s1p, s1n\}$ 与 $s2limb$ 的最大公因子，两个操作数都必须非 0。

`mp_size_t mpn_gcdext (mp_limb_t *r1p, mp_limb_t *r2p, mp_size_t *r2n, mp_limb_t *s1p, mp_size_t s1n, mp_limb_t *s2p, mp_size_t s2n)` [函数]

计算 $\{s1p, s1n\}$ 与 $\{s2p, s2n\}$ 的最大公因子。储存 gcd 于 $\{r1p, retval\}$，第一辅助因子于 $\{r2p, *r2n\}$，若辅助因子为负则 $*r2n$ 为负。$r1p$ 和 $r2p$ 应该分别有 $s1n + 1$ `limbs` 的空间，但是返回值和通过 $r2n$ 储存的值才显示实际结果的长度。

要求满足 $\{s1p, s1n\} \geq \{s2p, s2n\}$，而且都必须非 0。$\{s1p, s1n+1\}$ 与 $\{s2p, s2n+1\}$ 的区域被清除。

辅助因子 $r1$ 满足 $r_2 s_1 + k s_2 = r_1$。第二个辅助因子 k 不会计算出来，但是很容易由 $(r_1 - r_2 s_1)/s_2$ 得到。

`mp_size_t mpn_sqrtrem (mp_limb_t *r1p, mp_limb_t *r2p, const mp_limb_t *sp, mp_size_t n)` [函数]

计算 $\{sp, n\}$ 的二次根，将结果置于 $\{r1p, \lfloor n/2 \rfloor\}$，将余数置于 $\{r2p, retval\}$，$r2p$ 需要 n `limbs` 的空间，但是返回值显示实际长度。

$\{sp, n\}$ 的最高有效 `limb` 必须非 0，$\{r1p, \lfloor n/2 \rfloor\}$ 与 $\{sp, n\}$ 必须完全分离。$\{r2p, n\}$ 与 $\{sp, n\}$ 必须相同或者完全分离。

如果不要求得到余数，那么 $r2p$ 可以为 `NULL`，此时返回值根据余数为 0 或非 0 分别为 0 或非 0。

返回值为 0 说明 $\{sp, n\}$ 是完全平方数，亦可见 `mpz_perfect_square_p`。

`mp_size_t mpn_get_str (unsigned char *str, int base, mp_limb_t *s1p, mp_size_t s1n)` [函数]

将 $\{s1p, s1n\}$ 转换成基为 $base$ 的 `raw unsigned char` 数组存储于 str，返回产生的字符数。字符串开头可能有 0，它不是 `ASCII` 码形式的，要把它转换成可打印格式，依赖于基和范围，需要加上 '0' 和 'A' 的 `ASCII` 码。$base$ 可以从 2 到 256。

输入 $\{s1p, s1n\}$ 的最高有效 `limb` 必须非 0，除了 $base$ 是 2 的方幂时，输入 $\{s1p, s1n\}$ 是不变的，其他情况下，输入是乱码的。

str 的空间必须足够 $s1n$ 长 `limbs` 所表示的最大整数加上一个额外的字符。

`mp_size_t mpn_set_str (mp_limb_t *rp, const unsigned char *str, size_t strsize, int base)` [函数]

将给定基 $base$ 下的 $\{str, strsize\}$ 处的字节串转换成 rp 处的 `limb` 串。

$str[0]$ 是最高字节，而 $str[strsize-1]$ 是最低字节，每个字节应该是从 0 到 $base-1$ 范围内的数值，而不是一个 `ASCII` 码字符，$base$ 可以从 2 到 256。

返回值是写到 rp 的 `limb` 数，如果最高输入字节非 0，那么 rp 处的最高 `limb` 也将是非 0 的，而且这里刚好只要所需数量的 `limbs`。

如果最高输入字节是 0，那么 0 最高 `limb` 可能写入 rp，并且会包含在返回值中。

$strsize$ 必须至少为 1，$\{str, strsize\}$ 与 rp 处的结果之间不允许有交叠。

`unsigned long int mpn_scan0(const mp_limb_t *s1p,unsigned long int bit)` [函数]

从比特位 *bit* 开始搜索 $s1p$ 直到下一个 0 比特。

要求在 $s1p$ 的比特位 *bit* 或其上具有 0 比特，这样函数才有所返回。

`unsigned long int mpn_scan1(const mp_limb_t *s1p,unsigned long int bit)` [函数]

从比特位 *bit* 开始搜索 $s1p$ 直到下一个 1 比特。

要求在 $s1p$ 的比特位 *bit* 或其上具有 1 比特，这样函数才有所返回。

`void mpn_random (mp_limb_t *r1p, mp_size_t r1n)` [函数]
`void mpn_random2 (mp_limb_t *r1p, mp_size_t r1n)` [函数]

产生一个长度为 $r1n$ 的随机数，将它存储于 $r1p$。最高 `limb` 往往是非 0 的。`mpn_random` 产生均匀分布的 `limb` 数据，`mpn_random2` 产生二进制表示的 01 长串。

`mpn_random2` 往往用于测试 `mpn` 函数的正确性。

`unsigned long int mpn_popcount (const mp_limb_t *s1p, mp_size_t n)` [函数]

计算 $\{s1p, n\}$ 中的 1 比特个数。

```
unsigned long int mpn_hamdist (const mp_limb_t  *s1p,const mp_limb_t
                                                *s2p, mp_size_t n)
```
[函数]

计算 $\{s1p, n\}$ 与 $\{s2p, n\}$ 的海明距离，也就是两个操作数的不同比特位的个数。

`int mpn_perfect_square_p (const mp_limb_t *s1p, mp_size_t n)` [函数]

当且仅当 $\{s1p, n\}$ 为完全平方数返回非 0。

6.1 Nails

本节的所有东西很大程度上都是经验所得，在 GMP 的未来版本中可能不再采用或者采取不兼容的改变。

Nails 是一个经验性的东西，它是 `mp_limb_t` 的头上留下来不使用的一些比特。在一些处理器上它可以明显地加快进位处理。

所有函数在入口处接受 `limb` 数据都希望 nails 比特是 0，返回数据的 nails 也应该是 0。这应该应用于 `limb` 向量以及单独的 `limb` 参数。

Nails 可以通过配置‘`--enable-nails`’来激活。默认情况下比特数会根据适合主处理器的情况来选择，但是指定的值可以通过‘`--enable-nails=N`’来选定。

在 `mpn` 级，严格的说，nails 编译版本与非 nails 编译版本是源程序及二进制目标码都不兼容的。但是只通过 `mpn` 函数来处理 `limbs` 可能在两种版本下都可以很好的工作。明智的使用下面的宏定义可以使得两中版本在源码级兼容。

对于高级程序，也就是 `mpz` 等，nails 编译版本与非 nails 编译版本是源程序及二进制目标码都完全兼容的。

`GMP_NAIL_BITS` [宏]
`GMP_NUMB_BITS` [宏]
`GMP_LIMB_BITS` [宏]

`GMP_NAIL_BITS` 是 nails 比特的个数，若未使用 nails 则其为 0。`GMP_NUMB_BITS` 是一个 `limb` 中数据比特的个数。`GMP_LIMB_BITS` 是一个 `mp_limb_t` 的比特数。

在所有情况下有：

```
GMP_LIMB_BITS == GMP_NAIL_BITS + GMP_NUMB_BITS
```

`GMP_NAIL_MASK` [宏]
`GMP_NUMB_MASK` [宏]

一个 `limb` 的 nails 部分和数据部分的比特掩码，若未使用 nails 则 `GMP_NAIL_MASK` 为 0。

`GMP_NAIL_MASK` 往往不会用到，因为 nails 部分可以通过 `x >> GMP_NUMB_BITS` 来得到，而且这样可以少用一个大常量，这对一些 `RISC` 芯片来说是有所帮助的。

`GMP_NUMB_MAX` [宏]

可存储于 `limb` 的数据部分的最大整数，这与 `GMP_NUMB_MASK` 相同，但是在作比较而不是位操作时可以更明显一些。

“nails”的概念本来自于手脚指甲，它在 limb(原意肢体的分支，如手和脚) 的端部。“numb”是 number 的缩写。未来将允许非 0 的 nail，这样可能提供整数在 `limb` 向量中不唯一的表示。这对向量处理器是很有帮助的，因为进位只可能传递到一到二个 `limb`。

第 七 章 随机数函数

GMP 中的伪随机序列用不同类型的 gmp_randstate_t 变量来生成，这个变量保存了所选算法和当前状态信息。这个变量必须调用某个 gmp_randinit 函数来初始化，并且可以由某个 gmp_randseed 函数来产生种子。

老式的随机数函数不接受 gmp_randstate_t 参数，而是共享这个类型的一个全局变量。它们使用默认的算法而且不需要种子。建议关心随机性的应用使用可接收 gmp_randstate_t 参数的新式随机数函数。

7.1 随机状态初始化

`void gmp_randinit_default (gmp_randstate_t state)` [函数]

用默认算法初始化 *state*。这是速度与随机性之间的权衡，建议没有特殊要求的应用使用它。

`void gmp_randinit_lc_2exp (gmp_randstate_t state, mpz_t a, unsigned long c, unsigned long m2exp )` [函数]

用线性同余算法 $X = (aX + c) \mod 2^{m2exp}$ 初始化 *state*。

这个算法中 X 的低位不是很随机。最低有效比特将不会有超过 2 的周期，第二有效比特不会超过 4，等等。因此实际上使用的是 X 的高位的半部分。

当随机数要产生 $m2exp/2$ 以上比特时，可以进行多次循环迭代，然后将结果连接起来。

`int gmp_randinit_lc_2exp_size(gmp_randstate_t state,unsigned long size)` [函数]

像前面的 gmp_randinit_lc_2exp 一样用线性同余算法初始化 *state*，a、c 和 $m2exp$ 从一个表中选择，选取使得每个 X 都有 *size* 比特 (或更多) 可用，也就是说，$m2exp = 2 \geq size$。

若成功则返回值为非 0，如果 *size* 比表提供的数据要大，那么返回值为 0。当前支持的最大 *size* 为 128。

`void gmp_randinit (gmp_randstate_t state, gmp_randalg_t alg, ...)` [函数]

这个函数已过时。

用选定的算法初始化 *state*。唯一的选择是 GMP_RAND_ALG_LC，也就是前面描述的 gmp_randinit_lc_2exp_size。需要 unsigned long 的第三个参数，它也就是前面那个函数的 *size*。GMP_RAND_ALG_DEFAULT 和 0 与 GMP_RAND_ALG_LC 相同。

gmp_randinit 对 gmp_errno 中的所有比特置 1，以此显示发生了错误，若 *alg* 不被支持则错误代码是 GMP_ERROR_UNSUPPORTED_ARGUMENT，若 *size* 参数太大则是 GMP_ERROR_INVALID_ARGUMENT。需要注意的是这个错误报告是非线程安全的 (这是用 gmp_randinit_lc_2exp_size 取而代之的一个好理由)。

`void gmp_randclear (gmp_randstate_t state)` [函数]

释放 *state* 所占的空间。

7.2 随机状态种子

`void gmp_randseed (gmp_randstate_t state, mpz_t seed)` [函数]
`void gmp_randseed_ui (gmp_randstate_t state, unsigned long int seed )` [函数]

对 *state* 置初始种子。

种子的长度决定了它可能产生的不同随机数序列的个数，种子的“质量”是给定种子相对于前面所用种子的随机性，并且这将会影响各个随机数序列的随机性。如果产生的数是用于重要应用，例如产生加密密钥，那么选择种子的方式是很关键的。

传统的做法是把系统时间作为种子，但是这种用法需要加倍的小心。如果应用经常更换种子，而系统时钟率较低，那么可能产生重复的数列，而且系统时间很容易猜测，因此如果对不可预知性有所要求，那么不应该把它作为唯一的种子源。在一些系统中有一个特殊的设备‘`/dev/random`’提供了更适合用作种子的随机数据。

第 八 章 格式输出

8.1 格式字符串

gmp_printf 及其友函数与标准 C 函数 printf 一样接收格式字符串。格式规范是下面这种形式：

```
% [flags] [width] [.[precision]] [type] conv
```

GMP 为 mpz_t,mpq_t,mpf_t 分别增加类型‘Z’,‘Q’,‘F’，并为 mp_limb_t 数组增加类型‘N’。‘Z’,‘Q’和‘N’的行为像整数一样，‘Q’有必要的话会打印一个‘/’和一个分母。‘F’的行为像浮点数。例如：

```
mpz_t z;
gmp_printf ("%s is an mpz %Zd\n", "here", z);
mpq_t q;
gmp_printf ("a hex rational: %#40Qx\n", q);
mpf_t f;
int n;
gmp_printf ("fixed point mpf %.*Ff with %d digits\n", n, f, n);
const mp_limb_t *ptr;
mp_size_t size;
gmp_printf ("limb array %Nx\n", ptr, size);
```

对于‘N’，像前面所述的 mpn 函数一样希望是最低位优先原则组织 limbs，负值的长度可以用来打印负值。

所有的标准 C printf 类型的行为与 C 库 printf 相同，可以自由的与 GMP 扩展混合使用。在目前的实现中，格式字符串的标准部分简单的提交给 printf 而只有 GMP 扩展才直接处理。

标志按下面的规范接收。GLIBC 类型的‘’’只针对于标准 C 类型 (非 GMP 类型)，而且只有 C 库支持时才有效。

0	用 0 填充 (而不是空格)
#	显示基为‘0x’、‘0X’或‘0’
+	总显示符号
(空格)	显示空格或者‘-’号
’	组数字，GLIBC 类型 (非 GMP 类型)

可选的长度和精度可以用数字在格式字符串中给定，或者像标准 printf 函数一样用‘*’来接收另一个 int 型的参数。标准类型的接收按下面的规范。‘h’和‘l’是方便的，对其他的来说，类型依赖于编译器类型 (或头文件) 而输出依赖于 C 库。

h	short
hh	char
j	intmax_t 或 uintmax_t
l	long 或 wchar_t
ll	long long
L	long double
q	quad_t 或 u_quad_t
t	ptrdiff_t
z	size_t

GMP 类型是：

F	mpf_t，浮点型换算
Q	mpq_t，整型换算
N	mp_limb_t 数组，整型换算
Z	mpz_t，整型换算

换算按下面规范接收。‘a’和‘A’总是支持 mpf_t 但是依赖于 C 库的标准 C 浮点型，‘m’和‘p’依赖于 C 库。

a A	十六进制浮点数，C99 风格
c	字符
d	十进制整数
e E	科学计数法格式浮点数
f	定点浮点数
i	与 d 相同
g G	定点或科学计数法浮点数
m	strerror 字符串，GLIBC 风格
n	存储迄今为止所写字符
o	八进制整数
p	指针
s	字符串
u	无符号整型
x X	十六进制整数

‘o’, ‘x’和‘X’对标准 C 类型来说是无符号的，但是对‘Z’, ‘Q’和‘N’来说它们是有符号的。‘u’对‘Z’, ‘Q’和‘N’来说是没有意义的。

‘n’可用于任意类型，甚至是 GMP 类型。

其他被 C 库 printf 支持的类型或换算不能通过 gmp_printf 来使用，例如包括 GLIBC register_printf_function 支持的扩展寄存。当前也不能支持 POSIX 的‘$’类型数值参数。

精度区域对整数‘Z’和浮点数‘F’类型来说是通常的意义，但是当前还没有定义‘Q’的精度区域，精度区域是不可用的。

与函数 mpf_get_str 所做的一样，mpf_t 换算只产生能精确表示操作数的位数，如果要求更高精度 0 将作为填充。即使对值为整数的 mpf_t 的‘f’换算，这也可能发生，例如 2^{1024} 对 128 比特精度的 mpf_t 只会产生 40 个十进制位数，然后在十进制小数点之后填充 0。像‘%.Fe’和‘%.Ff’这样的空精度区域可以用来指定只需要使用有效位。

十进制小数点字符 (或字符串) 可以从定义了 localeconv 的当前系统 locale 设定中获得，C 库对标准浮点输出也是如此处理的。

格式字符串只作为普通字符来解释，多字节字符不会被识别。这在未来可能有所改变。

8.2 函数

下面的每个函数都对应相应的 C 库函数。基本的 printf 函数接收变量参数列表，vprintf 接收参数指针。

需要强调的是，如果格式字符串不可用或者参数不能符合指定格式，那么任何这些函数的行为都将是不可预料的。GCC 格式字符串检测将不可用，因为它无法识别 GMP 扩展。

基于文件的函数 gmp_printf 和 gmp_fprintf 返回 -1 显示有错误发生。如果正在使用的 C 库 printf 函数变体返回 -1，那么所有这些函数都将返回 -1，但是通常这都不会发生。

```
int gmp_printf (const char *fmt, ...)                                    [函数]
int gmp_vprintf (const char *fmt, va_list ap)                            [函数]
```

打印到标准输出 stdout。返回所写字符数，若错误发生则返回 -1。

```
int gmp_fprintf (FILE *fp, const char *fmt, ...)                         [函数]
int gmp_vfprintf (FILE *fp, const char *fmt, va_list ap)                 [函数]
```

打印到流 rp。返回所写字符数，若错误发生则返回 -1。

```
int gmp_sprintf (char *buf, const char *fmt, ...)                        [函数]
int gmp_vsprintf (char *buf, const char *fmt, va_list ap)                [函数]
```

在 buf 中建立 0 结束的字符串。返回所写的字符数，不包括 0 结束符。

buf 和串 fmt 的空间不允许有交叠。

不推荐使用这两个函数，因为对 buf 可用空间越界没有保护。

```
int gmp_snprintf (char *buf, size_t size, const char *fmt, ...)          [函数]
int gmp_vsnprintf (char *buf, size_t size, const char *fmt, va_list ap)  [函数]
```

在 buf 中建立 0 结束的字符串，写不超过 $size$ 字节。要得到全输出，$size$ 必须足够大以容纳字符串和 0 结束符。

返回值是可以产生的全部字符数，不包括 0 结束符。如果 $retval \geq size$，那么输出截断为前 $size - 1$ 个字符，并添加 0 结束符。

在区域 $\{buf, size\}$ 与字符串 fmt 之间不允许有交叠。

注意返回值是 ISO C99 snprintf 风格，即使 C 库函数 vsnprintf 是更老的 GLIBC2.0.x 风格亦是如此。

```
int gmp_asprintf (char **pp, const char *fmt, ...)                       [函数]
int gmp_vasprintf (char *pp, const char *fmt, va_list ap)                [函数]
```

在用当前内存分配函数分配的内存块中建立 0 结束的字符串。块的长度为字符串长加上一个 0 结束符。将块的地址置于 $*pp$，返回所产生的字符数，不包括 0 结束符。

与 C 库函数 asprintf 不同，如果没有足够可用空间 gmp_asprintf 不返回 -1，它让当前的内存分配函数来处理之。

```
int gmp_obstack_printf (struct obstack *ob, const char *fmt, ...)       [函数]
int gmp_obstack_vprintf(struct obstack *ob,const char *fmt,va_list ap)  [函数]
```

添加到当前 obstack 对象，使用与 obstack_printf 相同的风格，返回所写字符数，0 结束符不写入。

fmt 不能处于当前 obstack 对象中，因为对象生成后可能被移动。

只有当 C 库提供了 obstack 属性，这些函数才可用，也就是说也许只有在 GNU 系统中才可用。

第 九 章 格式输入

9.1 格式输入字符串

gmp_scanf 及其友函数与标准 C 函数 scanf 一样接收格式字符串。格式规范是下面这种形式：

```
% [flags] [width] [type] conv
```

GMP 为 mpz_t,mpq_t,mpf_t 分别增加类型‘Z’,‘Q’,‘F’，并为 mp_limb_t 数组增加类型‘N’。‘Z’,‘Q’和‘N’的行为像整数一样，‘Q’有必要的话会打印一个‘/’和一个分母。‘F’的行为像浮点数。

传递到 gmp_scanf 时，GMP 变量不需要 &，因为它已经是‘引用调用’。例如

```
/* to read say "a(5) = 1234" */
int n;
mpz_t z;
gmp_scanf ("a(%d) = %Zd\n", &n, z);
mpq_t q1, q2;
gmp_sscanf ("0377 + 0x10/0x11", "%Qi + %Qi", q1, q2);
/* to read say "topleft (1.55,-2.66)" */
mpf_t x, y;
char buf[32];
gmp_scanf ("%31s (%Ff,%Ff)", buf, x, y);
```

所有的标准 C printf 类型的行为与 C 库 printf 相同，可以自由的与 GMP 扩展混合使用。在目前的实现中，格式字符串的标准部分简单的提交给 printf 而只有 GMP 扩展才直接处理。

标志按下面的规范接收。‘a’和‘’’依赖于 C 库的支持，‘’’不能应用于 GMP 类型。

*	读取而不储存
a	分配缓冲区 (字符串换算)
’	组数字，GLIBC 类型 (非 GMP 类型)

标准类型的接收按下面的规范。‘h’和‘l’是方便的，对其他的来说，类型依赖于编译器类型 (或头文件) 而输入依赖于 C 库。

h	short
hh	char
j	intmax_t 或 uintmax_t
l	long 或 wchar_t
ll	long long
L	long double
q	quad_t 或 u_quad_t
t	ptrdiff_t
z	size_t

GMP 类型是：

F	mpf_t，浮点型换算
Q	mpq_t，整型换算
Z	mpz_t，整型换算

换算按下面规范接收。‘p’和‘[’依赖于 C 库的支持，其他的是标准的。

c	字符或字符串
d	十进制整数
e E f g G	浮点数
i	具有基指示器的整数
n	读取迄今为止所有字符
o	八进制整数
P	指针
s	无空位符字符串
u	十进制整型
x X	十六进制整数
[	集合中字符组成的字符串

‘e’, ‘E’ ‘f’, ‘g’和‘G’是相同的，它们都可以读取定点或科学计数法格式浮点数，‘e’和‘E’都可以用于科学计数法格式中指数。

‘x’和‘X’是相同的，都可以接收大写或小写的十六进制数。

‘o’, ‘u’, ‘x’和‘X’都可以读取正值或负值。对于标准 C 类型这些被描述为“无符号”换算，但是很少会影响某些溢出处理，因此负值是允许的。GMP 类型是没有溢出的，所以‘d’和‘u’是相同的。

‘Q’类型读取给定的分子和 (可选的) 分母，如果值不是规范型，那么在对它作任何计算之前需要调用 mpq_canonicalize 来进行规范化。

‘Qi’将对分子和分母按不同基分别读取，例如‘0x10/11’将是 16/11，而‘0x10/0x11’将是 16/17。

‘n’可用于上面所有的类型，甚至是 GMP 类型。允许用‘*’来抑制赋值，但是在区域内不会做任何事情。

其他 C 库函数 scanf 可用的换算或者类型不能用于 gmp_scanf。

除了‘c’和‘[’换算，区域前的空位符在读取后会被立即抛弃。

十进制小数点字符 (或字符串) 可以从定义了 localeconv 的当前系统 locale 设定中获得，C 库对标准浮点输出也是如此处理的。

格式字符串只作为普通字符来解释，多字节字符不会被识别。这在未来可能有所改变。

9.2 格式输入函数

下面的每个函数都对应相应的 C 库函数。基本的 scanf 函数接收变量参数列表，vscanf 接收参数指针。

需要强调的是，如果格式字符串不可用或者参数不能符合指定格式，那么任何这些函数的行为都将是不可预料的。GCC 格式字符串检测将不可用，因为它无法识别 GMP 扩展。

在 fmt 字符串和任何产生的结果之间不允许有交叠。

```
int gmp_scanf (const char *fmt, ...)                                    [函数]
int gmp_vscanf (const char *fmt, va_list ap)                            [函数]
```

从标准输入 stdin 中读数据。

```
int gmp_fscanf (FILE *fp, const char *fmt, ...)                         [函数]
int gmp_vfscanf (FILE *fp, const char *fmt, va_list ap)                 [函数]
```

从流 fp 中读数据。

```
int gmp_sscanf (const char *s, const char *fmt, ...)                    [函数]
int gmp_vsscanf (const char *s, const char *fmt, va_list ap)            [函数]
```

从 0 结束字符串 s 中读数据。

这些函数中的每个函数与标准 C99 scanf 有相同的返回值，也就是成功解析和储存的区域个数。‘%n’一个一个的区域进行读取但是可以被‘*’中止，而且不计算返回值。

如果在需要一个匹配时到达了文件尾部或者文件错误，又或是字符串尾部，如果没有先前中止的区域匹配时，那么返回的是 EOF 而不是 0。匹配指的是在格式字符串或者‘%n’之外的区域中需要一个字母字符。格式字符串中的空位符是可选的匹配，在这种风格下不会导致 EOF。一个区域中读取后立即抛弃的前导空位符不能算作是匹配。

第 十 章 用户内存分配

默认情况下 GMP 使用 malloc、realloc 和 free 来做内存分配，如果它们失败，那么 GMP 打印一条信息到标准错误输出和程序终端。

可选的函数可以指定不同的内存分配方式和内存用尽时不同的错误处理方式。

```
void mp_set_memory_functions (                                          [函数]
            void *(*alloc_func_ptr) (size_t),
            void *(*realloc_func_ptr) (void *, size_t, size_t),
            void (*free_func_ptr) (void *, size_t))
```

用参数所给的函数代替当前使用的内存分配函数，如果其中某个参数为 NULL 那么使用相应的默认函数。

这些函数将会用于所有 GMP 所作的内存分配，如果函数可用那么它要与临时空间以及 GMP 配置使用的 alloc 分离。

必须保证调用 mp_set_memory_functions 时没有正在应用先前的内存分配函数分配内存的 GMP 对象，这通常意味着应该在任何其他 GMP 函数之前调用它。

向这个函数提供的参数应该像下面这样进行声明：

```
void * allocate_function (size_t alloc_size)                            [函数]
```

用至少 *alloc_size* 字节进行空间分配，返回分配的新空间的指针。

```
void * reallocate_function (void *ptr,  size_t old_size,                [函数]
                                        size_t new_size)
```

将先前为 *old_size* 字节的块 *ptr*，重定义分配长度为 *new_size* 字节。

如果有必要或者有希望的话，可以移动这个块，这种情况下，*old_size* 和 *new_size* 中较小者个字节将拷贝到新的地址，返回值是重定义了长度的块，若有所移动则是新的地址否则就是 *ptr*。

ptr 总不会是 NULL，它总是先前已分配的块。*new_size* 可以比 *old_size* 大也可以比它小。

```
void deallocate_function (void *ptr, size_t size)                       [函数]
```

解除由 *ptr* 指向的分配空间。

ptr 总不会是 NULL，它总是先前已分配的 *size* 字节的块。

这里用到的字节 *byte* 是 sizeof 操作符所用的单位。

传递到 reallocate_function 和 deallocate_function 的 *old_size* 参数是出于方便的目的，当然如果没有必要的话，也可以忽略它。默认函数使用 malloc 及其友函数，但有时也可能不使用它们。

不允许从这些函数返回错误，如果它们被返回，那么必须进行指定的处理。特别地 allocate_function 和 reallocate_function 不允许返回 NULL。

获得不同的致命错误操作是用户内存分配函数的一个好用法，例如给出一个图形对话而不是简单的打印到 stderr。但是有多大可能能够用尽实际内存又是另一个问题。

如何从像内存用尽这样的错误中进行恢复，当前还没有定义方法，而是必须停止程序运行。一个 longjmp 或者抛出一个 C++ 异常将会产生不可预料的结果。这在未来将有所改变。

GMP 可能应用分配的块来保存其他分配块的指针。这将限制为只能使用保守的垃圾收集方式。

因为 GMP 默认内存分配使用 malloc 及其友函数，这就造成即使程序一开始做的就是 mp_set_memory_functions，这些函数也是会被链接的。如果认为这是个问题的话，那么必须修改 GMP 的源代码。

第 十一 章 内部结构

本章的目的只是提供一些信息，这里描述的内部结构在 GMP 的未来版本中可能有所改变。希望与未来版本兼容的应用应该只用前面几章描述的接口。

11.1 整数内部结构

mpz_t 变量用符号和数值来表示整数，空间动态分配和重分配。包括下面这些区域：

_mp_size 所表示整数的 limb 数，或者若表示负数时则是 limb 数的负。0 通过设置 _mp_size 为 0 来表示，此时不使用 _mp_d 数据。

_mp_d 一个指针，它指向表示数值的 limb 数组。正如每个 mpn 函数所作的那样，这个数组是以“小结尾”方式存储的，因此 _mp_d[0] 是最低有效 limb，而 _mp_d[ABS(_mp_size)-1] 是最高有效 limb。当 _mp_size 非 0 时，_mp_d 的最高有效 limb 非 0。

当前至少有一 limb 会被分配，因此，mpz_set_ui 永远不需要重分配，而 mpz_get_ui 总可以无条件的得到 _mp_d[0](尽管只有在 _mp_size 非 0 时它的值才可能有用)。

_mp_alloc 当前分配在 _mp_d 的 limb 数，自然地有 _mp_alloc >= ABS(_mp_size)。当 mpz 例程要 (或将要) 增加 _mp_size，它会检测 _mp_alloc 看是否有足够的空间，若不够，则重分配。MPZ_REALLOC 通常用来做这件事情。

很多像 mpz_and 这样的位逻辑运算函数把负值作为 2 的补码来处理，但是符号和数值是在内部使用的，而在计算时需要做必要的调整。有时这并不是很好的，但是符号和数值对其他例程是最好的。

一些内部的临时变量通过 MPZ_TMP_INIT 来建立，它们通过 TMP_ALLOC 来获得 _mp_d 空间，而不是内存分配函数。应该注意保证有足够大的空间以避免进行重分配。

11.2 有理数内部结构

mpq_t 变量用 mpz_t 分子和分母。

规范形式调整分母为正的 (且非 0)，分子分母无公因子，0 唯一的表示为 0/1。

通常认为在计算的每一步约去公因子是最好的办法，一次 GCD 需要 $O(N^2)$ 操作，所以对小的整数作几次即时的 GCD 总比到后来对大的做一次 GCD 来的好一些。知道分子分母没有公因子可用于如 mpz_mul 这样的函数，使得只需两次 GCD 而不是四次。

对于存在简单因式分解或能快速约去的情况，这种方式是极其非优化的，但是 GMP 无法知道这会在什么时候出现，这需要应用自己去处理。mpq_numref 和 mpq_denref 是分子和分母的直接引用，当然 mpz_t 变量可以直接被使用，因此 mpq_t 的这种框架仍然是比较合理的。

11.3 浮点数内部结构

有效的计算是 GMP 浮点数的主要目标，全 limb 的使用和简单的例程使之能方便实现。

mpf_t 浮点数有一个可变精度的尾数和有符号单精度机器字指数。尾数用符号和数值来表示。包括以下区域：

_mp_size 当前使用的 limb 数，或者若表示负数时则是 limb 数的负。同时置 _mp_size 和 _mp_exp 为 0 以表示 0，在这种情况下 _mp_d 不使用。(在未来的版本中，可能不再定义 _mp_exp)。

_mp_prec 尾数的精度，以 limbs 为单位。在任何计算中目的都是产生结果的 _mp_prec 个 limb(最高有效 limb 非 0)。

_mp_d 一个指针，它指向表示尾数的 limb 数组。正如前面的 mpn 函数所作的那样，这个数组是以“小结尾”方式存储的，因此 _mp_d[0] 是最低有效 limb，而 _mp_d[ABS(_mp_size)-1] 是最高有效 limb。
最高有效 limb 是非 0 的，但是对它的值没有其他的约束，特别是最高的 1 比特可以是最高有效 limb 的任意一比特。
_mp_prec+1 个 limbs 会分配到 _mp_d，额外一 limb 是出于方便的目的 (参考下面)。在计算中不会有重分配，只会有通过 mpf_set_prec 对精度的改变。

_mp_exp 指数，以 limb 为单位，确定隐藏的小数点的位置。0 表示小数点正好在最高有效 limb 之前。正值表示向更低 limb 的位移，因此值应该 ≥ 1。负值表示小数点远远高于最高有效 limb。
自然地，指数可以是任意的值。它不必处于 limbs 的内部，可以远远高于或远远低于。不包含在 {_mp_d,_mp_size} 的 limbs 被看作是 0。

应该注意下面的要点：

低位的 0 最低有效 limb_mp_d[0] 等可能为 0，但是这些低位的 0 可以被忽略。例程可能作低位 0 检测并避免它们以节约后续计算的时间，但是在大多数例程中不会实现这样的检测。

尾数长度范围

如果数值可以用更少的 limb 表示，那么正被使用的 limb 量 _mp_size 可能比 _mp_prec 要小。这就意味着更低的精度值和更小的整数值存储于高精度的 mpf_t 仍然可以进行有效的操作。
_mp_size 也可以比 _mp_prec 大，首先允许一个数用完 _mp_d 的所有 _mp_prec+1 个 limb，其次，函数 mpf_set_prec_raw 减小 _mp_prec 而保持 _mp_size 不变，因此长度可以比 _mp_prec 任意的大。

舍入 所有的舍入都在 limb 边界进行。通过最高非 0limb 计算 _mp_preclimb 将保证应用所需要的最小精度可以获得。

简单的向 0 截尾舍入用起来是很高效的，因为无需检查额外的 limb 以及增减 limb。

移位运算 因为指数是以 limb 为单位的，所以在基本运算中没有像 mpf_add 和 mpf_mul 等函数那样的移位运算函数。当遇到要改变指数时，需要做的只是修改指针使之指向相应的 limb。

当然 mpf_mul_2exp 和 mpf_div_2exp 可能需要移位运算，但是可以选择在此处需要移位的以 limb 为单位的指数，或者在任何其他地方都需要移位运算的以比特为单位的指数。

_mp_prec+1limbs 的使用

_mp_d 的额外一比特 (_mp_prec+1 而不是 _mp_prec 比特) 对可能从它的操作中得到进位的 mpf 例程很有帮助。例如 mpf_add 会对 _mp_preclimbs 作 mpn_add，如果没有进位那么它就是结果，而如果有进位，那么它会被存放在额外的 limb 空间中，而此时 _mp_size 变成 _mp_prec+1。
当一个变量中存放了 _mp_prec+1 个 limb 时，对预期的精度来说，不需要低 limb，而只需要高 _mp_prec 个 limb。但是把它归 0 或者将高位部分移下也是没有必要的。后续的计算读值时将简单的提取它们所需要的高 limbs，如果目标有相同的精度的话，那么这个量也会是 _mp_prec。这将不会需要一个以上指针的调整，而且必须在任何地方都进行检测，因为目的精度可能与源精度是不同的。

如果可用的话，像 mpf_set 这样的拷贝函数将保留完全的 _mp_prec+1 个 limb。这保证了满足 _mp_size 等于 _mp_prec+1 的变量可以进行准确的全值拷贝。严格来说这是没有必要的，因为应用所需精度只要求 _mp_prec 个 limb，但是它考虑的是从一个变量到另一个相同精度的变量的 mpf_set 应该会产生一个额外的拷贝。

__GMPF_BITS_TO_PREC 将应用所需精度转换为 _mp_prec，以比特为单位的值将舍入为一个全 limb，然后会增加一个额外的 limb，因为 _mp_d 的最高有效 limb 可能只是非 0，而且可能只包含一比特。

__GMPF_PREC_TO_BITS 做相反的转换，在把它转换为比特之前先将额外的一 limb 移除。用 mpf_get_prec 回读的净效果是将精度简单地舍入为一个 mp_bits_per_limb 的倍数。

应用精度

需要注意的是，这里增加到高位的额外一非 0limb 是另外分配到 _mp_d 的额外 limb。例如对 32 比特的 limb，需要 250 比特的应用将舍入为 8limb，然后将非 0 的额外 limb 加到高位，并置 _mp_prec 为 9，然后 _mp_d 得到 10limbs 的分配空间。用 mpf_get_prec 回读将得到 _mp_prec-1limb，乘上 32 比特，也就是 256 比特。

严格的说，实际上高 limb 至少有一比特，这就意味着有 3 个 32 比特 limb 的浮点数至少有 65 比特，但是为了方便 mpf_t，一般简单的认为是 64 比特，刚好是 limb 长度的一个倍数。

11.4 Raw 输出内部结构

mpz_out_raw 输出使用下面的格式。

长度	数据字节

用 4 字节表示的长度是后续数据的字节数或者表示负数时是字节数的 2 的补码，首先写的是它的最高字节。数据字节是整数的绝对值，从最高有效字节写起。

最高有效字节总是非 0 的，所以在所有系统中都是相同的，与 limb 的比特长度无关。

在 GMP 1 中，前导的 0 字节会写到数据字节中以填充为整 limb 长度。为了兼容性，mpz_inp_raw 也可接受这种结构。

在长度和数据字节中都采用“大结尾”是经过深思熟虑的，这使得数据更容易在文件的十六进制堆中读取。不幸的是它也意味着 limb 数据在读和写时必须反向，所以无论是“大结尾”还是“小结尾”都不能从 _mp_d 读或写。

参考文献

[1] Mike Loukides & Andy Oram, Programming with GNU SoftWare, O'REILLY. 石祥生 瞿炯 石秋生 译 《用 GNU 软件编程》，电子工业出版社，1997.