第一章　Stata 统计软件使用入门

Stata 是在统计学、计量经济学、社会学、政治学等领域非常常用的统计软件，其名字是英文单词 Statistics and data 的缩写。Stata 不仅提供了统计学与计量经济学最常用的模型，还提供了非常方便的数据管理工具，极大的减轻了统计应用的工作量。除此之外，Stata 还提供了强大的编程功能，用户不仅可以使用 Stata 编写脚本，还可以编写自己的 Stata 命令等。此外，Stata 还自带 Mata 语言——一个类似于 Matlab 的矩阵运算语言，极大的扩展了 Stata 的可编程能力。在这节中我们就简单介绍一下 Stata 的使用方法。

1.1　Stata 介绍

Stata 为商业软件，因而必须购买才能使用。当然，如果学校或者组织已经购买，可以直接联系学校相关部门索取使用权。一般来说，更新版本的 Stata 提供了更快的速度以及更多的功能，因而推荐尽量使用更高版本的 Stata，截至目前（2020 年 3 月），最高版本为 Stata 16。由于 Stata 14 之后使用了 Unicode 编码，解决了跨系统时的乱码问题，因而推荐尽量使用 Stata 14 之后的版本，避免多人合作时出现乱码问题。

在支持的操作系统方面，Stata 几乎支持市面上所有主流的桌面操作系统，如 Windows、Ubuntu（Linux）、MacOS 等。此外，同一版本号的 Stata 还区分不同的版本，一般分为 IC 版、SE 版和 MP 版本，版本之间的首要区别在于能够处理的数据量的不同，比如 IC 版本最多只能支持 2048 个变量，SE 版本支持 32767 个变量，而 MP 版本多支持多达 120000 个变量。此外，MP 版本还支持多核并行运算，因而大大提高了运算速度。

1.1.1　Stata 用户界面

在安装好 Stata 之后，直接点击 Stata 图标可以进入 Stata 的图形用户界面，在 Linux 中，可以通过运行 Stata 安装目录的 xstata（或者 xstata-se、xstata-mp）打开 Stata 的图形用户界面[1]。

图 (1.1) 是在 Ubuntu 下 Stata 的界面，主要包括了菜单栏、工具栏、结果

[1]在 Linux 中，stata、stata-se、stata-mp 三个程序文件可以用来在终端中打开无图形用户界面的 Stata。

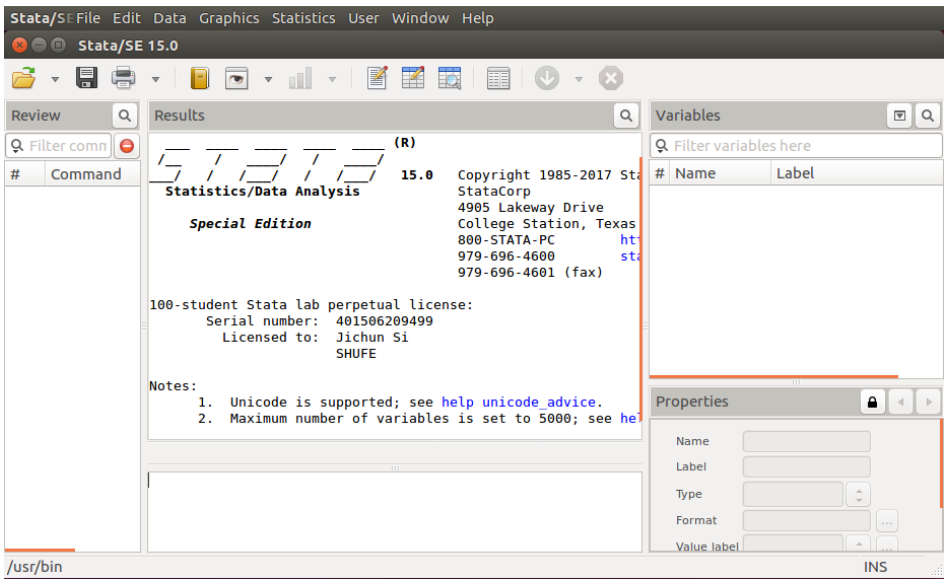

图 1.1: Stata 窗口

(Results)、变量（Variables)、变量属性（Properties)、命令历史（Review）和命令输入框以及最下方显示当前文件夹的状态栏:

- 菜单栏：包括常见的文件、编辑、窗口和帮助等菜单，此外，还有：数据（Data）菜单，包括了数据整理的常用功能；画图（Graphics）菜单，包括了各种画图功能；统计（Statistics）菜单，包括了 Stata 提供的统计分析功能。
- 工具栏：包括常用的打开、保存、打印等功能，此外，还有日志（Log)、Do-文件编辑器（Do-file Editor)、数据编辑器（Data editor)、数据浏览器（Data brwoser)、和变量管理器。其具体功能我们将在下面介绍。
- 结果（Results)：命令运行的结果将显示在此，右键单击选择 Preferences 可以更改显示风格。
- 变量（Variables)：显示了内存中数据的变量名（Name）及其标签（Label)，其中标签是变量的具体描述，我们将在下面介绍。双击某个具体的变量可以快速将该变量名填在命令输入框中。
- 变量属性（Properties)：选中的变量的属性，如数据类型等。
- 命令历史（Review)：在 Stata 中所有执行的命令历史都显示在这里。即使使用菜单等鼠标操作，在执行时也会转化为 Stata 命令执行，同时显示在命令历史中。单击某一条具体的历史可以快速将命令填在命令输入框中，双击会直接运行选中的命令。

- 命令输入框 (Command): 即光标的位置，由于所有的 Stata 操作都可以通过命令方式完成，因而该输入框在 Stata 的使用中起着至关重要的作用。

尽管多数操作都可以使用菜单和鼠标完成，然而我们仍然强烈推荐使用命令的方式操作 Stata。

小试牛刀，我们接下来试着使用 Stata 打开一个数据文件并做一些简单的描述性统计。单击文件 (File) 菜单，找到「Open」菜单项，在弹出的对话框中找到「cfps_adult.dta」[2]，点击「Open」，在「Variables」窗格中就看到了很多诸如「pid, fid14, provid14, ...」等的「变量」，如此我们就打开了这个数据文件。接下来点击「Statistics」菜单中的「Summaries, tables and tests -> Summary and descriptive statistics -> Summary statistics」，弹出一个对话框，从「Variables」中选择「p_income」这个变量，确认，在「Results」窗口中会显示一个表格，列出了「p_income」这个变量的观测数、均值、标准差、最小值、最大值等统计量。

以上过程比较繁琐。注意到在「Results」窗口中，描述性统计表格之前有一行代码，即:「summarize p_income」，这就是我们刚刚计算描述性统计操作的内部执行的命令。实际上，以上打开数据、进行描述性统计也完全可以使用命令来完成。在「Command」窗口中输入:

```
su p_income
```

会得到同样的结果。

1.1.2 Stata 中的命令

在 Stata 中使用命令是非常简单的。比如当我们打开 Stata 时，会提醒我们「Maximum number of variables is set to 5000; see help set_maxvar.」，即变量个数上限被设置为了 5000，可以使用 set maxvar 来修改变量个数的限制，比如可以在命令输入框中使用:

```
set maxvar 10000
```

将变量最大个数设置为 10000。此外，该提醒还告诉我们，如果对于 Stata 命令有任何的疑问，可以使用 help 命令获得帮助，比如:

```
help set
```

命令就可以得到关于 set 命令的所有使用帮助。Stata 的 help 命令为每个命令都提供了非常详尽的解释说明，如果对于某个命令的使用有疑问，可以直接使用「help」得到帮助。此外，如果需要更加详细的命令使用帮助，还可以在 Stata

[2]本讲义中的数据文件可以在https://github.com/sijichun/MathStatsCode中找到，接下来，我们将使用北京大学中国家庭跟踪调查（China Family Panel Studies, CFPS）2014 年部分数据和变量作为示例。

安装文件夹下的「docs」文件夹下面找到非常完整的 Stata 使用手册[3]，是 Stata 最权威、最全面的使用文档。

从 help set 的结果中我们可以看到，set 命令有非常多的用法。比如在执行某项统计分析之后，很有可能会出现结果太长在结果窗口中一屏幕不能完全显示的情况，此时程序会终止，并显示「—more—」，需要手动点击才能继续程序的运行。如果希望程序不要中断，将所有结果一并显示出来，可以使用 set more off 命令。此外，可以使用 set matsize 设定矩阵的最大的规模，当 Stata 出现「matsize too small」的时候可以使用该命令；在老一点的 Stata 版本中，可能还需要使用 set max_memory 设置 Stata 可使用的最大内存数量。

Stata 具有种类繁多的命令，然而这些命令有一定的规律，一般来说，Stata 的命令具有如下格式：

```
1 [prefix:]command [varlist] [=exp] [if] [in] [weight]
2                  [using filename] [, options]
```

其中方括号代表可选项，以上语法可以在「help language」中找到。各个部分含义如下：

- prefix：前缀，为可选项，通常用来对命令进行一些必要的修饰，前缀语句结束后一定有一个冒号以示区分。比如如果在一个命令前面加「quietly:」，那么该命令运行将不会在结果窗口中显示任何结果；而如果加入「noisily:」，将会强制显示结果。
- command：命令的名称，如上面已经介绍的 set 命令、help 命令等，以及用于描述性统计的 summarize、用于回归的 regress 命令等。
- varlist：需要进行操作的变量列表。
- =exp：表达式，将在下面具体介绍。
- if：某个命令可能不需要对所有的观测进行操作，可以使用 if 子句挑出那些需要被操作的观测。
- in：同样是挑选观测进行操作，只是挑选标准为变量的排序（行号）。比如「br in 1/10」仅仅显示前十个观测。
- weight：权重，在 Stata 中，默认所有的观测都是等权重的，然而由于抽样等原因，每个观测的权重可能并非相等，此时可以使用 weight 子句指明权重，Stata 支持 fwight, pweight, aweight, iweight 等四种权重，具体说明可以在「help weight」中查看。
- using filename：指定需要操作的文件名。

[3]其中 i.pdf 提供了详细的索引。

- options：命令的选项，跟在一个英文逗号后面，在多数命令中为可选项，控制着命令运行的各种细节。当有多个选项时，只需要一个逗号即可，不同选项之间用空格隔开。

我们将在接下来的学习中不断用到上述命令结构。

此外，除了 Stata 自带的命令以外，还有大量其他非 Stata 自带的、用户编写的命令可以使用。在使用这些命令之前，需要首先安装这些命令。安装命令一般有如下两种方式：

- ssc 安装：直接输入「ssc install 命令名」即可。比如，我们经常会使用 outreg2 命令导出统计分析结果，而这一命令不是 Stata 自带命令，我们可以直接使用「ssc install outreg2」安装这一命令。
- help 安装：直接输入「help 命令名」，Stata 会自动搜索相关的命令，并返回在《Stata Journal》中相关的结果。可以在弹出的窗口中找到「st****」(* 代表数字)，点击进入相应界面，点击「click here to install」安装。

在比较老的版本的 Stata 中，以上 help 命令可能需要换成 findit (search) 命令。

最后，很多 Stata 命令还有缩写的形式，一般为命令名称的前几位字母。比如描述性统计「summarize」命令可以直接使用「su」或者「sum」替代，回归命令「regress」可以用「reg」代替，「browse」可以用「br」代替等等。在命令的 help 中，如果命令（或者选项）有一部分字母加下划线，那么下划线部分就是该命令或者选项的缩写。

1.2 文件与变量

1.2.1 文件

Stata 有所谓的「工作目录」的概念，比如在图 (1.1) 中，状态栏中的「/usr/bin」代表当前工作目录为「/usr/bin」。设定了工作目录之后，如果接下来不指明路径，那么所有的打开文件、保存文件操作都默认在这个文件夹中，或者以这个文件夹为基准。可以使用「cd」命令改变工作目录，比如使用命令「cd "/home/sijichun"」即将工作文件夹修改到「/home/sijichun」目录下，Windows 下类似使用，比如「cd "d:\sijichun\"」。

此外，Stata 还支持 Linux/Unix/Windows 相似的基本文件命令，比如可以使用「pwd」命令获取当前工作目录，使用「ls」或者「dir」命令显示工作目录的文件和目录，使用「rm」或者「erase」命令删除文件。在接下来的介绍中，我们假设在当前目录下可以找到所需要的所有文件（比如数据文件.dta 或者.do 文件），如果提示文件找不到，请自行使用 cd 命令改变路径。

Stata 数据文件以.dta 作为后缀名，我们可以使用「use」命令打开数据文件。比如，如果数据文件「cfps_adult.dta」保存在「d:\sijichun\」目录下，可以使用如下命令：

```
cd "d:\sijichun\"
use "cfps_adult.dta", clear
```

打开数据集「cfps_adult.dta」。尽管多数情况下文件名「cfps_adult.dta」可以不加双引号，然而加双引号是非常良好的习惯。此外，后缀名.dta 也可以省略。

如果在打开数据之前，Stata 内存中已经有数据，那么以上 use 命令会报错。解决以上问题可以有两种方法：

- 使用 clear 命令，清除内存中的所有数据；
- 为 use 命令加入 clear 选项，即：「use cfps_adult, clear」。

注意以上命令将会清除内存中的所有数据，所以在使用以上命令时，如果内存中的数据仍然有用，一定要保存数据，保存数据可以使用 save 命令，比如「save "adult.dta"」，同样，引号和.dta 可以省略。如果文件已经存在，save 命令仍然会报错，需要在 save 命令后面加入 replace 选项。此外，如果使用 use 命令打开某个数据集，进行一定操作之后，希望覆盖原始数据，可以忽略文件名，直接使用「save, replace」即可。当然在实际操作中，由于覆盖是一个不可逆操作，因而建议在任何情况下不要覆盖原始数据。

打开数据文件后，可以通过点击工具栏上的数据浏览器查看数据，或者直接使用「brwose」命令打开数据浏览器查看数据。此外，还可以在 browse 命令后面加 if 或者 in 子句，比如命令「br if provcd14==11」即可查看所有北京市家庭的数据，而命令「br in 1/10」可以查看前 10 行的数据。

在默认条件下，为了保证数据的安全性，数据浏览器中是不可以修改数据的。如果希望单独修改数据，需要在工具栏或者数据浏览器的工具栏中点击数据编辑器进行编辑。或者，可以使用「edit」命令打开数据编辑器。

Stata 还支持从多种数据源中导入、导出数据，包括 Excel、CSV、SAS 格式等。可以从「File 菜单-import」中打开导入数据的对话框，选择需要导入的文件名，以及相关的选项即可导入数据。在 Excel 或者文本文件（delimited text data）中，可以选择直接使用数据的第一行作为变量名；在导入文本文件时，可以选择编码（encoding）避免乱码问题。当然，以上所有导入导出操作也可以使用 import/export 命令完成，详细选项可以查看 help。

值得一提的是，如果导入的数据在数据浏览器中显示为褐色，说明该列数据被导入成为字符串。如果认为该列变量应该为数值型，可以使用「destring」命令将字符型转化为数值型，具体选项可以查看 help。当然，如果一旦出现这个问题，在使用 destring 之前，检查哪些数据中可能出现了被认为是字符串的字符是非常有必要的。

1.2.2 变量

Stata 中的数据是以「变量（variables）」为基本单位的，一个变量就是数据表中的一列。比如，当我们打开数据「cfps_adult.dta」之后，我们可以在「变量」

窗口中看到「pid, fid14, provcd14,...」等变量名，而打开数据浏览器，数据的列名称即为变量名。此外，在变量窗口中每个变量都对应着一个「标签（label）」，用来具体描述变量的含义。

在 Stata 中我们可以使用「generate」命令产生新的变量，该命令基本语法为：

```
gen [type] newvar =exp [if] [in]
```

其中：

- type 为数据类型，为可选项。Stata 支持包括整型（int）、长整型（long）、浮点型（float）、双精度型（double）以及字符串等类型，具体可以使用「help data types」查看 Stata 支持的数据类型的完整说明。
- newvar 为新的变量名，最长为 32 个字符，以字符或者下划线（_）开头，不能以数字开头，不能为保留关键字。实际上由于新版的 Stata 支持 Unicode 编码，因而中文也可以被用作变量名，然而出于可移植性等原因，并不建议使用中文变量名。变量名可以使用「rename」命令修改。
- exp 为表达式，通常为其他变量的算数运算或者函数，常见的算数运算符，比如加（+）、减（-）、乘（×）、除（/）、幂运算（^）等都被支持。

比如，以下代码就产生了个人总收入中除工作收入之外的其他收入：

```
use "cfps_adult.dta"
gen other_income=p_income-qg12
```

值得注意的是，在以上 gen 命令运行结束之后，Stata 提示「(61 missing values generated)」，即除正常数据以外，产生了 61 个缺失值（missing values）。在 Stata 中，缺失值以一个英文句号「.」来表示，我们可以使用「br if other_income==.」来查看具体哪些观测其新生成的变量为缺失值。或者，也可以使用：

```
br other_income p_income qg12 if other_income==.
```

只显示 other_income 变量为缺失值的 other_income p_income qg12 三个变量。

在上面的「br」命令中，我们使用了「if」子句。正如前面介绍的，「if」子句用于挑出那些我们希望进行操作的观测，其后面跟着逻辑语句。在 Stata 中，支持常见的关系运算符如大于（>）、小于（<）、大于等于（>=）、小于等于（<=）、等于（==）、不等于（!=）以及逻辑运算符，如逻辑与（&）、逻辑或（|）、逻辑非（!）。例如，以下命令：

```
gen post90=1 if cfps_birthy>=1990
```

创建一个新的变量 post90，对于 90 后，post90=1，否则为缺失值「.」。而如果想要生成一个变量，比如对于 90 后 post90s=1，否则等于 0，那么可以直接使用以下命令：

```
gen post90s=cfps_birthy>=1990
```

即如果表达式为逻辑语句，那么当某个观测判断为真时 =1，否则 =0。

需要注意的是，在 Stata 中，缺失值「.」是一个比任何实数都大的值，因而如果使用「sort」命令对刚刚生成的 post90 变量进行升序排序：

```
sort post90
br post90
```

在弹出来的数据浏览器中，我们会发现所有的缺失值都排在了后面。因而实际上如果 cfps_birthy 含有缺失值，那么其对应的新产生的 post90/post90s 都等于 1，而非缺失值。这就会导致可能会有一些观测，由于 cfps_birthy 值缺失，而使用以上代码生成 90 后的虚拟变量时，把这些观测误认为是 90 后。因而在产生以上变量之前，一定要先检查 cfps_birthy 值是否有缺失的情况。

除了使用算数运算、逻辑运算作为 gen 命令的表达式之外，Stata 还提供了大量的函数以供使用，比如常见的取绝对值（abs()）、对数函数（log()）、指数函数（exp()）、开平方（sqrt()）、取整函数（int()）、向上取整（ceil()）、向下取整（floor()）、取余函数（mod()）、logit 函数（logit()）等等，此外，还有大量的日期函数、统计函数、三角函数、字符串函数以及随机数生成函数等等，可以使用「help functions」查询 Stata 所支持的函数。

需要注意的是，使用函数也有可能产生缺失值的情况。比如自然对数 log() 其定义域为 $(0, \infty)$，如果对 0 或者小于 0 的数取对数，就会产生缺失值。取对数在数据分析中是非常常见的操作，所以在取对数时一定要注意检查数据的取值范围，按照接下来的描述性统计的做法检查数据是非常有必要的。

最后，Stata 的所有系统和用户定义的标量都可以用在 gen 语句的表达式中。在 Stata 中，支持如下几个系统变量：

- _n：观测的行号
- _N：所有观测的个数（行数）
- _cons：常数，总是等于 1
- _pi：圆周率 π
- _rc：最近一次 capture 命令捕获的错误代码
- _b[varname]：最近估计的统计模型中，变量 varname 的系数
- _se[varname]：最近估计的统计模型中，变量 varname 的标准误

比如，如果我们需要生成一个新的变量，其值为行号，那么我们只需要使用命令：

```
gen id=_n
```

如此便生成了一个新的变量 id，其值为观测在数据浏览器中的行号。其他系统变量作为标量或者字符串，都可以使用「display」命令，比如「di _pi」将显示 3.1415927，而「di _N」将会显示目前内存中的观测数。

以上介绍了生成数据，我们还可以使用「replace」命令来修改数据，其语法与 gen 命令一样。比如，我们可以结合 gen 和 replace 命令生成 90 后的虚拟变量：

```
gen post90=1 if cfps_birthy>=1990
replace post90=0 if post90==.
```

由于第一条 gen 命令对于非 90 后产生了缺失值，因而第二条使用 replace 命令将缺失值修改为 1。以上两条语句与：

```
gen post90s=cfps_birthy>=1990
```

是等价的。同样，以上两种方法都要注意 cfps_birthy 值缺失的问题，即我们可能会错误地将 cfps_birthy 值缺失的观测误认为是 90 后。

作为示例，假设我们希望生成 100 个观测，分别是 50 个个体在 $t = 0, 1$ 两期的对数收入，假设第 0 期的对数收入 $x_{i1} \sim N(10, 5)$，第 1 期的对数收入 $x_{i2} \sim N(11, 6)$，那么生成以上伪数据可以使用如下代码：

```
clear
set obs 100
gen id=ceil(_n/2)
gen time=mod(_n-1,2)
gen x=rnormal()*sqrt(5)+10 if time==0
replace x=rnormal()*sqrt(6)+11 if x==.
```

其中我们首先使用 clear 命令清除所有数据；由于现在内存中没有任何数据，因而我们必须使用 set obs 设定需要产生的随机数的个数，即总的观测数；接下来我们使用 ceil() 向上取整函数，得到了行号除以 2 的向上取整，作为个人的 id，同时使用行号-1 除以 2 的余数作为时间；最后，我们使用了 rnormal() 函数产生标准正态分布，并变换到我们需要的正态分布。

由于变量名的诸多限制，以及为了编写代码时的方便，一般变量名不会是数据的完整描述。特别是在一些调查数据中，变量名通常是被一些代码所替代。此时我们可以使用变量的「标签 (label)」对变量添加描述。我们可以使用「label variable」命令对变量添加标签，比如，以下代码为刚刚生成的 x 添加了标签：

```
label variable x "随机生成的对数收入"
```

这样我们就可以在变量列表中看到变量的解释说明了。值得注意的是，标签虽然是对变量的解释说明，但也不宜太长，因为在画图的时候，默认的标题是标签，在没有标签的情况下使用变量名，如果变量的标签太长会导致画图时的坐标轴标题太长。

除了可以对变量名加标签之外，还可以对数据的值加标签。对于分类数据，我们在 Stata 中经常以数值进行存储，比如对于「性别」这个变量，我们经常会使用 0 代表女性，1 代表男性；对于「省份」这个变量，我们经常会使用 11 代表北京，31 代表上海，41 代表河南，37 代表山东等等。尽管使用数字代替分类变量在处理时非常方便，但是不便于人们阅读，为了解决这个问题，可以使用「label」命令为数据加标签，如以下代码为上述生成的 post90 变量添加了标签：

```
gen post90s=cfps_birthy>=1990
label def lab_post90 0 "90前"
label def lab_post90 1 "90后", add
label values post90s lab_post90
```

我们首先使用「label define」命令定义了一个标签，叫做「lab_post90」，并将 0 定义为「90 前」，1 定义为「90 后」，进而使用「label values」命令将刚刚定义的标签「lab_post90」附加给变量「post90s」。如此我们打开数据浏览器，「br post90s」，就可以看到这个变量显示的是蓝色的「90 前」或者「90 后」的标签，而非数字。注意数据的标签显示为蓝色，而字符串为褐色，标签仅仅是为了显示的便利，变量的值仍然是 0 或者 1。可以通过「h la」查看 label 的其他用法。

以上介绍了生成变量，与之相对应的是删除变量或者观测。在 Stata 中，可以使用 drop 和 keep 两个命令完成对变量或者观测的删除。其中「drop varlist」代表删除「varlist」中的变量，而「keep varlist」代表只保留「varlist」中的变量，其他变量全都删除。如果想要删除某些观测而非变量，可以在后面加 if 或者 in 子句，比如「drop if」就删除了满足条件的所有观测，而「keep if」则保留了满足条件的所有观测。比如在 cfps_adult.dta 中使用：

```
drop if te4==-8 | te4==-1
```

就删除了所有最高学历为「不适用」或者「不知道」的观测。当然，以上语句等价于：

```
keep if te>0 & te4!=.
```

即只保留 te4 为正值且不缺失的观测。

最后，我们还可以使用「varname[n]」来单独读取变量「varname」的第 n 个观测的值，比如「x[1]」就是变量 x 的第一个观测，而「x[_N]」就是变量 x 的最后一个观测。

以上特性经常被用来生成滞后项。在 Stata 中，我们可以通过「tsset」命令设定时间序列，其语法为：

```
tsset timevar
```

其中 timevar 为代表时间的变量，在数据中必须是唯一的[4]。在定义了时间变量

[4]与之相对应的还有 xtset，即定义面板数据，在面板数据中要求每个个体内部时间变量是唯一的。

之后，我们就可以使用所谓的滞后和向前算子（L. 和 F.）来生成滞后变量，以下程序展示了使用两种方法产生滞后和向前变量：

```
clear
set obs 100
gen t=_n
gen x=rnormal()
drop if t==50
gen x_lag_2=x[_n-1]
gen x_fwd_2=x[_n+1]
gen x_lag2_2=x[_n-2]
tsset t
gen x_lag=L.x
gen x_fwd=F.x
gen x_lag2=L2.x
order t x x_lag_2 x_lag x_fwd_2 x_fwd x_lag2_2 x_lag2
```

首先我们产生了一个唯一可识别的变量 t，以及一个随机的 x。接下来我们使用了两种方法产生 x 的滞后和向前变量，分别通过方括号实现和滞后、向前算子实现，最后使用「order」命令改变这些变量的显示顺序以方便比较。对于 t=1 的观测，由于其滞后项为 t=0 或者 _n=0，在数据集中没有 t=0 或者 _n=0 的观测，因而不管用什么方法，产生的 x_lag(_2) 的第一个观测都是缺失值，x_fwd(_2) 的最后一个观测同理也为缺失值。

值得注意的是，尽管以上两种方法产生的滞后和向前变量在大多数情况下都是相等的，但是注意到在以上程序中，我们删除了 t=50 的观测。如果使用第一种方法，即方括号的方法，其产生新变量完全根据行号（_n），删除 t=50 的观测，那么第 50 行就变成了 t=51 的观测，其 _n-1 为 t=49 的观测，因而 x_lag_2 在第 50 行的值（t=51）就是 x 第 49 行的值；而如果使用第二种方法，由于 t=51 的滞后为 t=50，但是我们删除了 t=50 的观测，所以 x_lag 在第 50 行的值（t=51）为缺失值。

1.2.3 数据框

在 Stata16 中首次加入了对数据框（Data frames）的支持，使用数据框我们可以很方便地同时操作多个数据集而不用频繁打开、关闭文件，或者频繁使用 preserve/restore 操作。由于数据框的数据常驻内存，而频繁打开数据文件需要频繁读取甚至写入硬盘，因而在可能的情况下，特别是数据量大且内存足够的情况下，优先使用数据框可以带来性能上的改进。

在打开 Stata 之后，Stata 默认创建了一个名叫「default」的数据库，为了查看现在内存有哪些数据框，可以直接使用「frames dir」命令。

如果需要新建数据框，可以使用「frames create」命令，比如：

```
frames create sub_data
```

命令就创建了一个名字叫做「sub_data」的数据框。此时如果使用「frames dir」命令会发现该数据框是空的，没有任何数据。如果需要在创建数据框时在这个数据框新建变量，可以直接将变量名写在后面，比如：

```
frames create sub_data1 a b
```

在创建新的数据框 sub_data1 的同时，在这个数据框里面新建了两个变量 a 和 b。如果需要往这个数据库中添加数据，可以直接使用「frames post」命令，注意数据需要用括号包括：

```
frames post sub_data1 (1101) (1999)
```

不同的数据框之间可以使用「frames change」命令进行切换，切换之后执行的操作都是对当前打开的数据框进行的。比如我们刚刚创建了「sub_data」数据框，但是其中没有数据，我们可以使用如下命令切换到该数据框并在该数据框中打开数据文件：

```
frames change sub_data
use datasets/cfps_family_econ
```

可以使用「frames dir」命令查看当前打开的数据框是哪一个。

或者，可以使用「frame」前缀指定数据框进行操作，比如：

```
frames change default
frame sub_data1: use datasets/cfps_family_econ, clear
frame sub_data1: gen log_income=log(fincome1)
```

以上代码先将当前数据框转换为 default，然后在不切换数据框的情况下，在 sub_data1 数据框中打开了一个数据集，并对 sub_data1 数据框中的 fincome1 变量进行了描述性统计。

如果需要拷贝整个数据集，可以使用「frame copy」命令，比如：

```
frame copy sub_data new_sub_data
```

可以将「sub_data」数据库完整拷贝到「new_sub_data」数据框。

如果需要拷贝一个数据集的部分变量或者部分观测到新的数据框，可以使用「frames put」命令：

```
use datasets/cfps_family_econ
frame put fid14-ft1, into(family_subvar) // 拷贝部分变量
frame put if provcd14==11, into(family_subobs) // 拷贝部分观测
```

最后，如果需要删除数据库，可以使用「frames drop」命令：

```
frame drop sub_data
```

或者，可以使用「clear frames」命令清楚内存中所有的数据框。当然，也可以使用「clear all」命令清楚包括所有数据库在内的所有 Stata 对象。

1.3 描述性统计

描述性统计是进行数据分析所必须的先行步骤，包括使用各种描述性统计量制作的表、图等对数据进行探索性的分析。Stata 提供了丰富的描述性统计工具，我们将简要介绍一下使用 Stata 做描述性统计最常用的工具。

1.3.1 连续变量或者虚拟变量

在 Stata 中做描述性统计最常用的命令就是「summarize」，该命令提供了多数常用的描述性统计量，比如最常用的观测数（Obs）、均值（Mean）、标准差（Std. Dev.）、最小值（Min）、最大值（Max）等。除此之外，在 su 命令后面加上「detail」选项，可以汇报更加详细的描述性统计量，比如一些重要的分位数、偏度、峰度等等。

如果 su 命令后面不加任何变量，那么默认会汇报内存中所有数值型变量的描述性统计。或者，在 su 后面也可以加需要进行描述性统计的变量列表。变量列表可以一次性把要做描述性统计的所有变量都列举出来，或者可以使用通配符。正如多数系统的命令一样，符号「*」可以作为通配符，比如「qg*」代表所有以 qg 开头的变量；此外，Stata 还支持「-」通配符，即按照变量的排列顺序，从某个变量到另外一个变量的所有变量，比如「qg12-p_income」即代表从 qg6 到 p_income 的所有变量。在文件 cfps_adult.dta 中，以下三条命令是等价的：

```
use "cfps_adult.dta", clear
su qg12 qg1203 p_income
su qg12* p_income
su qg12-p_income
```

在论文中我们经常需要汇报描述性统计或者其他统计分析的结果，如果手动一个个输入到 Word 或者 LaTeX 中，是一件非常痛苦的事情。在 Stata 中，我们可以非常方便的使用「outreg2」命令导出我们的统计结果。注意该命令不是 Stata 自带的命令，因而需要使用「ssc install outreg2」进行安装。其语法为：

```
outreg2 using myfile, [{sum(log)|sum(detail)} replace
        eqkeep() eqdrop() keep() drop()]
```

其中：

VARIABLES	(1) N	(2) mean	(3) sd	(4) min	(5) max
qg12	37,147	8,415	18,816	-8	800,000
qg1203	37,147	1,649	18,023	-8	3.000e+06
p_income	37,086	8,934	18,819	-9	442,000

表 1.1: outreg2 示例

- myfile 为要导出的文件名，该命令会通过后缀名（.doc 或者.tex）判断需要导出的格式；
- sum(log) 或者 sum(detail) 为指定要导出的描述性统计是否要报告更详细的统计量。为了做描述性统计，两者必须选择一个；
- replace 表示，如果 myfile 存在，那么覆盖这个文件；
- eqkeep() 和 eqdrop()，表示要保留或者舍弃的描述性统计量；
- keep() 和 drop()，表示要保留或者舍弃的变量。

注意在运行「outreg2, sum()」命令之前，并不需要先做运行 su 命令，可以直接使用该命令导出描述性统计的结果。比如如下命令：

```
use "cfps_adult.dta", clear
outreg2 using summary.tex, replace sum(log)
        keep(qg12-p_income)
```

会得到如表 (1.1) 的结果——一个标准的描述性统计表。

注意到，在表1.1中，三个变量均出现了负数，然而根据三个变量的标签，这三个变量分别代表三种收入，但是收入最小值应该为 0，不可能出现负数。这就是我们做描述性统计的一个重要原因，即发现数据中的异常，并及时处理这些异常。通过查看数据的标签以及调查问卷，我们发现这些负数可能代表了缺失、没有调查等不正常的情况，因而在实际分析中，我们应该对这些情况做出恰当处理，比如最简单的，drop if p_income<0。

此外，为了编程方便，在 Stata 命令运行结束之后，经常会保存一些结果在内存中。这些结果通常分为 r-class 和 e-class[5]。其中 r-class 是一般的返回结果，而 e-class 专指估计（estimation）的结果。这些结果可能是一些标量，也有可能是一些矩阵等等。

比如，对于 su 命令，可以使用「help su」命令查看 su 命令结束后保存在内存中结果的列表。根据 help 命令的结果，在 su 命令结束后，r(N) 就保存了观测的个数，r(mean) 保存了均值，而 r(sd) 保存了标准差等等。如果在运行 su

[5]还有其他类型的返回结果，如 s-class、n-class 等，详见 help return。

命令时指定了不止一个变量，那么这些 r-class 结果只会保存最后一个变量的结果。

我们可以方便的在接下来的命令中使用这些返回变量。比如下面的代码就产生了对 p_income 的标准化后的变量：

```
use "cfps_adult.dta", clear
drop if p_income<0
quietly: su p_income
gen std_p_income=(p_income-r(mean))/r(sd)
su p_income std_p_income
```

由于我们只是希望计算均值和标准差，因而使用了 quietly 禁止 su 命令的输出；产生了标准化的收入之后，我们再次做描述性统计，发现新生成的变量已经被标准化为均值为 0、方差为 1 的变量了。

除了以上介绍的「summarize」命令之外，还有其他的描述性统计命令可以使用，如相关系数矩阵「correlate」命令、Spearman 秩相关命令「spearman」等等。可以自行查阅相关帮助学习。

1.3.2 离散变量

以上「summarize」命令给出的描述性统计，如均值、方差等，只对连续型随机变量或者 0-1 型随机变量有意义，然而有一些变量属于分类变量或者顺序变量，它们的值代表了不同的水平（level），其绝对数值的加减等算数运算没有意义，因而均值、方差等描述性统计量也就没有意义了。

比如，「最高学历」这个变量主要有文盲、小学、初中、高中等不同水平，在数据中分别用 1\2\3\4 来代替，但是这些数字只是代表了不同水平，其算数运算没有意义。

处理这类数据我们经常会画频率或者频数表。在 Stata 中，我们可以使用「tabulate」命令制作频率表。比如，如果我们希望制作最高学历这一变量的频率表，那么只要使用命令「ta te4」即可。注意在结果表格里面，其分类是使用的数据的 label 而不是具体的值，如果需要使用具体的值放在表格里面而不用 label，可以在命令后面加「nolabel」选项。

同样的，在结果表格中，我们发现有负值存在，比如-8 代表「不适用」，即没有调查这个问题，「-1」代表不知道。在接下来更详细的统计分析中一定要注意对这些观测做恰当处理。

当我们有两个分类变量需要同时分析时，通常会使用所谓的列联表。比如，我们希望同时研究最高学历和性别这两个变量，研究最高学历是否与性别有关系，在 Stata 中可以直接使用 tab 命令完成，比如：

```
use "cfps_adult.dta", clear
drop if te4<0
tab te4 cfps_gender, ch
```

最高学历	性别 女	男	Total
文盲/半文盲	198	134	332
小学	262	388	650
初中	577	708	1,285
高中/中专/技校/职高	250	296	546
大专	126	107	233
大学本科	82	106	188
硕士	4	5	9
Total	1,499	1,744	3,243

Pearson chi2(6) = 40.4383 Pr = 0.000

表 1.2: 列联表示例

以上程序首先删除了那些学历不正常的数据，然后使用 tab 命令制作了列链表，如表 (1.2) 所示。此外，为了对这两个变量的独立性做假设检验，我们还添加了 ch 选项，汇报了 Pearson's χ^2 统计量。

表中我们可以看到两个分类变量所组成的各个分类里面的频数，以及边际频数。最后，Pearson's χ^2 检验 $p-value$ 接近于 0，因而可以拒绝原假设，认为最高学历和性别不是独立的。

此外，分类变量还与 Stata 中的「因子变量」有着密不可分的联系。对于一个分类变量，我们可以用 tab 命令产生其虚拟变量，比如对于最高学历 te4 这个变量，可以使用：

```
use "cfps_adult.dta", clear
drop if te4<0
tab te4, gen(edu)
```

如此就产生了 7 个虚拟变量（dummy variables）edu1-edu7，分别对应着 te4 的 7 种可能的取值：文盲/半文盲、小学、初中等等。比如，变量 edu1 的 label 为「te4== 文盲/半文盲」，即这个变量对于文盲/半文盲来说 =1，否则 =0；变量 edu2 的 label 为「te4== 小学」，即这个变量对于小学文化的观测来说 =1，否则 =0，后面以此类推。

如果我们使用 su edu* 命令，就可以得到这 7 个虚拟变量的描述性统计，每个变量的均值实际上就是对应种类占全部样本的比例。比如，edu1 的均值为 0.1029，意味着样本中有 10.29% 的人是文盲/半文盲。注意由于所有的 edu* 都是虚拟变量，即 0-1 变量，因而其标准误 r(sd) 总是等于 $\sqrt{r(mean)\cdot[1-r(mean)]}$。

以上我们使用 tab 命令产生了一个分类变量的虚拟变量，而在 Stata 中，可以不用具体产生这些虚拟变量，而是直接用所谓「因子变量」即可进行分析。比如以上的描述性统计我们可以直接用：

```
use "cfps_adult.dta", clear
drop if te4<0
su i.te4
```

命令即可得到虚拟变量的描述性统计，使用 i.te4 与先生成 te4 的虚拟变量，再使用虚拟变量是等价的，不过 i.te4 的好处是避免了产生新的虚拟变量，即麻烦又占内存。这里需要注意两点，首先是，作为因子变量的取值不能为负，因而我们必须首先把 te4<0 的观测删掉，不然会报错；其次是，仔细观察使用因子变量所产生的描述性统计，只汇报了 6 个分类的描述性统计，文盲/半文盲类被忽略掉了，这是由于因子变量经常用在回归分析中，而在回归分析中由于识别条件，因而不能把所有的虚拟变量同时加进去。当使用 i.te4 这类语法时，Stata 会把值最小的分类（文盲/半文盲）省略，作为比较的基准。如果想要改变这个基准的类别，可以使用 ib#.te4，其中 # 为想要使用第几类作为基准，比如如果想要用小学作为基准，那么可以使用 ib2.te4。如果想要全部显示，在 su 的选项中添加 allbaselevel 选项即可。

此外，还可以使用 # 符号将两个或者多个分类变量联合起来产生新的分类变量，即两个分类变量的乘积。比如「i.te4#i.cfps_gender」即产生了两种性别和七种学历共 14 种组合。如果我们使用：

```
use "cfps_adult.dta", clear
drop if te4<0
su i.te4#i.cfps_gender
```

就得到了 13 组均值和标准差。

除了 i. 算子之外，还有 c. 算子。i. 算子代表后面的变量是分类变量，而 c. 算子表明后面是连续型变量。同样可以使用 # 符号产生新的交互项，比如 i.te4#c.p_income 就产生了 7 个新的变量，即 7 个虚拟变量分别与 p_income 的乘积，比如新产生的第一个变量为：

$$i.te4\#c.p\_income\,(1)=\begin{cases} p\_income & \text{文盲/半文盲} \\ 0 & \text{其他} \end{cases}$$

其他 7 个变量以此类推。使用：

```
use "cfps_adult.dta", clear
drop if te4<0
su i.te4#c.p_income
```

可以查看这七个变量的描述性统计。注意这七个变量都是混合型随机变量，即有离散的部分（te4 是否属于某一组），也有连续的部分（如果属于这一组 p_income 的值）。

最后，分类变量还经常在「by」前缀中使用。与「quietly」一样，「by」命令是一个前缀，用于根据某个分类变量分别执行后面的程序，其语法为：

```
by varlist [, sort rc0]: stata_cmd
bysort varlist [, rc0]: stata_cmd
```

其中 varlist 是一个或者多个分类变量，冒号后面代表 Stata 命令。由于在使用 by 前缀时需要先进行排序（比如使用 sort 命令先对 varlist 排序），为了方便起见，可以在 by 命令后面加入「sort」选项，或者直接使用 bysort。选项「rc0」代表即使在某一类中命令出错，也不停止程序而是继续执行。比如，如下命令就根据教育程度分别对收入进行了描述性统计：

```
use "cfps_adult.dta", clear
bysort te4: su p_income
```

1.3.3 作图

Stata 有非常强大的绘图功能。在「Graphics」菜单中可以找到大多数需要的统计图的类型。最常用的图形比如：

- 柱状图（bar）：使用「graph bar」命令，可以对不同变量的描述性统计画柱状图，也可以根据一个分类变量画频率图。
- 直方图（histogram）：使用「histogram」命令，适用于连续的随机变量，是对密度函数的样本近似。
- 线图（line）：表示一个变量与另外一个变量之间的关系，比如可以表示 GDP 随着时间的发展趋势，或者消费随着收入的变化等等。
- 散点图（scatter）：同样表示两个变量之间的关系，不同的是将数据点按照两个变量的坐标直接画在图上。
- 箱线图（box plot）：将数据的中位数、上下四分位数、最大值、最小值用箱线表示的图。
- .....

对于这些命令我们将不再具体展开，我们下面将以实际代码展示如何画图。

以上的这些图都可以再次进行组合，即多种图形组合在一张图上。我们可以使用「twoway」命令很方便的将不同的图组织在一张图中，下面我们将通过示例展示 twoway 命令的用法。

此外，还有一些辅助指令，比如可以使用「graph save」命令将生成的图片保存成.gph 格式；使用 graph combine 命令将已经保存的图排列在一张图中；使用 graph export 命令导出.png 等格式的图片文件。

以下是一个画图的综合示例：

```
use "cfps_adult.dta", clear
drop if te4<0
drop if p_income<=0

gen lincome=log(p_income)
quietly: su lincome
gen std_lincome=(lincome-r(mean))/r(sd)
gen norm_dens=normalden(std_lincome)
label variable norm_dens "Std. Normal density"
sort std_lincome
twoway (hist std_lincome)
       (line norm_dens std_lincome)
       , saving(g1, replace)

sort std_lincome
gen empirical_cdf=_n/_N
gen norm_cdf=normal(std_lincome)
label variable norm_dens "Std. Normal CDF"
twoway (line empirical_cdf std_lincome)
       (line norm_cdf std_lincome)
       , saving(g2, replace)

gen age=2014-cfps_birthy
gen age2=age^2
quietly: reg lincome age age2
predict predict_lincome
sort age
twoway (scatter lincome age)
       (line predict_lincome age)
       , saving(g3, replace)

graph box p_income, over(te4) saving(g4, replace)

graph combine g1.gph g2.gph g3.gph g4.gph, col(2)
graph export example_graph.png, replace
```

图 (1.2) 给出了以上程序的运行结果。

在以上程序中，我们首先剔除了收入和教育水平存在问题的观测。在第一幅图中，我们首先对个人收入取 log，再将对数收入做标准化处理，得到 std_lincome。

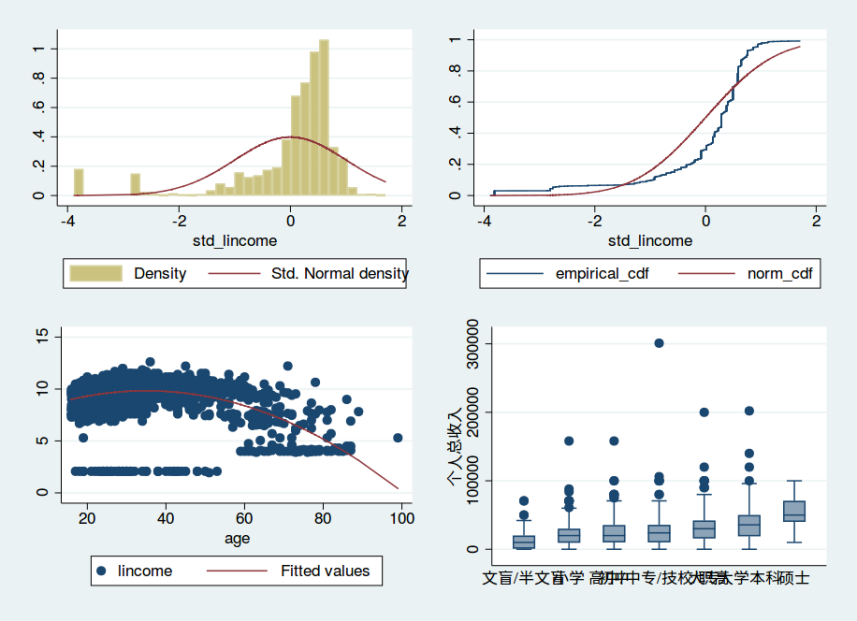

图 1.2: 画图示例

接下来，我们使用 normalden() 函数对每个 std_lincome 产生其对应的正态分布密度函数值，并对 std_lincome 进行排序，最后画出 std_lincome 的直方图以及标准正态分布的密度函数，并保存为 g1.gph。

在第二幅图中，我们先对 std_lincome 排序，这样 _n/_N 就是其经验分布函数，同时我们用 normal() 函数计算了标准正态分布的分布函数，并将其画在同一张图中进行比较，最后将图保存为 g2.gph。

在第三张图中，我们先生成了年龄和年龄的平方两个变量，并用线性回归模型，用年龄和年龄的平方解释对数收入，使用 predict 命令得到预测值。接着由于我们的横轴是 age，因而需要对 age 进行排序，将散点图和预测值画在一张图中，保存为 g3.gph。

最后，我们根据最高学历进行分类，分别画出了每种学历的对数收入的箱线图，保存为 g4.gph。

得到了以上四张图以后，我们用 graph combine 命令将四张图组合在一起，并指定四张图排成两列。最终，使用 graph export 命令把最终的图保存成.png 格式。

在 Stata 中有种类繁复的画图命令，每种命令又有大量选项，比如有控制线条、散点、坐标轴、标签、背景的样式、颜色等等等等的各种选项，具体命令和选项可以借助 help 文档进行学习和使用。

1.4 do-文件与 log-文件

以上我们初步学习了 Stata 命令的一些初步知识，通过在 Stata 窗口的命令输入框中输入命令可以完成 Stata 大部分的操作。然而以上单条命令运行的方式并不方便。当我们有大批量的任务需要完成时，如果一条条输入命令，一次命令输错了，由于 Stata 没有撤销的功能，我们就必须从头再重新来过一次。所以多数情况下，我们使用 Stata 命令都是通过写 do 文件的方式来完成的。

一个 do 文件就是一个以.do 为后缀名的文本文件。我们把需要进行的操作全部书写到一个文本文件里面，并保存成.do 的后缀名，然后直接在 Stata 命令框中输入「do do_filename」就可以批量运行 do 文件里面的所有程序了。或者，在 Linux 命令行下，使用「stata do_filename」（或者是 stata-se 等）可以直接在终端中运行 do-file。

Stata 自带了一个 do 文件的编辑器，可以通过工具栏上的 do 文件编辑器按钮打开。在这个编辑器里面可以直接写 Stata 命令，点击右上方的 Execute(do) 按钮可以一键运行该文件。或者，也可以选中文件的一部分并点击 Execute(do) 按钮，Stata 将会只运行选中的部分。

我们之前介绍了可以使用「set more off」将「-more-」关闭，一般我们在写.do 文件时总是会先使用该命令，防止程序因为-more-的出现而终止。

就像所有语言一样，do 文件也有注释。Stata 的注释同样分为单行注释和多行注释。单行注释有两种，一种以两个斜杠开头，即以「//」开头，可以出现在任何位置，只要出现了两个斜杠，就代表从这里到这一行的末尾都是注释。或者，如果这一行都是注释，可以直接以星号「*」开头。注意「//」可以允许注释从一行的后半段开始，而「*」开头的注释必须整行都是注释。多行注释以「/*」开始，以「*/」结束。被注释的部分在 do 文件编辑器中都会显示绿色。

注释有一个巧妙的应用是在 do 文件中换行。在 Stata 中，默认 do 文件每一行是一条命令。但是有的时候命令太长，全都写在一行不便于阅读，一个可行的方案是在命令的中间加三个斜杠「///」后面跟一个回车，然后在新的一行中继续写命令，这样 Stata 就不会认为是新增加了一行。或者，也可以将回车包在多行注释里面，即在「/*」和「*/」中间加入回车，Stata 也会忽略这个回车。在本讲义中，为了方便展示，通常会对比较长的命令进行换行展示。

由于经常会使用 Stata 处理变量非常多、非常复杂的调查数据等等，因而养成包括为变量取容易理解的名字、为每一块程序写注释、为关键的行写注释等好的习惯是非常重要的，特别是处理大型任务时，良好的习惯可以大大提高效率。

有的时候我们在 do 文件中不希望某一块代码的输出打印到结果对话框中。前面我们介绍了使用 quietly: 前缀关闭某一行命令的输出，而在写 do 文件的时候，如果一大块代码都不希望有任何输出，每行都写一个 quietly: 前缀是非常麻烦的。遇到这种情况，可以使用 quietly{} 语句，将所有的不希望输出的语句都写在大括号里面。如果在这个大括号里面有某行语句我们希望其输出结果，就单独在这一行里面加入 noisily: 前缀。以下的程序就关掉了所有的输出，只保留了一个 su 的输出：

```
set more off
use "cfps_adult.dta", clear
quietly{
    su p_income
    gen demean_income=p_income-r(mean)
    noisily: su demean_income
}
```

do 文件不仅仅提供了批量处理的方便途径，也是我们接下来要学习的 Stata 编程的基础。

在通常情况下，Stata 的命令输出结果仅仅在 Reuslts 窗口显示。如果我们希望长久的保留下命令运行的过程（而不是仅仅保留统计结果，比如使用 outreg2），那么可以使用 Stata 的日志文件（log 文件）。

在 Stata 中，可以使用如下命令开始一个新的 log 文件：

```
log using filename [, append replace {text|smcl}]
```

其中 filename 是 log 文件的文件名，append 表明当 filename 存在时，新的内容将会附加在已经存在的文件后面，而 replace 意味着直接覆盖已经存在的文件。Stata 的日志文件有两种格式，文本文件即 text 格式和 smcl 格式。文本文件可以使用任何文本编辑器打开，而 smcl 格式只能用 Stata 打开。如果 log 文件需要传阅给其他人而又不能确保其他人安装了 Stata，可以使用文本文件，否则尽量使用 smcl 格式，因为保留了字体、颜色等格式。

使用以上命令之后，一直到「log close」命令，中间的命令过程都将被记录下来。如果希望暂时停止 log 文件的记录，可以使用「log on/off」命令临时打开或者关闭 log 文件的记录。更多关于 log 文件的命令和说明可以查看帮助文档，help log。

1.5 数据处理

1.5.1 生成数据

之前我们介绍了如何使用「gen」命令产生一个新的变量，然而使用「gen」命令只能对同一个观测的不同变量之间进行运算，而无法分组进行运算得到新的变量。比如，几个不同的个体都存在于同一个家庭中，使用「gen」命令只能对每一个人的不同变量进行操作，而不能对同一个家庭内部不同个体的数据进行加总等操作，比如不能计算家庭总收入、总支出等。此时，一个想法是可以将「gen」命令和「by」前缀一起使用。

实际上这个想法是可行的。比如，如果使用「by」前缀分组，「_n、_N」将分别代表组内的排序和个数，因而我们可以将「gen」命令和「_n、_N」结合

在一起使用产生一个组的内部编号以及组内成员个数。比如如下命令就产生了家庭内部的一个序号和家庭人数：

```
use "cfps_adult.dta"
bysort fid14: gen family_id=_n
bysort fid14: gen family_pop=_N
```

此外，「by」命令还支持如下语法：

```
by varlist1 (varlist2) : stata_cmd
```

与之前的「by」命令相比，多了一个括号，括号中是一个额外的变量列表。这种形式的「by」命令将会检查数据是否已根据 varlist1 和 varlist2 共同排序，如果已经排好序，只按照 varlist1 进行分类运行 stata_cmd 命令。

使用这个特性，我们可以使用如下命令产生一个新的变量「f_max_income」，其值为家庭中个人收入最多的那个人的收入：

```
bysort fid14 (p_income ): gen f_max_income=p_income[_N]
```

尽管结合使用不同命令可以完成一些比较复杂的生成新变量的任务，但是在实际处理时仍然有很多限制。为此 Stata 还提供了「扩展的 gen 命令」，即「egen」命令，其基本语法为：

```
egen newvar = fcn(arguments) [if] [in] [, options]
```

其中 newvar 为新的变量名，fcn 为可以在 egen 命令中使用的函数的名字，arguments 为 fcn 的参数，不同的 fcn 有不同的参数类型。

egen 命令支持非常多的 fcn 函数，主要分为以下两类：

- 生成分组代码，比如：

```
egen provid=group(provcd14)
```

即根据 provcd14 这个变量生成了一个新的代码 provid，只要 provcd14 的值相同，代码也相同，否则不同。通常用于字符串变量转化为代码。

- 按行分组类：通常需要在 options 中加入 by(varlist) 选项，在运行时会首先根据 varlist 分组，再进行计算，常用的函数如：均值（mean）、中位数（median）、百分位数（pctile）、最大值（max）、最小值（min）、众数（mode）、个数（count）、求和（total）、标准差（sd）、偏度（skew）、峰度（kurt）等。比如，以下程序产生了家庭中的成人人数、家庭中个人收入的最大值、最小值以及家庭总收入：

```
egen f_num_of_adult=count(pid), by(fid14)
egen f_max_income=max(p_income), by(fid14)
egen f_min_income=min(p_income), by(fid14)
```

```
egen f_sum_income=total(p_income), by(fid14)
```

- 按列计算类：这类 fcn 一般 arguments 为变量列表，选项中也要求不能有 by(varlist) 选项，其功能是完成一些按列运算，这些运算通常是 gen 命令需要比较繁琐的操作才能完成的，比如：varlist 中元素的个数（anycount）、varlist 中是否存在某个元素（anymatch）、将连续变量分组（cut）、varlist 值是否相等（diff）、行均值（rowmean）、行最大值（rowmax）、行最小值（rowmin）、行中位数（rowmedian）、行加总（rowtotal）、行方差（rowsd）等等。比如以下程序通过不同的变量生成「是否为党员」这一指标，并比较了不同变量生成的指标是否相同：

```
egen pmember=anymatch(qn401_s_*), values(1)
gen pmember2=(qn402>0 & qn402~=.)
gen pmember3=pn401a==1
egen pconsistent=diff(pmember pmember2 pmember3)
sum pconsistent
```

以上程序中，首先使用「您是哪些组织成员」四个变量判断个人是否为党员，如果四个变量中有一个为 1，产生的 pmember 就等于 1；接下来使用「入党时间」这个变量判断是否为党员；最后使用数据中本来就有的「是否中共党员」变量判断是否为党员；最后，使用 egen 的 diff 函数判断三个是否党员变量是否相等（=1 代表三个变量不一致），使用描述性统计发现大约有 6% 的个体存在三个标准不一致的情况。

此外，egen 命令还支持很多其他函数，可以使用「help egen」查看 egen 命令支持的函数列表。

1.5.2 合并数据

实际中经常遇到的另外一个问题是合并数据。合并数据有两种：纵向的添加更多的观测（append）以及横向的添加更多的变量（merge），在 Stata 中，可以非常方便的使用「append」、「merge」等命令合并数据。

命令「append」的作用是将一个数据文件「附加」在主文件的后面。如图 (1.3) 所示，如果我们打开了 file1.dta，我们希望将 file2.dta 附加在 file1.dta 的后面，那么可以使用如下代码实现：

```
use file1.dta, clear
append using file2.dta
save file3
```

以上代码首先打开 file1.dta 作为主文件，并使用 append 命令将 file2.dta 附加在主文件后面，最后将合并的文件存为 file3.dta。

_n	pid	fid	year	age	pincome
1	11	1	2010	29	150000
2	12	1	2010	28	100000
3	21	2	2010	35	200000
4	22	2	2010	33	80000

file1.dta

_n	pid	fid	year	age	pincome
1	11	1	2012	31	156000
2	12	1	2012	30	110000
3	21	2	2012	37	250000
4	22	2	2012	35	85000

file2.dta

⇓append

_n	pid	fid	year	age	pincome
1	11	1	2010	29	150000
2	12	1	2010	28	100000
3	21	2	2010	35	200000
4	22	2	2010	33	80000
5	11	1	2012	31	156000
6	12	1	2012	30	110000
7	21	2	2012	37	250000
8	22	2	2012	35	85000

file3.dta

图 1.3: append 命令示意

「append」命令还有几个选项，比如 gen() 选项可以用来生成一个标示数据来源文件的变量、nolabel 用来指示程序不要拷贝标签，force 选项用来强制 append 命令将不同类型的数据（比如数字和字符串）合并起来等等。详细使用方法可以查看「help append」。

另一个非常常用的合并数据的需求是横向的合并，即根据一些 id 将不同数据来源的数据合并在一起，比如常见的将保存在不同文件的个人收入数据与健康数据合并、将个人数据与对应的家庭数据合并、将企业数据与对应的地区级别的数据合并等等。这些合并有这么几个特点：首先，必须有一个或者几个变量标示两个数据集之间观测的对应关系，比如个人的 id 可以用来将两个数据集的个人一一对应起来，家庭 id 可以将个人对应到每一个家庭，而省份代码可以将企业对应到地区；其次，不同的合并有不同的对应关系，比如个人的收入和健康数据之间应该是一一对应的，而个人数据与家庭应该是多对一的关系。

根据以上特征，Stata 提供了多个不同的「merge」命令：

```
merge 1:1 varlist using filename [, options]
merge m:1 varlist using filename [, options]
merge 1:m varlist using filename [, options]
merge m:m varlist using filename [, options]
merge 1:1 _n using filename [, options]
```

其中 merge 命令后面的 1:1、m:1、1:m、m:m 等用来标示主文件（目前打开的文件）和 using 文件之间的对应关系是一对一的、多对一的、一对多的还是多对多的，冒号前面的代表主文件（master），冒号后面的代表 using 文件。varlist 代表指定的用于识别观测是否对应的变量列表，比如个人的 id、家庭的 id 或者

_n	pid	fid	year	age	pincome
1	11	1	2010	29	150000
2	12	1	2010	28	100000
3	21	2	2010	35	200000
4	22	2	2010	33	80000
5	31	3	2010	54	20000

file1.dta

_n	fid	year	hincome
1	1	2010	256000
2	2	2010	300000
3	1	2012	300000
4	2	2012	350000

file2.dta

⇓merge

_n	pid	fid	year	age	pincome	hincome	_merge
1	11	1	2010	29	150000	256000	3
2	12	1	2010	28	100000	256000	3
3	21	2	2010	35	200000	300000	3
4	22	2	2010	33	80000	300000	3
5	31	3	2010	54	20000	.	1
6	.	1	2012	.	.	300000	2
7	.	2	2012	.	.	350000	2

file3.dta

图 1.4: merge 命令示意

将家庭 id 与时间变量一起识别。

比如在图 (1.4) 中，我们希望将个人的数据（file1.dta）与家庭数据（file2.dta）合并在一起，其中 pid 代表个人的 id，而 fid 代表家庭 id。此外，数据中还有 year 变量用于标示年份。如果我们首先打开 file1，那么主文件是 file1，using 文件是 file2，应该是一个多对一的关系，因而应该用 m:1。使用如下命令可以完成以上任务：

```
use file1.dta, clear
merge m:1 fid year using file2.dta
keep if _mege==3
drop _merge
save file3
```

在以上程序中，我们使用 fid 和 year 识别出每个人、每个年份属于哪个家庭和年份。注意到，有的数据是主文件中有但是 using 文件没有与之对应的，比如 file1 的第五行；有的数据是主文件没有但是 using 文件有的，比如 file2 的 3、4 行；有的是完全可以匹配起来的，在 merge 命令运行结束以后，会产生一个新的变量「_merge」，标示了数据来源：1 代表仅来自主文件；2 代表仅来自 using 文件；3 代表匹配。因而在接下来的程序中，我们只保留了匹配的样本，最后删掉了 _merge 这个变量（如果不删除，下一次使用 merge 命令会报错）。

此外，在 1:m 中的主文件、m:1 中的 using 文件、1:1 中的两个文件都要求 varlist唯一识别了观测。比如，在以上的例子中，要求 file2 不能出现 fid 和 year 同时相同的两个或者多个观测，否则会报错。当这种错误发生时，可以使用「duplicates」命令排查，比如，「duplicates list varlist」可以列出所有的 varlist

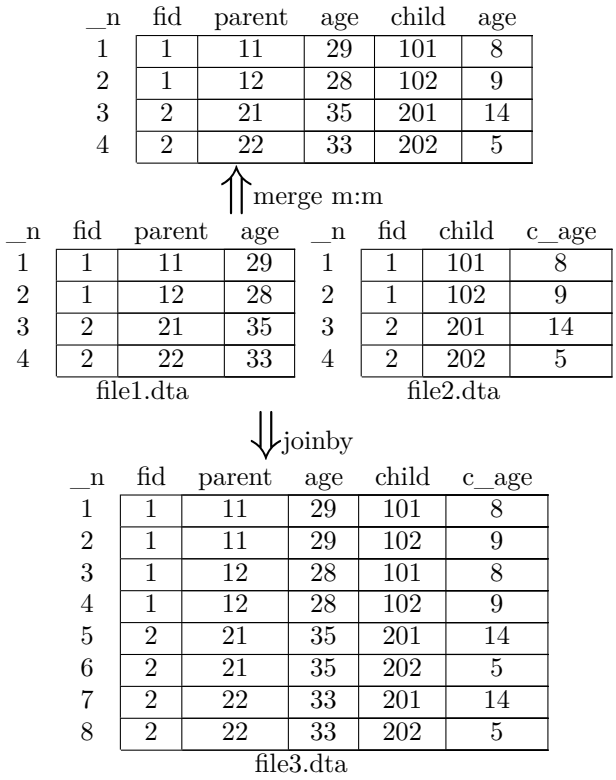

图 1.5: joinby 命令示意

重复的观测，而「duplicates drop」可以删除重复观测。

原则上，使用 varlist 至少要在一个文件中唯一识别观测。尽管 Stata 提供了 m:m 的 merge 命令，但是其实际运行并不会产生笛卡尔乘积，而是将主文件的第一个观测与 using 文件的第一个匹配上的观测进行匹配，第二个相同的观测与第二个匹配上的进行匹配。比如对于图 (1.5) 中的 file1 和 file2，使用 merge m:m 仅仅是按照顺序将两个文件匹配起来，这显然在绝大多数情况下不是我们希望得到的结果。所以在实践中，应该尽量避免使用 merge m:m，因为几乎没有任何情况需要我们使用 merge m:m。

如果对于图 (1.5) 中的数据，我们希望产生每个家庭内部家长和孩子的笛卡尔乘积，可以使用「joinby」命令，其语法如下：

```
joinby [varlist] using filename [, options]
```

比如对于以上任务，可以简单的使用：

```
use file1.dta, clear
joinby fid using file2.dta
save file3
```

即可得到家长和孩子的笛卡尔乘积，如图 (1.5) 所示。

对于 Stata 16 及以上版本，我们还可以选择使用数据库完成 merge 操作，即将两个文件分别在两个 frame 中打开，然后使用「frlink」命令将两个数据框逻辑上联系在一起，之后使用「frget」命令从匹配好的数据框中匹配数据。值得注意的是，「frlink」命令只允许 m:1 以及 1:1 两种关系，即当前使用的数据框必须更加「微观」。比如，我们可以使用如下程序进行匹配：

```
use datasets/cfps_adult
frames create family
frame family: use datasets/cfps_family_econ
frlink m:1 fid14, frame(family)
```

以上代码首先在 default 数据框中打开了「cfps_adult」数据文件，再创建了「family」数据框，并在其中打开「cfps_family_econ」文件，最后使用 frlink 命令，以「fid14」这个变量作为识别变量，将 family 这个数据框逻辑上匹配到当前数据中。匹配成功后，如果需要使用「cfps_family_econ」文件中的变量，可以使用「frget」命令，比如：

```
frget fincome1, from(family)
frget ave_p_income=fincome1_per, from(family)
gen delta_p_income=p_income-ave_p_income
```

就从「family」数据框中的变量「fincome1」调入到当前数据集中，也可以通过赋值的方式，将「family」数据框中的变量「fincome1_per」调入到当前数据集中并重命名为「ave_p_income」，之后就可以直接对这些变量进行操作了。

1.5.3 加总与转置

此外，Stata 还提供了其他的一些方便的数据操作，我们将介绍三个操作：数据加总、数据转置以及重复观测的处理。

现实中我们经常会碰到数据加总（aggregate）的问题，比如将个人收入加总为地区平均收入、中位数收入，以及将企业增加值加总为行业总增加值等等。之前已经介绍过使用「egen」命令可以完成这些计算，然而使用 egen 命令以后，数据仍然是个人（企业）级别的而非地区（行业）级别的，这个时候就需要使用「collapse」命令，其基本语法为：

```
collapse clist [if] [in] [weight] [, options]
```

其中 clist 可以为以下形式：

```
[(stat)] varlist [ [(stat)] ... ]
[(stat)] target_var=varname ...
```

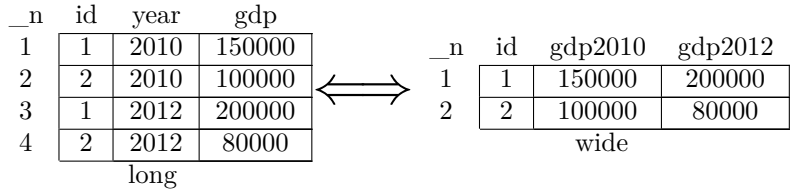

图 1.6: reshape 命令示意

其中 stat 为需要计算的统计量（比如均值、求和、分位数、个数等），varlist 是需要进行加总的变量列表。注意 stat 是可以省略的，如果省略，则默认为均值 (mean)。此外，有时我们会为同一个变量计算多个加总变量，比如对于收入，我们可能同时计算其均值和中位数，为了避免变量名冲突，我们必须要为新产生的变量命名，即以上语法中的 target_name。如果一个变量仅计算一个统计量，那么 target_name 可以忽略，默认为原始变量名。

比如，我们可以使用以下程序计算不同省份的平均收入、平均消费以及中位数收入：

```
use "cfps_family_econ.dta"
collapse (mean) expense m_income=fincome1 /*
      */(median) med_income=fincome1, by(provcd14)
```

以上程序使用 collapse 命令，by(provcd14) 选项表明我们要根据省份进行分组，第一个 (mean) 表示我们要计算后面这些变量的均值，而 (median) 表示要计算后面变量的中位数；由于 expense 只出现了一次，因而没有必要为其提供新的变量名，但是 fincome1 出现了两次，所以我们将结果中的平均收入命名为 m_income，中位数收入命名为 med_income。该程序运行结束以后，打开数据浏览器可以发现仅剩下了 29 个观测，分别是每个省份的平均支出、平均收入和中位数收入。

「collapse」命令支持许多函数，如百分位数（p#）、最大值（max）、最小值（min）、均值的标准误（semean）等，这些函数的具体使用可以查看「help collapse」。

在一些数据中，我们还会碰到「长形式」和「宽形式」数据之间的转换。如图 (1.6) 所示，左边为「长形式」，右边为「宽形式」。在这两者之间转换可以使用「reshape」命令。比如，在图 (1.6) 中，如果我们希望从「长形式」变换为「宽形式」，可以使用如下命令：

```
reshape wide gdp, i(id) j(year)
```

其中「reshape wide」表示要将数据转换为「宽形式」，后面是想要转换的变量列表，选项 i() 代表要保留的作为行的 ID 变量，而 j() 代表要变换成列的变量。比如，在这里「i(id)」代表转换后每行一个 id，而「j(year)」表示将变量 gdp 变换为「gdp2010、gdp2012」等等多个变量。

反之，如果我们希望将「宽形式」变换为「长形式」，可以使用如下命令：

```
reshape long gdp, i(id) j(year)
```

该命令与之前的命令仅仅在于将「wide」改为「long」。需要注意的是，如果数据是「宽形式」的，那么 j() 里面的变量（year）是不存在的，因而这里 j() 里面实际上指定了一个新的变量。

最后，同样的，「reshape」命令还有一些其他的高级用法，比如「reshape xij」、「reshape xi」等等，具体可以查看「help reshape」。

1.6 标量、矩阵与宏

在其他编程语言中，「变量」通常是一个指代内存中一块数据的一个简称。在 Stata 中，「变量」指的是数据集中的一列数据，与其他编程语言中的「变量」概念比较类似的是标量（**scalar**）、矩阵（**matrix**）和宏（**macro**）。

1.6.1 标量

标量（scalar）可以用来存储数字和字符串，而不仅仅是数字。在 Stata 中，可以使用 scalar 命令声明一个标量，比如，命令「scalar a=4」就声明了一个标量 a，其值为 4。「display」命令可以用于显示标量的值，比如：「di a」可以将标量 a 中的值打印到屏幕上。使用 scalar list 命令可以得到内存中保存的所有标量，「scalar drop」可以用来删除标量，如果需要删除所有标量，可以使用「scalar drop _all」。

声明标量时可以进行运算，之前介绍的函数都可以用来当做标量的运算，比如：

```
scalar question="What is the answer? "
scalar answer=21*2
di question answer
scalar randa=rnormal()
scalar randb=runiform()
scalar randb=randa+randb
di randb
```

注意在以上程序中，尽管已经声明了一个标量 randb，在接下来的计算中，如果对 randb 进行修改，仍然需要使用 scalar 命令。

此外，之前提到的命令运行结束之后的 r-class 和 e-class 等结果也都可以用于标量的运算，比如如下程序就计算了方差的 t 统计量以及 p 值：

```
clear
set obs 42
gen x=rnormal()
```

```
quietly: su x
scalar mean_x=r(mean)
scalar sd_x=r(sd)
scalar t=sqrt(42)*mean_x/sd_x
scalar p=t(41,t)
di "t=" t ", and p-value=" p
```

在上例中，我们先生成了 42 个正态分布 $x_i \sim N\left(\mu_0, \sigma_0^2\right)$，这里令 $\mu_0 = 0, \sigma_0^1 = 1$，由于：

$$\sqrt{N}\frac{\bar{x} - \mu_0}{s} \sim t\left(N-1\right)$$

在原假设：$H_0 : \mu_0 \geq 0$ 的条件下，以上检验为左侧检验。在上述程序中，计算了检验统计量 t 以及相应的 p 值。

在 Stata 中，使用 scalar 时可以直接以其名称代替，但是这潜在可能会导致变量与标量之间命名的冲突。特别是，在 Stata 中，变量名是可以用缩写的，比如一个变量名为「yeardummy」，如果没有其他以「year」开头的变量，那么直接使用 year，Stata 会默认此处指的是 yeardummy。此时，如果恰好有一个标量名也为 year，那么 Stata 会优先解释为变量而非标量，因而 Stata 会将名称「year」解释为变量「yeardummy」而非标量「year」。比如在运行上面 t 统计量的例子之后，如果接着运行：

```
gen time=1990
di "t=" t ", and p-value=" p
```

那么显示的结果将是：「t=1990, ...」，显然 Stata 认为这里的 t 是指的变量 time 而非标量 t。解决这一问题的方法是在使用标量时，用 scalar() 声明括号里面的是标量而非变量，即：

```
di "t=" scalar(t) ", and p-value=" p
```

如此就得到了正确的标量 t。当数据比较复杂时，这样的命名冲突可能非常普遍，因而在使用标量时用 scalar() 进行声明是非常好的习惯。当然，这个问题也可以用下面要讲的「tempname」来解决。

1.6.2 * 矩阵

上面介绍了标量，与标量相对应的是矩阵（matrix）的概念。在 Stata 中，矩阵可以直接使用 Stata 提供的命令进行一些简单的处理，也可以使用 Mata 语言进行更加复杂的运算。在这里我们先介绍 Stata 的矩阵相关命令。

在 Stata 中创建一个矩阵有多种方式，第一种方式是我们可以通过手动输入的方式新建一个矩阵，使用「matrix input」命令可以通过手动输入方式新建一个矩阵，使用「matrix list」命令可以在屏幕上显示矩阵，比如：

```
mat input A=(1,2\3,4)
mat list A
```

通过以上程序我们新建了一个 2×2 的矩阵 A，并将该矩阵的值打印在结果窗口中。此外，我们还可以使用「matrix define」命令新建一个矩阵（matrix input 和 matrix def 中的 input 和 def 都可以省略），常用的矩阵运算，如加、减、转置、矩阵相乘、Kronecker 乘积（# 运算符）等都可以使用，比如：

```
mat A=(1,2\3,4)
mat B=(1,0\1,1\0,1)
mat C=A*B'+(B*A)'
mat D=B*B'
mat K=A#B
mat list C
mat list D
mat list K
```

其中符号「'」代表转置。注意在 list D 命令执行后，Stata 会告诉我们 D 矩阵是一个对称（symmetric）矩阵。对称矩阵在矩阵运算中有着重要的作用，比如针对对称矩阵的求逆、特征值等都有更加快速而准确的算法，因而对于对称矩阵，应该尽量使用这些专门针对对称矩阵的函数。

同样的，使用「matrix dir」命令可以列出内存中现有的所有矩阵及其维数，「matrix rename」命令可以修改矩阵名，使用「matrix drop」命令可以删除矩阵。

此外，注意到在所有显示的矩阵中，每一行和每一列都有名字，比如「r1, r2, ..., c1, c2, ...」等等。如果我们计算了 Kronecker 乘积，那么其行和列的名字分别变成了 c1:c1, c1:c2, ..., r1:r1, r1:r2, ... 等等。为矩阵的行、列命名主要是为了在估计模型之后能够方便的将估计结果的向量、矩阵与变量相对应，比如在使用了 regress 命令之后，我们得到了一个系数向量 e(b) 和一个方差矩阵 e(V)，其列（行）的名字就是相应变量的名字。

在 Stata 中，我们可以使用「matrix colnames」以及「matrix rownames」为矩阵的行和列命名，比如：

```
mat A=(1,2\3,4\5,6\7,8)
mat colnames A=A1 A2
mat rownames A=obs1 obs2 obs3 obs4
mat list A
```

为矩阵的行、列命名一个好处是方便矩阵的下标操作。比如，在进行命名之后，我们可以方便的使用「A["obs2", "A1"]」代表矩阵 A 的第二行第一列。注意以上使用行和列名的表述将返回一个子矩阵（sub matrix），而非一个标量，而使用「A[2,1]」将会返回标量。比如下面的命令：

```
mat A=(1,2\3,4\5,6\7,8)
mat colnames A=A1 A2
mat rownames A=obs1 obs2 obs3 obs4
mat subA=A["obs2"..."obs4","A2"]
mat subA2=A[2..4,2]
mat list subA
mat list subA2
mat B=A/A[2,1]
mat B1=A/A["obs2","A1"]
mat B2=A/A[rownumb(A,"obs2"),colnumb(A,"A1")]
```

在以上程序中，subA 和 subA2 是一样的，即当取子矩阵时，行列名和数字都可以使用；但是如果我们希望得到一个标量，那么只能使用数字的形式，因而最后一句「mat B1=A/A["obs2","A1"]」会报错。为了解决这一问题，可以使用「rownumb()」、「colnumb()」函数，给出矩阵行、列的名字，得到其行号和列号。

生成矩阵的第二种方法是直接使用矩阵函数生成一些特殊的矩阵，比如函数「I(n)」就生成了 $n \times n$ 的单位阵，而「J(r,c,z)」生成了一个 $r \times c$ 的矩阵，其矩阵元素全为 z。比如，我们经常提到的一个矩阵：

$$P = \frac{1}{n}\iota\iota', M = I - P$$

可以用如下命令生成：

```
scalar n=4
mat P=1/scalar(n)*J(scalar(n),1,1)*J(1,scalar(n),1)
mat M=I(scalar(n))-P
mat x=(1\2\3\4)
mat P2=P*P
mat M2=M*M
mat Px=P*x
mat Mx=M*x
mat prod=Px'*Mx
mat list M
mat list M2
mat list P
mat list P2
mat list Px
mat list Mx
mat list prod
```

以上程序产生了两个 $n \times n$ 的幂等矩阵 M 和 P，并验证了其为幂等矩阵（即 $M^2 = M$，$P^2 = P$），最后验证 Px 的结果为 $\bar{x}\iota$，而 Mx 为 x 向量去平均，最后 Px 与 Mx 是正交的（内积为 0）。

除了以上函数以外，还有很多其他的矩阵函数可以供使用，比如：

- 向量与矩阵转化：矩阵转化为向量（vec(M)）、向量转化为对角阵（diag(v)）、取得矩阵的对角线元为向量（vecdiag()）；
- 矩阵的运算：行列式（det()）、迹（trace()）；
- 矩阵求逆：对称矩阵的求逆（invsym()）、一般矩阵求逆（inv()）；
- 矩阵行、列操作：行数（rowsof()）、列数（colsof()）；
- Cholesky 分解：cholesky()，将一个正定的对称矩阵 S 分解为 $S = RR'$，其中 R 为下三角矩阵。

此外还有很多其他的矩阵函数，可以使用「help matrix functions」查看所支持的所有矩阵函数。

上面提到了矩阵分解，除了 Cholesky 分解之外，Stata 还有其他命令完成奇异值分解（SVD）、特征值分解等。比如，「matrix svd」命令用来计算矩阵的奇异值分解，而「matrix symeigen」完成了对称矩阵的特征值分解，「matrix eigenvalues」计算一个一般矩阵的所有特征值。如果需要计算一个一般的方阵的特征值分解，需要使用 Mata 的「eigensystem()」函数，「help mf_eigensystem」命令提供了具体的使用方法。

如果需要从数据中创建矩阵，可以使用「mkmat」命令。该命令后面跟变量名列表，并使用这些变量创建矩阵。在下面的例子中，展示了其使用方法：

```
set obs 10
gen x=runiform()
gen y=rnormal()
mkmat x
mkmat y
matrix list x
mkmat x y, matrix(M)
matrix list M
matrix M2=(x,y)
matrix list M2
```

在以上程序中，我们首先产生了两个变量 x 和 y，然后使用 mkmat 命令分别创建了两个矩阵 x 和 y，其值等于两个变量的值。接着我们展示了如何使用 mkmat 命令产生了一个 10×2 的矩阵 $[x, y]$。注意在产生的矩阵中，其列的名字都被赋予为变量名。

反过来，我们也可以将矩阵存储为变量，使用「svmat」命令可以完成以上任务。下面的例子中，我们展示了如何使用矩阵命令将两个相关的变量标准化为两个不相关、标准差为 1 的变量：

```
clear
set more off
set obs 100
// 生成数据
gen x=rnormal()
gen y=rnormal()+x
// 协方差矩阵
corr x y, covariance
matrix corrmat=r(C)
// 将x,y形成内存，放在X中
mkmat x y, matrix(X)
// 对称矩阵的特征根分解
// 分解之后G为特征向量组成的矩阵，lam为特征向量
matrix symeigen G lam=corrmat
// 将特征值开方
matrix lam[1,1]=sqrt(lam[1,1])
matrix lam[1,2]=sqrt(lam[1,2])
// 产生协方差矩阵的1/2次方
matrix sqrt_corrmat=G*diag(lam)*G'
// X乘以协方差矩阵的-1/2次方
matrix transform_X=X*invsym(sqrt_corrmat)
// 命名
matrix colnames transform_X=trans_x trans_y
// 将矩阵的列保存为变量，变量名为上面命名的名字
svmat transform_X, names(col)
// 查看现在的协方差矩阵
corr trans_*
```

在上例中，我们首先产生了两个相关的变量 x 和 y，并使用「corr x y, covariance」命令计算了两个变量的协方差矩阵。命令执行后，协方差矩阵存储在 r(C) 中，因而我们将这个协方差矩阵保存下来，记为 $corrmat$，同时将变量 x 和 y 保存为矩阵 X。

接下来，由于 $corrmat$ 是一个实对称矩阵，因而我们可以使用特征值分解将该矩阵分解为：$corrmat = G\text{diag}\,(lam)\,G'$，其中 $lam = (\lambda_1, \lambda_2)$ 为一个包含着特征值的行向量，而 G 为对应的特征向量组成的矩阵。接下来，我们计算了

$$corrmat^{1/2} = G\text{diag}\left(\sqrt{\lambda_1}, \sqrt{\lambda_2}\right) G'$$

再计算

$$transformX = X * corrmat^{-1/2}$$

即得到了标准化之后的 X。最后为 transform_X 的列命名，并使用 svmat 命令将 transform_X 保存为变量。最后，使用 corr 命令检查是否经过变换以后每个变量的标准差为 1、相关系数为 0。

然而，如果数据过大，有时会提示「matsize too small」，遇到这种情况时，可以使用「set matsize」命令重新设置矩阵的最大大小。然而，即使在 MP/SE 版本中，允许的最大大小也不过 11000，现实中数据的观测数很容易就超过 11000 条的限制了。此时，一些方便的计算矩阵交叉乘积的命令可以很大程度上规避以上的限制。此外，Mata 是不受 matsize 限制的，因而对于更加复杂的运算通常需要使用 Mata 语言。

在 Stata 中，计算交叉乘积主要有以下命令：

- 「matrix accum」：计算 $\sum_{i=1}^{N} x_i x_i' = X'X$，其中 x_i 为 $k \times 1$ 的列向量，

$$X = \begin{bmatrix} x_1' \\ x_2' \\ \vdots \\ x_N' \end{bmatrix}$$

- 「matrix glsaccum」：计算 $\sum_{i=1}^{N} X_i' W_i X_i = X'BX$，其中：

$$B = \begin{bmatrix} W_1 & 0 & \cdots & 0 \\ 0 & W_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & W_N \end{bmatrix}$$

- 「matrix opaccum」：计算 $\sum_{i=1}^{N} X_i' e_i e_i' X_i$，其中 e_i 为 $n_i \times 1$ 的列向量；
- 「matrix vecaccum」：计算 $\sum_{i=1}^{N} y_i x_i' = y'X$，其中 $y = (y_1, ..., y_N)'$ 为 $N \times 1$ 的列向量。该命令默认变量列表中的第一个变量为 y，其他变量为 x。

实际上以上程序使用上述介绍的 makat 命令等都可以实现，然而当数据了过大时，mkmat 命令受限于 matsize，但是 accum 等命令一般来说可以避免矩阵过大的问题。以下程序展示了如何使用 mkmat 命令实现 accum 命令的功能，以及如何使用 accum 等命令实现最小二乘估计：

```
clear
set more off
set obs 100
```

```
// 生成数据
gen x1=rnormal()
gen x2=runiform()
gen y=x1+x2+rnormal()
// 变量转化为矩阵
mkmat x1 x2, matrix(X)
mkmat y
matrix Xc=(X,J(rowsof(X),1,1))
// X'*X
matrix XX=X'*X
matrix XcXc=Xc'*Xc
matrix list XX
matrix list XcXc
// 使用accum
matrix accum Xcum=x1 x2, noconstant
matrix accum Xccum=x1 x2
matrix list Xcum
matrix list Xccum
// 计算最小二乘估计
matrix vecaccum Xycum=y x1 x2
matrix list Xycum
matrix b=invsym(Xccum)*Xycum'
matrix list b
```

注意当我们使用 accum、vecaccum 等命令时，默认在变量列表后面加入常数项，如果不希望加入常数项，需要使用「noconstant」选项声明。通过以上程序，我们展示了如何使用 mkmat 命令实现 accum 命令的功能，然后通过计算：

$$b = \left[(X'X)^{-1}\right][X'y] = \left[\sum_{i=1}^{N} x_i x_i'\right]^{-1}\left[\sum_{i=1}^{N} x_i y_i\right]$$

得到了最小二乘估计。值得注意的是，以上命令结束以后，b 的行名称已经非常聪明的默认使用变量名称指定了。

1.6.3 宏

在 Stata 中，宏（macro）是比标量和矩阵更为常用的工具。一个宏实际上就是用一个字符串代替另外一个字符串。在 Stata 中区分两种宏：

- 局部宏：即所谓的 local，其作用域被限制在一个程序（program）内部或者一个 do 文件内部，或者 Stata 控制台内部。使用「local」命令定义，引

用时宏名称需要以左引号（「`」，即键盘上数字 1 左边的按钮）和右引号（「'」，即键盘上回车左边的按钮）包围起来。例如：

```
local answer "42"
di `answer'
```

即创建了一个叫做「anwser」的局部宏，接下来我们使用「`answer'」引用这个宏，这个宏在运行之前被字符串「42」所替代，因而实际运行时运行的命令为「di 42」。

- 全局宏：即所谓的 global，其作用域为全局，一处声明，到处可用。使用「global」命令定义，引用时需要在宏名称前面加上「$」符号。比如：

```
global answer "42"
di $answer
```

与局部宏不同的是，比如在某个 do 文件中声明了全局宏，执行该 do 文件以后，在命令窗口中仍然可以引用该全局宏；而如果在 do 文件中声明的是局部宏，那么 do 文件结束后，在命令窗口中局部宏不可被引用，反之亦然。此外，在任何地方定义的全局宏，在 program 中都可以使用，但是对于局部宏，program 中只能使用 program 内部定义的局部宏。

需要注意的是，宏仅仅是做了字符串替换的工作，因而以下命令会报错：

```
local question "What is the anwser? "
local answer "42."
di `question' `answer'
```

这是由于在实际执行的时候，第三行命令首先被翻译成了：「di What is the anwser? 42.」，然而 Stata 无法识别 What 等一系列字符，因为这些字符既不是字符串也不是标量。正确的用法是在单引号前后加双引号：

```
local question "What is the anwser? "
local answer "42."
di "`question'" "`answer'"
```

如此最后一条命令就被翻译成：「di "What is the anwser?" "42."」，即显示两个字符串，程序正确运行。

实际上我们可以在程序的任何位置上使用宏进行替代，程序运行的方式总是先将宏替换掉，再重新解析命令，比如我们可以使用宏把正常的命令改的面目全非：

```
clear
set more off
set obs 100
```

```
scalar p=0.5
gen x=runiform()
gen y=rnormal()
local Knowledge "scatter"
local is "y"
local power "x"
local Francis "scale(scalar(p))"
local Bacon `"title("Knowledge is power.")"'
`Knowledge' `is' `power', `Francis' `Bacon'
```

在上面程序中，我们试图生成两个随机数，并画出他们的散点图，要求散点的大小仅仅是正常大小的一半，我们使用不同的宏代替了不同的部分，最后将这些宏组织在一起。尤其需要注意的是，在 Bacon 这个宏中，由于宏的值「title("Knowledge is power.")」中出现了双引号，如果不做特殊处理会产生歧义，因而我们使用了「`"」和「"'」作为字符串的声明。

实际上我们不仅仅可以将宏定义为字符串，也可以将其赋值为数值，即直接在宏名称后面加等于号，当然，这些数值也可以使用函数运算：

```
local a=sqrt(3)
di `a'
di "`a'"
```

然而实际应用中一定要注意的是，如果希望把宏当做数值进行运算，那么一定要使用等号，否则 Stata 不会将其看作是数字，比如：

```
local a=2+2
local b 2+2
di 4*`a'+2
di 4*`b'+2
```

以上代码中，a 和 b 的区别仅仅在于等号的有无，然而结果是完全不一样的，使用 a 的结果为 18，而使用 b 的结果为 12，因而需要特别注意。

当宏中存储的为数值时，我们还可以使用自增（减）操作符「++」、「- -」，即执行「++i」等价于执行「local i=`i'+1」，「- -i」等价于执行「local i=`i'-1」。但是需要注意的是，「i++」、「i- -」是不可用的。

此外，宏的引用支持嵌套，比如：

```
local s1 "string1"
local s2 "string2"
local n=2
di "`s`n''"
```

最后一行代码中，Stata 会自动首先将 ‘n’ 用 2 代替，再将 s2 用 string2 代替，因而最终结果是显示「string2」。

最后，宏还有一个重要的应用就是为变量、标量、矩阵、文件等提供临时的名字。前面已经提到过标量名可能被识别成变量名的问题，对于一些比较复杂的数据以及一些比较复杂的程序来说，变量、标量等的名称重复是一个很重要的潜在威胁，可能会导致一些不可预期的结果。为此，Stata 有三条命令可以帮助解决此问题：

- tempvar：产生一个局部宏，包含了一个临时的变量名；
- tempname：产生一个局部宏，包含了一个临时的标量或者矩阵名；
- tempfile：产生一个局部宏，包含了一个临时的文件名；

使用以上的三条命令可以避免重名的情况发生，但是需要注意的是，这些临时变量名在程序结束时都将被清除。比如：

```
clear
set obs 100
tempname a
tempvar x
di "`x'"
scalar `a'=sqrt(_pi)
gen `x'=1/`a'
gen x=`x'^2
```

将以上程序写在 do 文件中执行，当执行结束以后，只会产生一个新的变量 x，其他的临时标量、临时变量都不复存在。注意以上的临时变量 ‘x’是一个局部宏，其值并不等于”x”，实际上通过 di ”‘x’” 可以发现，其值应该为「___000001」之类的名字。

值得注意的是，如果使用 tempname 为一个宏命名，那么需要使用嵌套的 “” 才能取得宏的值，比如：

```
tempname a
local `a'=3
di ``a''^2
```

在 di 命令中，最内层的 ‘a’ 取得了宏「a」的值，即一个临时名字。如果需要使用 ‘a’ 的值，外面还需要再嵌套一层，即 “a” 代表了宏 ‘a’ 的值。

1.7 条件与循环语句

在任何程序语言中，对于程序流程的控制，比如条件、循环等都是必不可少的。在 Stata 中自然也提供了程序流程控制的相关命令。

1.7.1 条件

在 Stata 中，条件语句由「if」和「else」语句组成，其基本语法为：

```
if exp {
  commands
}
else {
  commands
}
```

当然，else 可以不出现。其中 exp 是一个逻辑语句，如果 exp 为真，那么就会执行大括号里面的语句，否则执行 else 大括号里面的语句。注意在 Stata 中，左大括号后面不能跟任何字符（注释、空格除外），且右大括号必须单独一行。Stata 中没有「elseif」或者「elif」等语句，可以使用：

```
if exp {
  commands
}
else if{
  commands
}
else {
  commands
}
```

完成同样的功能。

比如，下面的程序完成了一个简单的判断-输出的功能：

```
local a=42
if `a'==42 {
  di "Yes, it is the answer!"
}
else if `a'>42 {
  di "Too big..."
}
else {
  di "Too small..."
}
```

1.7.2 循环

Stata 中有三种循环语句:「while」、「foreach」和「forvalues」。我们将分别介绍。

「while」语句的语法为:

```
while exp {
  commands
}
```

与 if 语句一样，exp 为逻辑语句，当为真时即继续执行大括号里面的程序，否则跳出循环。比如下面的程序计算了 $\sum_{i=10}^{20} i$ 的值:

```
local i=9
local sum=0
while `++i'<=20 {
  di `i'
  local sum=`sum'+`i'
}
di `sum'
```

在以上的程序中，「`++i'」实现了自增运算，程序执行时先自增 1，再与 20 进行比较，只要小于等于 20，就会执行大括号里面的加法。

在循环中,还可以使用「continue」命令来控制循环继续执行,使用「continue, break」命令来控制循环退出，比如上面的程序等价于:

```
local i=10
local sum=0
while 1 {
  if `i'<=20 {
    local sum=`sum'+`i'
    local i=`i'+1
    continue
  }
  else {
    continue, break
  }
  di `i'
}
di `sum'
```

上述程序中，使用「while 1」创建了一个无限循环，循环的继续和退出使用「continue」和「continue, break」命令来控制。当 i<=20 时，coutinue 控制程

序继续执行，否则退出。注意到，当程序遇到「continue」时，会立刻结束本次循环，进入下一次循环，碰到「continue, break」时会立刻停止循环，因而第 12 行代码永远不会执行。如果删掉「continue」，程序也会给出正确的结果，但是将会打印出 11 个数字（为什么是 11 个?）。

以上程序还可以使用「forvalues」循环语句来实现，其基本语法为：

```
forvalues lname = range {
  commands
}
```

其中 lname 为一个局部宏，range 为一个整数区间，其语法可以为：

- $\#_1\,(\#_d)\,\#_2$：从 $\#_1$ 到 $\#_2$，每次增加 $\#_d$
- $\#_1/\#_2$：从 $\#_1$ 到 $\#_2$，每次增加 1，等价于 $\#_1\,(1)\,\#_2$
- $\#_1\ \ \#_t\ \ to\ \ \#_2$ 或者 $\#_1\ \ \#_t\ \ :\ \ \#_2$：从 $\#_1$ 到 $\#_2$，每次增加 $\#_t - \#_1$

比如上面的求和程序可以写为：

```
local sum=0
forvalues i = 10/20 {
  local sum=`sum'+`i'
}
di `sum'
```

当然每次增加的量也可以为负数，比如如下写法都是等价的：

- forvalues i=20(-1)10
- forvalues i=10 11 : 20
- forvalues i=20 19 : 10

此外「continue」和「continue, break」命令都可以用在「forvalues」循环中。

最后，Stata 还提供了「foreach」循环，允许将一个列表作为循环的来源。一个列表就是一个以空格分隔开的元素的集合，比如：

```
foreach a in 1 2 3 4 5 {
    di `a'^2
}
```

即将列表 {1,2,3,4,5} 的每一个元素分别循环一遍。当然，列表的元素也可以不是数字，字符串、变量名等都可以当做列表的元素：

```
foreach a in a b c d e f {
    di "`a'"
}
```

更一般的写法是先把列表写在一个宏里面，比如上面的程序可以写为：

```
local strlist "a b c d e f"
foreach a in `strlist' {
    di "`a'"
}
```

在以上程序中，我们先定义了一个局部宏 strlist，再在 foreach 语句中引用这个局部宏，因而达到了跟之前程序一样的效果。

Stata 对宏、变量名等的 foreach 语句做了进一步优化，比如对于上面的宏，我们可以写为：

```
local strlist "a b c d e f"
foreach a of local strlist {
    di "`a'"
}
```

实际效果是一样的。注意在上面的程序中，「in」变成了「of」，同时使用 local 声明后面的 strlist 是一个局部宏。对于全局宏，也可以使用「foreach of global」的语法。使用这样的语法执行速度将会比直接使用引用的形式要快。

有的时候我们需要对一些分类变量的值进行遍历，这个时候可以使用「levelsof」命令取得这些变量可能的取值。比如，在「cfps_adult.dta」文件中，我们可以使用：

```
use cfps_adult.dta
levelsof te4, local(edu)
di "`edu'"
```

获得变量 te4 的所有可能的取值，并保存在局部宏 edu 中。因而我们也可以使用如下命令生成 te4 的虚拟变量：

```
use cfps_adult.dta
drop if te4<0
levelsof te4, local(edu)
foreach v of local edu{
  gen edu`v'=te4==`v'
}
```

在以上程序中，首先删除了 te4<0 的异常值，然后使用「levelsof」命令获得了 te4 所有可能的取值，再对 te4 所有可能的取值进行循环，分别产生相应的虚拟变量。以上循环实际上等价于「tab te4, gen(edu)」。

此外，如果需要循环的是一个变量列表，那么可以用「foreach of varlist」进行优化。使用该命令的好处是 varlist 可以使用通配符。比如在「cfps_adult.dta」中，有四个关于「您是哪些组织成员」的变量：「qn401_s_1 - qn401_s_4」，不

同变量之间可能重合，比如既属于「共产党」，又属于「工会」。以下程序展示了如何产生「您是哪些组织成员」的虚拟变量：

```
use cfps_adult.dta
local zuzhi ""
foreach q of varlist qn401_s_*{
  levelsof `q', local(lqn)
  foreach l of local lqn{
    local exist_in_zuzhi=0
    foreach z of local zuzhi{
      if `l'==`z'{
        local exist_in_zuzhi=1
        continue, break
      }
    }
    if `exist_in_zuzhi'==0{
      local zuzhi "`zuzhi' `l'"
    }
  }
}
foreach v of local zuzhi{
  if `v'>0{
    gen zuzhi`v'=qn401_s_1==`v'
    label variable zuzhi`v' "qn401_s==`v'"
    foreach q of varlist qn401_s_2 - qn401_s_4{
      replace zuzhi`v'=1 if `q'==`v'
    }
  }
}
```

在以上程序中有两个大循环，其中第一个大循环的目的是为了获得四个变量所有可能的取值，而第二个循环是为了创建虚拟变量。注意在两个循环中，我们分别使用了一次「foreach of varlist」，其中「varlist」都是由变量的通配符完成的。当然，以上的第二个循环可以使用「egen anymatch」命令代替。

除了「foreach of local」和「foreach of varlist」之外，Stata 还有「foreach of global」、「foreach of newlist」（用于新变量名）、「foreach of numlist」，具体使用可以查看「help foreach」。

1.8 程序

在 Stata 中，一个程序（program）就是一个完成特定功能的代码段。使用程序可以将经常碰到的功能以固定的形式保存下来，方便以后使用，提高代码重用率。此外，程序还是接下来要学习的做「模拟」以及其他高级编程的基础，因而接下来我们学习一些简单的程序的写法。

1.8.1 程序入门

在 Stata 中，程序使用命令「program [define]」进行声明，并以 end 结尾。比如，一个最简单的程序：

```
program answer
  di "42."
end
```

接下来，直接使用「answer」命令，就可以直接打印出「42.」。s

不过，如果已经定义过「answer」命令，试图再次定义这个命令时，会提示「program answer already defined」，即「answer」命令已经被定义过，不能再重新定义。这时我们需要使用「program drop」命令将「answer」命令删掉再重新定义。然而在 do 文件中，如果在第一次定义「answer」命令之前执行用「program drop answer」也会报错，因而我们会在前面加上「capture」命令，忽略这个错误。因而，以上程序在 do 文件中完整的写法应该是：

```
cap program drop answer
program answer
  di "42."
end
```

1.8.2 语法解析

以上展示了一个简单的程序的书写。绝大多数情况下，程序必须和现有的环境交互，因而必须向程序中传递参数。下面我们将分别介绍 Stata 程序书写中参数传递的两种方式。

1.8.2.1 通过参数传递

第一种方法是通过参数进行传递。类似 Linux 中的 Shell 编程，在 Program 中，可以使用宏「'#'」获取传递的参数，其中 # 为数字或者「'*'」。如果传递的是数字，那么「'#'」指代第 # 个参数，比如「'1'」指代第一个参数，「'2'」指代第二个参数，以此类推。此外，「'0'」指代用户输入的所有参数（包括多余的空格等），而「'*'」指代的同样是所有参数，不同的是删掉了重复的空格等等。以下程序展示了其用法：

```
cap program drop test_arg
program test_arg
  di "`0'"
  di "`*'"
  di "`1'"
  di "`2'"
  di "`3'"
end
test_arg a b  c d
```

上述程序运行后，将得到显示结果：

```
a b  c d
a b c d
a
b
c
```

在以上程序中，首先定义了一个新的程序 test_arg，其功能是将传递的参数打印出来，比如「`1'」对应用户输入的第一个参数「a」，以此类推。注意到「`0'」与「`*'」打印结果的差别（中间的空格）。

比如，以下代码很简单的产生了两个变量的平方的加权平均：

```
cap program drop weig_aver
program weig_aver
  tempvar v1 v2
  gen `v1'=`2'^2
  gen `v2'=`3'^2
  gen `1'=`v1'*`4'+`v2'*(1-`4')
end

use "cfps_adult.dta"
weig_aver aver_expn qg12 qg1203 0.5
```

以上程序中，产生了一个新的变量「aver」（「`1'」对应），等于「qg12」和「qg1203」（「`2'」和「`3'」对应）的平方的加权平均，权重为 0.5（「`4'」对应）。

当然，以上语法看起来并不便于理解，所以我们经常会给这些参数命名。使用 args 命令可以为参数进行命名，比如上述程序可以改写为：

```
cap program drop weig_aver
program weig_aver
  args newvar var1 var2 w
```

```
  tempvar v1 v2
  gen `v1'=`var1'^2
  gen `v2'=`var2'^2
  gen `newvar'=`v1'*`w'+`v2'*(1-`w')
end

use "cfps_adult.dta"
weig_aver aver qg12 qg1203 0.5
```

通过 args 命令，相当于为「‘1’」、「‘2’」、「‘3’」等宏进行命名，程序更加易读。

此外，注意到我们在程序中使用了「tempvar」命令，该命令声明了两个宏 v1 和 v2，其内容为两个变量名。使用「tempvar」命令有如下两个好处：

1. ‘v1’ 和 ‘v2’ 不会与现有的变量名起冲突；
2. 「gen」命令产生了两个新变量 ‘v1’ 和 ‘v2’ 在程序结束以后会被自动删除，不会留下多余的中间变量。

因而，基于以上两点，在程序中使用「tempvar」包括「tempname」、「tempfile」等是非常良好的习惯。

此外，在 Stata 中还有一个非常有用的命令「macro shift」，该命令的作用是删掉「‘1’」，并将「‘2’」变为「‘1’」，「‘3’」变成「‘2’」等等，「‘*’」也将随之改变，比如：

```
cap program drop test_arg
program test_arg
  di "`0'"
  di "`*'"
  di "`1'"
  di "`2'"
  macro shift 1
  di "`0'"
  di "`*'"
  di "`1'"
  di "`2'"
end
test_arg a b  c d
```

将显示结果：

```
a b  c d
a b c d
a
```

```
b
a b  c d
b c d
b
c
```

这条命令通常用来区分因变量和自变量，比如以下程序使用「macro shift」命令区分了左手边的因变量和右手边的自变量，并进行了一个简单的回归：

```
cap program drop test_reg
program test_reg
  local dep "`1'"
  macro shift 1
  local control "`*'"
  reg `dep' `control'
end

use "cfps_adult.dta"
test_reg p_income qq1102 qp101 qp102
```

运行以上程序可以看到，通过「macro shift」命令，「p_income」被赋值为 'dep'，而「qp101 qp102」则被赋值为 'control'。

1.8.2.2 标准语法

除了以上的方法之外，Stata 还提供了更加规范的标准语法。我们在程序中可以声明的语法形式为：

```
cmd [varlist | namelist | anything]
    [if] [in] [using filename] [=exp]
    [weight] [,options]
```

其中：

- cmd：命令的名称
- varlist：变量列表，也可以为 varname（单独一个变量）、newvarlist（新变量列表）、newvarname（一个新变量）。varlist 后面可以加一些设定附在 varname 的后面，比如默认值（default=）、变量列表最少个数（min=#）、最大个数（max=#）等。实际上，「varname」和「varlist(max=1)」是等价的。
- namelist：名称列表。由于 varlist 只允许变量，不允许一般的宏的名字等等，此时需要 namelist。同样，「name」是「namelist(max=1)」的简称。

- anything:一般用来解析比较复杂的与标准语法相似的结构。使用 anything 之后，if、in 等接下来的设定都将看成是 anything 的一部分。
- options: 用户自定的选项，可以为整数 (integer，没有小数点)、实数 (real，有小数点)、数字列表 (numlist)、变量列表 (varlist)、名称列表 (namelist)、字符串 (string) 以及命令开关等。

其他关于 if、in 等的解释可以查看「hlep syntax」。我们通过下面的示例来展示使用「syntax」命令声明程序的语法:

```
cap program drop test_syntax
program test_syntax
  version 15
  syntax varlist(max=1), var1(varlist min=1)/*
     */ [optional integ(integer 1) scale(real 0.5)]
  di "`varlist'"
  di "`var1'"
  di "`optional'"
  if "`optional'"=="optional" {
    di "optional is set on."
  }
  else {
    di "optional is set off."
  }
  di "`integ'"
  di "`scale'"
end

use "cfps_adult.dta"
test_syntax p_income, var1(qq1102)
test_syntax p_income, var1(qq1102 qp101) /*
          */optional integ(2) scale(1.2)
```

注意到上面 syntax 语句中的方括号「[]」表示可选项，即在使用命令时不必须提供此项，因而选项「optional」、「integ」、「scale」三个都是可选的。其中:

- syntax 后面紧跟的 varlist 被设置为最多只能提供一个变量，等价于使用 varname，在程序内部可以使用「`varlist'」指代提供的变量;
- var1 由于不在方括号中，因而是必须要提供的选项，varlist 代表 var1 括号在要提供一个变量列表，「min=1」设定了至少要提供 1 个变量，在程序内部可以使用「`var1'」指代 var1() 括号内提供的变量列表;

- optional 是可选选项，这是一个开关，类似「save, replace」中的 replace 选项，只要程序使用时提供了这个选项，就被设置为 on，此时程序内部"`optional'"=="optional"，因而在程序内部我们使用「if "`optional'"=="optional"」判断该选项是否提供；
- integ 也是可选选项，括号中 integer 代表该选项接受一个整数，后面的 1 代表如果用户不提供，则默认为 1。
- scale 也是可选项，括号中的 real 代表该选项接受一个实数，默认值为 0.5。

最终程序运行结果如下：

```
. test_syntax p_income, var1(qq1102)
p_income
qq1102

optional is set off.
1
.5

. test_syntax p_income, var1(qq1102 qp101) /*
>           */optional integ(2) scale(1.2)
p_income
qq1102 qp101
optional
optional is set on.
2
1.2
```

最后，如果希望在标准语法的设定下使用参数传递 `1', `2', ... 的方式，可以使用命令「tokenize `varlist'」来实现。

1.8.3 返回值

上面讨论了如何向程序输入参数的问题，接下来我们讨论程序如何输出的问题。

程序有可能输出多种不同的结果，比如在上面的示例中，程序输出的最常用办法是打印出结果，当然这是最直观，但是在多数情况下并不是最方便的方法。此外，我们上面还展示了在程序中可以生成新的变量，这也是程序输出的常见形式。

Stata 的程序是可以带返回值的，比如我们前面介绍的 r()、e() 类型的结果就是程序的返回值。只不过，在 Stata 中，有多种不同类型的返回值，比如 r-class、e-class、s-class、n-class 等。在这里，我们将重点介绍 r-class 和 e-class。

1.8.3.1 r-class

r-class 是最简单的返回值，可以返回标量（scalar）、矩阵（matrix）、局部宏（local）等。在一条 Stata 命令（比如 su）结束以后，可以使用「return list」命令显示出该命令所有的返回值。

而在程序中返回 r-class 的结果是非常直接的，在程序中任何地方直接使用「return scalar」、「return matrix」等命令即可返回结果。与其他语言不同的是，运行「return」命令并不会导致程序的退出，因而可以一直不断的使用「return」命令返回不同的结果。不过，如果希望使用「return」命令返回结果，需要在使用「program」命令定义程序的语句后面加入「rclass」选项，以标明这个程序会返回 r-class 的结果。

在以下的程序示例中，我们展示了如何计算加权的百分位数，并将百分位数保存在 r(p#) 中：

```
cap program drop weight_pctl
program weight_pctl, sortpreserve rclass
  version 15
  syntax varlist(max=1), weight(varname)/*
        */ p(numlist integer >0 <100 sort)
  tempvar cum_weight stand_weight sum_weight
  egen `sum_weight'=total(`weight') if `varlist'~=.
  gen `stand_weight'=`weight'/`sum_weight' if `varlist'
      ~=.
  sort `varlist'
  gen `cum_weight'=`stand_weight' if _n==1
  replace `cum_weight'=`cum_weight'[_n-1]+/*
        */`stand_weight'[_n] if _n>1
  tempname N i
  local `N'=_N
  local `i'=0
  foreach n of numlist `p'{
    while `++`i'' <= ``N''{
      if `cum_weight'[``i'']>=(`n'/100){
        return scalar p`n'=`varlist'[``i'']
        continue, break
      }
    }
  }
end

use "cfps_family_econ.dta", clear
```

```
weight_pctl fincome1, weight(familysize) p(25 33 50 67
    75)
di r(p25) "—" r(p33) "—" r(p50) "—" r(p67) "—" r(p75)
su fincome1 [fweight=familysize], de
```

在以上程序中，使用「program」定义时，我们除了使用「rclass」标明本程序会返回 r-class 结果之外，还加入了「sortpreserve」选项，因为我们在程序中需要对数据排序，加入这个选项之后可以在程序结束之后保证排序不变；在 syntax 语句中，选项 p() 被声明为一个数字列表，需要是一个大于 0 小于 100 的整数。计算过程为，我们首先对 x_i 排序，令 $x_{(i)}$ 为排序之后的次序统计量，$w_{(i)}$ 为其对应的权重（比如在上例中，我们使用了家庭规模作为权重，即一个家庭代表三个人，那么 $w_i = 3$），首先我们先把权重标准化：

$$w^*_{(i)} = \frac{w_{(i)}}{\sum_{i=1}^{N} w_{(i)}}$$

如此标准化之后的权重总和为 1。接下来，我们计算了权重的累积和，即令：

$$cw_{(i)} = \sum_{\tau=1}^{i} w^*_{(\tau)}$$

给定一个整数 p，第一个使得 $cw_{(i)} \geq \frac{p}{100}$ 的即是 p 百分位数。比如，第一个使得 $cw_{(i)} \geq 0.5$ 的 x 即为加权之后的中位数。程序的最后，针对 numlist 'p' 的每一个整数，都计算其分位数，并使用「return scalar」将结果返回到「p'n'」中。

以上示例中使用了 CFPS 家庭经济情况的数据，按照家庭规模（同灶吃饭的人口）进行加权，计算了五个加权分位数，最后与 su 命令经过 fweight 加权之后的结果进行比较，是相同的。值得注意的是，加权的结果比不加权的结果更高，这是由于家庭收入与家庭规模的正相关性所导致的。

1.8.3.2 e-class

与 r-class 类似，e-lcass 用于返回程序的运算结果，而不同的是，e-class 专门用于返回参数估计的结果。

在参数估计中，参数估计的数值以及其标准误是通常最关心的两组参数。常用的参数估计方法，如矩估计、极大似然估计以及 M-估计等，其参数估计结果都是渐进正态的，因而一旦计算得到参数数值及其标准误，那么很容易的就可以构造其置信区间以及假设检验。而为了计算标准误，以及进行参数之间的假设检验等，需要首先计算参数之间的协方差矩阵。e-class 通过「ereturn」命令在程序中返回估计的系数（b）以及其协方差矩阵（V），其基本用法为：

```
//估计步骤
ereturn clear
ereturn post b V
```

```
ereturn display
```

其中「ereturn clear」用于将内存中的估计结果清空;「ereturn post」用于给出参数估计的结果（$1 \times K$ 维向量 b）以及协方差矩阵（$K \times K$ 维矩阵 V）;「ereturn display」用于展示参数估计的结果，即常见的 Stata 估计表格，包括标准误、t 值、p 值、95% 置信区间等。

需要注意的是，b 和 V 的行和列名必须相对应，一般为对应变量的名字，必要时可以使用「mat colnames」以及「mat rownames」修改行和列的名称。

此外，ereturn 命令还可以返回其他的标量、宏、矩阵等，比如「ereturn local depvar」指被解释变量,「ereturn scalar N=」通常指样本量等。可以使用「ereturn list」命令列出参数估计后内存中所有使用 ereturn 命令返回的结果。

以下给出了一个工具变量中计算两阶段最小二乘的程序，在使用两阶段最小二乘计算的基础上，由于第二阶段回归的标准误错误，因而继续调整标准误至正确的标准误，最后使用 ereturn 命令返回估计结果:

代码 1.1: Stata 程序示例：两阶段最小二乘

```
// 2SLS by hand
cap program drop twosls
program twosls, eclass
  version 15.0
  syntax varlist(max=1), endo(varlist) instru(varlist) /*
    */control(varlist)
  tempname dep Xhat XhhX pred_endo pred_var b sigma2 V
  tempvar resid
  local `dep' "`varlist'"
  quietly{
    // get the predicted value of endo x
    local `pred_var' ""
    foreach x of varlist `endo'{
      reg `x' `instru' `control'
      predict `pred_endo'_`x'
      local `pred_var' "``pred_var'' `pred_endo'_`x'"
    }
    local `Xhat' "``pred_var'' `control'"
    // ols of y on Xhat, get b
    reg ``dep'' ``Xhat''
    matrix `b'=e(b)
    mat colnames `b'= `endo' `control' _cons
    ///// compute variance under homoskedasticity
    // first compute residuals
```

```
    gen `resid'=``dep''
    foreach x of varlist `endo' `control'{
      replace `resid'=`resid'-`b'[1,colnumb(`b',"`x'")]*`
          x'
    }
    replace `resid'=`resid'-`b'[1,colnumb(`b',"_cons")]
    su `resid'
    scalar `sigma2'=r(Var)
    // then compute XhhX
    matrix accum `XhhX'=``Xhat''
    matrix `V'=`sigma2'*invsym(`XhhX')
    mat colnames `V'= `endo' `control' _cons
    mat rownames `V'= `endo' `control' _cons
    drop ``pred_var''
  }
  ereturn clear
  ereturn post `b' `V'
  ereturn local depvar "``dep''"
  ereturn display
end
// test the program
clear
set more off
set obs 1000
// generate fake data
gen z=rnormal()
gen u=rnormal()
gen x1=z+rnormal()+u
gen x2=z+0.5*rnormal()
gen y=2+x1+x2+u
// compare it with ivregress commmand
ivregress 2sls y (x1=z) x2
twosls y, endo(x1) instru(z) control(x2)
```

在以上程序中，我们首先使用第一阶段回归得到了内生变量的预测值，进而使用：

$$\mathbb{V}\left(\hat{\beta}^{2SLS}\right)=\hat{\sigma}^2\left(\hat{X}'\hat{X}\right)^{-1}$$

其中 $\hat{X}=Z\left(Z'Z\right)^{-1}Z'X$ 为对内生变量和其他控制变量的预测值，而 $\hat{\sigma}^2$ 为残差：$\hat{e}=Y-X'\hat{\beta}^{2SLS}$ 的样本方差。最后我们将手动实现的两阶段最小二乘与

Stata 自带的两阶段最小二乘估计量进行比对以验证程序的正确性。

1.8.4 示例：Gini 系数的计算

以下程序展示了一个计算 Gini 系数的 Stata 程序示例。

代码 1.2: Stata 程序示例：计算基尼系数

```
cap program drop ginindex
program define ginindex, byable(onecall)
  version 15
  syntax varlist(max=1), gen(name)
quietly{
  if "`_byvars'"==""{
    tempvar class
    gen `class'=1
    local _byvars "`class'"
  }
  // 临时变量
  tempvar rank totalincome cum rat num rect B delta
  // 排序并产生排序变量
  sort `_byvars' `varlist'
  by `_byvars': gen `rank'=_n
  // 产生地区户数、总收入和累积收入及比例
  egen `num'=count(`varlist'), by(`_byvars')
  egen `totalincome'=total(`varlist'), by(`_byvars')
  gen `cum'=`varlist' if `rank'==1
  by `_byvars': replace `cum'=/*
          */ `cum'[_n-1]+`varlist'[_n] if `rank'>1
  gen `rat'=`cum'/`totalincome'
  // 计算每个个体的柱形面积
  by `_byvars': gen `rect'=`rat'/`num'
  // 调整小样本误差，即减去柱子最上面小三角的面积
  gen `delta'=`rat'/2 if `rank'==1
  by `_byvars': replace `delta'=/*
          */ `rat'[_n]-`rat'[_n-1] if `rank'>1
  replace `rect'=`rect'-`delta'/`num'/2
  //最终的指数
  egen `B'=total(`rect'), by(`_byvars')
  gen `gen'=1-2*`B'
}
end
```

```
35
36 use "cfps_family_econ.dta", clear
37 ginindex fincome1, gen(gini_all)
38 bysort provcd14: ginindex fincome1, gen(gini)
```

1.9 高级编程

1.9.1 简单矩估计

对于简单的矩估计，我们可以使用 Stata 中的广义矩估计命令「gmm」来实现。

在矩估计中，最重要的是找到矩条件（moment conditions）。一般的，如果我们关心真实参数 $\theta_0 \in \Theta \subset \mathbb{R}^K$，只要我们可以找到 K 个矩条件，使得 θ_0 为矩条件方程：

$$\mathbb{E}\left[g\left(w_i,\theta\right)\right]=\mathbb{E}\left(\left[\begin{array}{c} g_1\left(w_i,\theta\right) \\ \vdots \\ g_K\left(w_i,\theta\right) \end{array}\right]\right)=0$$

的唯一解，那么我们就可以解以上总体矩方程组的样本方程等价形式：

$$\frac{1}{N}\sum_{i=1}^{N} g\left(w_i,\hat{\theta}\right)=0$$

解得 $\hat{\theta}$。

在 Stata 中，gmm 命令用于进行简单的矩估计的最简单语法为：

```
1 gmm (rexp1) (rexp2) ... , winitial(identity) [options]
```

其中 rexp1 等是我们设置的矩条件；winitial(identity) 对于简单的矩估计而言是一个固定的选项，我们暂且不管，在学习广义矩估计时再讨论；options 为一些其他选项。

比如，对于正态分布，如果 $x_i \sim N\left(\mu,\sigma^2\right)$，我们知道其前两阶矩分别为：

$$\begin{cases} \mathbb{E}\left(x_i\right)=\mu \\ \mathbb{E}\left(x_i^2\right)=\mu^2+\sigma^2 \end{cases}$$

那么我们可以使用如下代码进行正态分布的矩估计：

```
1 // 正态分布的矩估计
2 clear
3 set more off
4 // 产生正态分布数据，期望为3，方差为25
```

```
set obs 1000
gen x=rnormal(3,5)
gen x2=x^2
// 进行矩估计
gmm (x-{mu}) (x2-{mu}^2-{sigma2}), winitial(identity)
```

以上程序中，我们使用 rnormal() 函数生成了期望为 3，方差为 25 的正态分布，并使用以上矩条件进行了矩估计。

一个更加复杂一点的例子，如果 $x_i \sim Beta(\alpha, \beta)$，我们知道其前两阶矩分别为：

$$\begin{cases} \mathbb{E}(x_i) = \frac{\alpha}{\alpha+\beta} \\ \mathbb{E}(x_i^2) = \frac{\alpha\beta+\alpha^2(\alpha+\beta+1)}{(\alpha+\beta)^2(\alpha+\beta+1)} \end{cases}$$

那么估计 Beta 分布参数的代码为：

```
// Beta分布的矩估计
clear
set more off
// 产生Beta分布数据
set obs 1000
gen x=rbeta(0.5,3)
gen x2=x^2
// 矩估计
gmm (x-{alpha}/({alpha}+{beta}))/*
 */ (x2-({alpha}*{beta}+{alpha}^2*({alpha}+{beta}+1))/(({
    alpha}+{beta})^2*({alpha}+{beta}+1)))/*
 */, winitial(identity) from(alpha 1 beta 1)
```

在以上的程序中，我们使用 rbeta 函数生成了 $\alpha = 0.5, \beta = 3$ 的 1000 个 Beta 分布随机数，接着使用 gmm 命令完成了矩估计。值得注意的是，由于 α, β 的取值范围为 $(0, \infty)$，不能取 0 值，而 Stata 中默认的最优化算法初始值就是 0，所以我们使用了 from() 选项强制初始值都为 1。

1.9.2 广义矩估计与工具变量

以上我们在做简单的矩估计时，使用了 gmm 命令。其实 gmm 是**广义矩估计（generalized method of moments， GMM）**的命令，简单的矩估计是广义矩估计的一个特殊情况。

与简单的矩估计相比，广义矩估计广义在矩条件的个数上。在简单的据估

计中，对于 K 个参数，我们使用 K 个矩条件：

$$\mathbb{E}\left[g\left(w_i,\theta\right)\right]=\mathbb{E}\left(\left[\begin{array}{c} g_1\left(w_i,\theta\right) \\ \vdots \\ g_K\left(w_i,\theta\right) \end{array}\right]\right)=0$$

联立这 K 个方程就可以解出 θ。而广义矩估计则扩充了矩条件的个数，即矩条件为：

$$\mathbb{E}\left[g\left(w_i,\theta\right)\right]=\mathbb{E}\left(\left[\begin{array}{c} g_1\left(w_i,\theta\right) \\ \vdots \\ g_G\left(w_i,\theta\right) \end{array}\right]\right)=0$$

其中 $G\geq K$。我们这里假设识别条件，即真实参数 θ_0 是使得 $\mathbb{E}\left[g\left(w_i,\theta_0\right)\right]=0$ 的唯一解。

扩充矩条件个数带来的问题是如果方程个数大于未知参数个数，那么方程：

$$\frac{1}{N}\sum_{i=1}^{N}\left[g\left(w_i,\hat{\theta}\right)\right]=0$$

可能是无解的。比如，我们知道如果 $x_i\sim N\left(\mu,\sigma^2\right)$，那么 $\mathbb{E}\left(x_i^3\right)=\mu^3+3\mu\sigma^2$，那么我们可以使用其前三阶矩：

$$\begin{cases}\mathbb{E}\left(x_i\right)=\mu \\ \mathbb{E}\left(x_i^2\right)=\mu^2+\sigma^2 \\ \mathbb{E}\left(x_i^3\right)=\mu^3+3\mu\sigma^2\end{cases}$$

进行估计。然而，其样本形式的方程：

$$\begin{cases}\frac{1}{N}\sum_{i=1}^{N}\left(x_i-\mu\right) & =0 \\ \frac{1}{N}\sum_{i=1}^{N}\left(x_i^2-\mu^2-\sigma^2\right) & =0 \\ \frac{1}{N}\sum_{i=1}^{N}\left(x_i^3-\mu^3-3\mu\sigma^2\right) & =0\end{cases}$$

一般是没有解的，因为我们只需要使用前两个方程就可以解出 μ,σ^2，将其带入第三个方程，第三个方程一般而言不会严格成立（由于随机性的存在）。

为了解决这个问题，我们使用 GMM，即解如下最优化问题：

$$\hat{\theta}=\arg\min_{\theta}\left\{\sum_{i=1}^{N}\left[g\left(w_i,\theta\right)\right]\right\}'W\left\{\sum_{i=1}^{N}\left[g\left(w_i,\theta\right)\right]\right\}$$

其中 W 为一个正定的 $G\times G$ 维权重矩阵。比如，如果 $W=I_{G\times G}$ 即单位阵，那么以上方法相当于最小化了每个矩条件的平方和。比如，在正态分布的例子

中，相当于最小化：

$$\min_{\mu,\sigma}\left[\sum_{i=1}^{N}\left(x_i-\mu\right)\right]^2+\left[\sum_{i=1}^{N}\left(x_i^2-\mu^2-\sigma^2\right)\right]^2+\left[\sum_{i=1}^{N}\left(x_i^3-\mu^3-3\mu\sigma^2\right)\right]^2$$

这样我们可以保证计算出来的参数即使每个矩条件都不一定严格等于 0，但是都尽可能的接近于 0。

如下的例子展示了如何使用三个矩条件对正态分布进行估计：

```
// 正态分布的GMM估计
clear
set more off
// 产生正态分布数据，期望为3，方差为25
set obs 1000
gen x=rnormal(3,5)
gen x2=x^2
gen x3=x^3
// 进行矩估计
gmm (x-{mu}) (x2-{mu}^2-{sigma2}) /*
*/  (x3-{mu}^3-3*{mu}*{sigma2}), winitial(identity)
```

与之前的矩估计相比，差别在于我们加入了第三个矩条件，此外这里也可以解释选项 winitial(identity) 的含义，即使用 $W=I_{G\times G}$ 为初始的权重矩阵。

这里有一个问题，即 W 只能选择单位阵吗？当然不是。实际上任何的正定矩阵都可以，但是不同的 W 会导致估计的 $\hat{\theta}$ 有不同的精度（标准误不同），理论上，能够使得 $\hat{\theta}$ 最精确（最小标准误）的权重矩阵为：

$$W_{opt}=\frac{1}{N}\sum_{i=1}^{N}\left[g\left(w_i,\theta\right)g\left(w_i,\theta\right)'\right]$$

以上权重矩阵成为最优权重矩阵。然而值得注意的是，为了计算最有权重矩阵，我们必须知道 θ，但是 θ 使我们的待估参数，因而最优权重矩阵是未知的。

为了解决这个问题，可以使用两步法，即首先带入初始权重矩阵（比如单位阵）计算，得到 $\hat{\theta}_1$，再将 $\hat{\theta}_1$ 带入到：

$$\hat{W}_{opt}^{(1)}=\frac{1}{N}\sum_{i=1}^{N}\left[g\left(w_i,\hat{\theta}_1\right)g\left(w_i,\hat{\theta}_1\right)'\right]$$

中，得到最有权重矩阵的一个估计，再将 $\hat{W}_{opt}^{(1)}$ 带入到 GMM 的目标函数中，再次计算最优，得到 $\hat{\theta}_2$。我们可以简单的在 gmm 命令的选项中加入 twostep 选项完成两步的 GMM 估计。

或者，将以上过程不断迭代下去，可以得到更好的估计，在 gmm 命令的选

项中加入 igmm 选项即可。当然，不断的迭代直至收敛需要需要的计算量更大。Stata 默认做 twostep，如果不需要 twostep 或者 igmm，而强制使用初始权重矩阵，需要使用 onestep 选项控制。

然而，现实总是比模型更复杂的，我们找到的矩条件并不一定完全正确。比如，如果我们假设数据服从正态分布，但是其实数据为 Beta 分布，那么我们的假设就完全错了。如果矩条件个数大于参数个数，即 $G > K$，我们还可以使用额外的矩条件来检验是否所有的矩条件都是正确的，这个检验也叫做过度识别检验。

比如，在以上正态分布的例子中，如果 x_i 的确服从正态分布，那么根据前两个矩条件得到的估计带入第三个方程中，即使不等于 0，也应该接近于 0。或者等价的，GMM 的目标函数应该是约等于 0 的。GMM 方法的提出者 Hansen 发现，如果权重矩阵使用的是最优权重矩阵，那么 GMM 的目标函数渐进服从卡方分布：

$$J = \left\{ \sum_{i=1}^{N} \left[g\left(w_i, \hat{\theta} \right) \right] \right\}' W_{opt} \left\{ \sum_{i=1}^{N} \left[g\left(w_i, \hat{\theta} \right) \right] \right\} \overset{a}{\sim} \chi^2 \left(G - K \right)$$

因而可以使用该结论对 GMM 的目标函数是否为 0 进行检验。值得注意的是，为了进行该检验，矩条件个数必须大于（不能等于）参数的个数，此外权重矩阵必须为最优权重矩阵。该检验如果显著，则代表目标函数显著不为 0，意味着有的矩条件可能是错误的。当然，如果该检验不显著，也不代表所有的矩条件一定都是对的。为了进行这个检验，在 gmm 命令之后使用「estat overid」即可。

比如，如果我们的数据不服从正态分布，而是 $x_i \sim Beta\left(3,3\right)$ 分布，那么通过正态分布推出的前三阶矩条件：

$$\begin{cases} \mathbb{E}\left(x_i\right) = \mu \\ \mathbb{E}\left(x_i^2\right) = \mu^2 + \sigma^2 \\ \mathbb{E}\left(x_i^3\right) = \mu^3 + 3\mu\sigma^2 \end{cases}$$

就不成立，我们可以使用 Hansen 检验对其进行检验：

```
// 过度识别检验
clear
set more off
// 产生Beta分布数据
set obs 1000
gen x=rbeta(3,3)
gen x2=x^2
gen x3=x^3
// 进行矩估计
gmm (x-{mu}) (x2-{mu}^2-{sigma2}) /*
```

```
*/  (x3-{mu}^3-3*{mu}*{sigma2}), winitial(identity) igmm
// 检验
estat overid
```

在计量经济学的应用中，很多时候我们的矩条件具有如下的形式：

$$\mathbb{E}\left[g\left(w_i,\theta\right)\right]=\mathbb{E}\left(\left[\begin{array}{c} z_{1i}u_i \\ \vdots \\ z_{Gi}u_i \end{array}\right]\right)=0$$

其中一般 u_i 为误差项，而 z_{gi} 为**工具变量（instrumental variable）**。该类矩条件通常来自于条件期望形式：

$$\mathbb{E}\left(u_i|z_{1i},...,z_{Gi}\right)=0$$

只要上式满足，那么矩条件自然满足。

对于此种矩条件，gmm 命令可以方便的写为：

```
gmm (rexp1) (rexp2) ... , instruments(z1 z2 ...)
```

其中 rexp 代表 u_i，而选项 instruments 中填写工具变量。如果 instruments 选项缺失，自动使用 1 作为工具变量。

比如，对于简单的线性回归：

$$y_i=\beta_1+\beta_2\times x_{2i}+\cdots\beta_K\times x_{Ki}+u_i\triangleq x_i'\beta+u_i$$

外生性意味着误差项与 x_i 均值独立，即：

$$\mathbb{E}\left(u_i|x_i\right)=0$$

从而矩条件为：

$$\mathbb{E}\left(\left[\begin{array}{c} u_i \\ x_{2i}u_i \\ \vdots \\ x_{Ki}u_i \end{array}\right]\right)=0$$

一个简单的例子，如下代码展示了如何使用 gmm 命令完成简单的线性回归：

```
use cfps_adult.dta, clear
// 直接回归
reg qp102 qp101 i.cfps_gender
// GMM方法
gmm ( qp102- {xb: qp101 i.cfps_gender _cons}), /*
```

```
*/ instruments(qp101 i.cfps_gender)
```

其中我们使用了「{xb: qp101 i.cfps_gender i.provcd14 _cons}」的形式来代替 $x_i'\beta$（注意不要忘了常数项 _cons）。当然，我们并不是推荐使用 GMM 方法做回归，在此仅仅展示两种方法的等价性。类似的，对于工具变量回归，我们仅仅需要将 instruments 选项中填入工具变量和所有外生变量即可，与以上类似，再次不再赘述。

接下来我们使用一个更加经典而复杂的例子——C-CAPM 展示 GMM 的用法。基于消费的资本资产定价模型是宏观经济学和资产定价领域最经典的结论之一，其基本形式可以使用 Lucas' Tree 模型推导，即：

$$\mathbb{E}\left(1-\beta\left(1+r_{t+1}\right)\left(\frac{C_{t+1}}{C_t}\right)^{-\gamma}|\mathcal{I}_t\right)=0$$

其中 r_t 为收益率，C_t 为消费，$\mathcal{I}_t$ 为 t 期的信息集（比如 $c_t, r_t, c_{t-1}, r_{t-1}, ...$ 等等），β 为贴现因子，γ 为效用函数参数。Hansen and Singleton (1982) 使用 GMM 方法模型进行了估计，根据以上模型结果，所有 t 期的信息都可以作为工具变量。从而，可以使用如下 gmm 命令对如上模型进行估计：

```
use ccapm, clear
tsset qtr
// 计算消费增长率
gen cg=c/L.c
// GMM估计
gmm (1-{beta=1}*(1+F.r)*(F.cg)^(-1*{gamma=1})), /*
*/ instruments(cg L.cg r L.r)
estat overid
```

以上代码中 {beta=1} 以及 {gamma=1} 是为参数设置初始值的另一种方式。

估计结果中，Hansen 检验的结果显著，意味着所使用的矩条件是有问题的，这意味着基本的 C-CAPM 不能完全解释收益率。此外估计的结果显示 $\gamma < 0$ 意味着消费者是风险偏好的，这与标准模型假设有很大偏离，形成了所谓的股权溢价之谜（equity premium puzzle）。

最后，gmm 命令还有大量的细节，比如为 GMM 提供导数、优化算法的控制、多方程估计的使用、多方程不同工具变量的使用、计算最优权重矩阵时的异方差、自相关、聚类调整等等，这些都可以在 Stata 的文档中找到，在此不再赘述。

1.9.3 极大似然估计

在 Stata 中可以使用「ml」命令实现极大似然估计。

极大似然估计最核心的问题是如何写出似然函数：$\ln L(\beta|x)$。在写出似然函数后，接下来的一个重要步骤是使用最优化算法最大化似然函数。在 Stata 中，提供了比较方便的命令处理最大化的问题，在多数简单情况下可以得到比较好的最优化结果，因而最重要的问题是如何提供给 Stata 极大似然函数。

通常情况下，特别是在独立同分布的假设下，似然函数一般可以写为：

$$\ln L(\beta|x) = \sum_{i=1}^{N} l_i(\beta) = \sum_{i=1}^{N} \ln f(x_i|\beta)$$

其中 $f(x_i|\beta)$ 为密度函数。其一阶导数为：

$$\frac{\partial \ln L(\beta|x)}{\partial \beta} = \sum_{i=1}^{N} \frac{\partial \ln f(\beta|x_i)}{\partial \beta} = \sum_{i=1}^{N} g_i$$

此外，通常我们还会将某些参数设定为一些变量的函数，即：

$$\ln L(\beta|x) = \sum_{i=1}^{N} l_i(\beta) = \sum_{i=1}^{N} \ln f(x_i|\beta) = \sum_{i=1}^{N} \ln f(\theta_{i1}, ..., \theta_{iM})$$

其中

$$\theta_{im} = x'_{im}\beta_m$$

称之为第 m 个方程，其一阶导数为：

$$\frac{\partial \ln L(\beta|x)}{\partial \beta_m} = \sum_{i=1}^{N} \frac{\partial \ln f(\theta_{i1}, ..., \theta_{iM})}{\partial \theta_{im}} x_{im}$$

在 Stata 中，有多种提供极大似然函数的方法。包括：

- lf：输入为 θ_m，填补一个新的变量「lnfj」，其值为每个观测的 $l_i(\beta)$
  - 参数：lnfj theta1 theta2 ...
- d0：输入为 β，填补一个标量「lnf」，其值为 $\ln L(\beta|x)$。
  - 参数：todo b lnf
- d1：在 d0 的基础上提供一个向量「g」，其值为似然函数的一阶导数 $\frac{\partial \ln L(\beta|x)}{\partial \beta}$。
  - 参数：todo b lnf g
- d2：在 d1 的基础上再提供一个矩阵「H」，其值为似然函数的海塞矩阵：$\frac{\partial^2 \ln L(\beta|x)}{\partial \beta \partial \beta'}$。
  - 参数：todo b lnf g H

- lf0：输入为 β，填补一个新的变量「lnfj」，其值为每个观测的 $l_i(\beta)$。
  - 参数：todo b lnfj
- lf1：在 lf0 的基础上，填补 M 个向量：$g_{im} = \frac{\partial \ln f(\theta_{i1},...,\theta_{iM})}{\partial \theta_{im}}, m = 1, ..., M$
  - 参数：todo b lnfj g1 g2 ...
- lf2：在 lf1 的基础上再提供一个矩阵「H」，其值为似然函数的海塞矩阵：$\frac{\partial^2 \ln L(\beta|x)}{\partial \beta \partial \beta'}$。
  - 参数：todo b lnfj g1 g2 ... H
- gf0：类似于 lf0，但是不假设独立同分布，比如在面板数据中使用
  - 参数：todo b lnfj

在一些优化算法，如 BFGS 算法中，调用目标函数时可能需要计算导数，也有可能不需要计算导数而只需要计算目标函数值，因而需要使用 todo 这个变量标明是否需要计算导数，如果 'todo'=0 则不需要计算导数，如果需要计算一阶导数，则 'todo'=1。

在定义了目标函数后，可以使用「ml model」命令定义极大似然的问题，该命令需要指定：

1. 使用的方法，如 lf, do,... 等
2. 定义的目标函数
3. 极大似然所需要的数据，以「方程」的形式提供，其语法为：「([equname:] [varlist y=] [varlist x] [,options])」，其中 equname 为方程名，varlist y 为因变量，varlist x 为自变量，options 为一些选项，如 noconstant 选项制定方程 $x'_{im}\beta_m$ 的 x_{im} 中不包含常数项等。如果需要估计的是单参数，可以直接使用「()」来表示，即只有常数项的方程；或者使用「/paraname」来定义，比如「/alpha」代表需要估计参数 alpha。

以下我们给出一个简单的估计 Beta 分布的示例：

```
// Beta分布的极大似然估计
clear
set more off
// Beta分布的目标函数
cap program drop betaobj
program betaobj
  args lnfj theta1 theta2
  quietly{
    replace `lnfj'=log(betaden(`theta1',`theta2',x))
```

```
  }
end
// 产生Beta分布数据
set obs 1000
gen x=rbeta(0.5,3)
// 定义极大似然问题
ml model lf betaobj /alpha /beta
// 寻找初始点，可选
ml search
// 最大化 并得到结果
ml maximize
```

在以上的程序中，我们首先定义了一个程序「betaobj」，该程序接受「lnfj theta1 theta2」三个参数，其中 theta1 theta2 为输入，lnfj 为需要输出的每个观测的似然函数值 $l_i(\theta_1, \theta_2)$。接下来，将 lnfj 变量填充为 Beta 分布密度函数的对数，参数为 theta1 theta2，并对变量 x 计算每个观测的似然函数值。程序之后，我们使用 rbeta 函数生成了 $\alpha = 0.5, \beta = 3$ 的 1000 个 Beta 分布随机数，使用「ml model」命令定义极大似然估计问题，使用「ml search」命令寻找一个比较好的初始值（最优化算法需要，可选，此外还可以使用 ml init 命令制定初始值），最后使用「ml maximize」命令求以上似然函数的最大化，并得到估计结果。

然而以上程序有一个问题，即在程序 betaobj 中，我们直接使用了 Beta 分布的变量名 x，然而现实中需要估计的变量名称可能并不是 x。比如，如果生成 Beta 分布时，使用了 y 的名字：gen y=rbeta(0.5,3)，那么以上程序不能正确执行。为了解决这个问题，我们可以使用方程的形式重写以上的程序：

```
// Beta分布的极大似然估计
clear
set more off
// Beta分布的目标函数
cap program drop betaobj
program betaobj
  args lnfj theta1 theta2
  quietly{
    replace `lnfj'=log(betaden(`theta1',`theta2',$ML_y1))
  }
end
// 产生Beta分布数据
set obs 1000
gen x=rbeta(0.5,3)
// 定义极大似然问题
ml model lf betaobj (alpha: x = , freeparm) /beta
```

```
// 寻找初始点，可选
ml search
// 最大化 并得到结果
ml maximize
```

在以上程序中，「(alpha: x = , freeparm)」使用方程的形式指定了「因变量」x，alpha 为方程名，而 freeparm 选项则指定了方程名 alpha 为一个自由变量。在程序 betaobj 中，为了获得 alpha 这个方程（第一个方程）的因变量名，使用了全局宏 $ML_y1，即指代第一个因变量，同理，$ML_y2, $ML_y2,... 为之后的第二个，第三个因变量名。

以上 Beta 分布的例子仅仅估计了两个参数，实际上，每个参数都可以写成为一些自变量的函数。在这里我们给出一个泊松回归的例子。我们知道泊松分布的密度函数为：

$$f\left(y_i|\lambda\right)=\frac{\lambda^{y_i}}{y_i!}e^{-\lambda}$$

如果我们令 $\lambda_i=\exp\left(x_i'\beta\right)\triangleq\exp\left(\theta_i\right)$[6]，即我们允许不同个体的到达率 λ 可以随着个人特征的改变而改变，那么似然函数为：

$$f\left(y_i|x_i,\beta\right)=\frac{\exp\left(\theta_i y_i\right)}{y_i!}e^{-\exp(\theta_i)}$$

其似然函数为：

$$\begin{aligned}\ln L\left(\beta|x\right)&=\sum_{i=1}^{N}l_i\left(\beta\right)=\sum_{i=1}^{N}\ln f\left(y_i|x_i,\beta\right)\\&=\sum_{i=1}^{N}\left[y_i\theta_i-\exp\left(\theta_i\right)-\ln\left(y_i!\right)\right]\end{aligned}$$

由于 $\ln\left(y_i!\right)$ 不包含未知参数，因而可以省略，代码为：

```
// 目标函数
cap program drop poissonobj
program poissonobj
  args todo b lnfj
  tempvar theta
  mleval `theta'=`b', eq(1)
  quietly{
    replace `lnfj'=$ML_y1 *`theta'-exp(`theta')
  }
end
// 产生数据
```

[6]由于 $\lambda>0$，而 $x_i'\beta$ 可能小于 0，因而通常使用该方式建模。

```
set obs 1000
gen x1=rnormal()
gen x2=runiform()*3-1.5
gen lambda=exp(x1*2+x2+3)
gen y=rpoisson(lambda)
// 定义极大似然问题
ml model lf0 poissonobj (y = x1 x2)
// 寻找初始点，可选
ml search
// 最大化
ml maximize
```

在以上程序中，我们使用了 lf0，注意为了从 β 计算 $\theta_i = x_i'\beta$，我们使用了 mleval 命令，该命令计算方程 eq(#) 的自变量的线性组合。此外在以上程序中，我们使用了 lf0，如果使用 d0，需要使用 mlsum 命令手动加总。

以上程序我们还可以进一步优化。在以上程序中并没有提供似然函数一阶导数的信息，实际上，我们可以计算，其一阶导数为：

$$\begin{aligned}\frac{\partial \ln L(\beta|x)}{\partial \beta} &= \sum_{i=1}^{N} \frac{\partial \left[y_i\theta_i - \exp(\theta_i) - \ln(y_i!)\right]}{\partial \theta_i}\frac{\partial \theta_i}{\partial \beta} \\ &= \sum_{i=1}^{N} \frac{\partial l_i(\beta)}{\partial \theta_i} x_i\end{aligned}$$

实际上，由于 $\frac{\partial \theta_i}{\partial \beta} = x_i$ 总是成立，因而 Stata 中只需要提供 $\frac{\partial l_i(\beta)}{\partial \theta_i}$ 即可，对于每一个方程、每一个观测，$\frac{\partial l_i(\beta)}{\partial \theta_i}$ 为一个标量。计算可得：

$$\frac{\partial l_i(\beta)}{\partial \theta_i} = y_i - \exp(\theta_i)$$

从而，以上程序还可以进一步优化为：

```
// 目标函数
cap program drop poissonobj
program poissonobj
  args todo b lnfj g1
  tempvar theta etheta
  mleval `theta'=`b', eq(1)
  quietly{
    gen `etheta'=exp(`theta')
    replace `lnfj'=$ML_y1 *`theta'-`etheta'
    if (`todo'==0) exit //如果不需要计算导数，退出
    replace `g1'=$ML_y1 - `etheta' //导数
```

```
  }
end
// 产生数据
set obs 1000
gen x1=rnormal()
gen x2=runiform()*3-1.5
gen lambda=exp(x1*2+x2+3)
gen y=rpoisson(lambda)
// 定义极大似然问题
ml model lf1 poissonobj (y = x1 x2)
// 寻找初始点，可选
ml search
// 最大化
ml maximize
```

以上程序中使用 lf1，额外提供了导数信息。此外，如果使用 d1，需要使用 mlvecsum 命令手动加总导数，并将不同方程的导数拼接为 b 的导数。

以上展示了单方程模型的极大似然估计。接下来，我们通过工具变量中的有限信息极大似然（LIML）方法展示如何解决多方程的极大似然估计问题。如果我们有结构方程：

$$y_i = \beta_1 w_i + \tilde{x}_i'\tilde{\beta} + u_i$$

其中 $\mathbb{C}\mathrm{ov}(w_i, u_i) \neq 0$，然而我们有工具变量 $\tilde{z}_i$（可以不止一个），使用所有的外生变量对内生变量做预测，得到方程：

$$w_i = \tilde{z}_i'\delta_1 + \tilde{x}_i'\tilde{\delta} + v_i$$

记 $\beta = \left(\beta_1, \tilde{\beta}'\right)'$，$\delta = \left(\delta_1, \tilde{\delta}'\right)'$，$x_i = (w_i, \tilde{x}_i')'$，$z_i = (\tilde{z}_i', \tilde{x}_i')'$，同时假设 (u_i, v_i) 为联合正态分布，根据联合正态分布的性质，有：

$$u_i = \rho v_i + e_i$$

其中 $\rho = \frac{\mathbb{C}\mathrm{ov}(u_i, v_i)}{\mathbb{V}(v_i)}$，且 v_i, e_i 相互独立。带入得到：

$$\begin{aligned} y_i &= x_i'\beta + \rho\left(w_i - z_i'\delta\right) + e_i \\ w_i &= z_i'\delta + v_i \end{aligned}$$

从而联合密度函数为：

$$\begin{aligned} f\left(y_i, w_i | z_i, \beta, \delta\right) &= f\left(y_i | z_i, w_i, \beta, \delta\right) \cdot f\left(w_i | z_i, \beta, \delta\right) \\ &= \frac{1}{2\pi\sigma_e\sigma_v} \exp\left\{ -\frac{\left[y_i - x_i'\beta - \rho\left(w_i - z_i'\delta\right)\right]^2}{2\sigma_e^2} - \frac{\left(w_i - z_i'\delta\right)^2}{2\sigma_v^2} \right\} \end{aligned}$$

令 $\theta_{1i} = x_i'\beta, \theta_{2i} = z_i'\delta$，从而似然函数为：

$$l_i\left(\beta, \delta, \rho\right) = C - \ln\sigma_e - \ln\sigma_v - \frac{\left[y_i - \theta_{1i} - \rho\left(w_i - \theta_{2i}\right)\right]^2}{2\sigma_e^2} - \frac{\left(w_i - \theta_{2i}\right)^2}{2\sigma_v^2}$$

其导函数为：

$$\begin{cases} \frac{\partial l_i}{\partial \theta_{1i}} = \frac{1}{\sigma_e^2}\left[y_i - \theta_{1i} - \rho\left(w_i - \theta_{2i}\right)\right] \\ \frac{\partial l_i}{\partial \theta_{2i}} = -\frac{\rho}{\sigma_e^2}\left[y_i - \theta_{1i} - \rho\left(w_i - \theta_{2i}\right)\right] + \frac{1}{\sigma_v^2}\left(w_i - \theta_{2i}\right) \\ \frac{\partial l_i}{\partial \rho} = \frac{1}{\sigma_e^2}\left[y_i - \theta_{1i} - \rho\left(w_i - \theta_{2i}\right)\right]\left(w_i - \theta_{2i}\right) \end{cases}$$

此外，实际上 σ_e, σ_v 也是未知参数，但是给定 (β, δ, ρ)，对 σ_e, σ_v 求一阶条件，有：

$$\begin{aligned} \sigma_e^2 &= \frac{1}{N}\sum_{i=1}^{N}\left[y_i - \theta_{1i} - \rho\left(w_i - \theta_{2i}\right)\right]^2 \\ \sigma_v^2 &= \frac{1}{N}\sum_{i=1}^{N}\left(w_i - \theta_{2i}\right)^2 \end{aligned}$$

因而我们可以使用残差直接计算得到 σ_e, σ_v。我们使用以下程序估计以上的极大似然问题：

代码 1.3: Stata 中的极大似然估计

```
clear
set more off

// LIML by hand
cap program drop limlobj
program limlobj
  args todo b lnfj g1 g2 g3
  tempvar theta1 theta2 theta3 resid2 resid1
  tempname sigma1 sigma2
  mleval `theta1'=`b', eq(1)
  mleval `theta2'=`b', eq(2)
  mleval `theta3'=`b', eq(3)
```

```
  quietly{
    gen `resid1'=.
    gen `resid2'=.
    replace `resid2'=$ML_y2 - `theta2'
    replace `resid1'=$ML_y1 - `theta1' - `theta3'*`resid2
        '
    su `resid1'
    scalar `sigma1'=r(sd)
    su `resid2'
    scalar `sigma2'=r(sd)
    replace `lnfj'=-log(`sigma1')-log(`sigma2') /*
      */-`resid1'^2/(`sigma1'^2*2) /*
      */-`resid2'^2/(`sigma2'^2*2)
    if (`todo'==0) exit
    replace `g1'=1/(`sigma1'^2)*`resid1'
    replace `g2'=-1/(`sigma1'^2)*`resid1'*`theta3'/*
      */+1/(`sigma2'^2)*`resid2'
    replace `g3'=1/(`sigma1'^2)*`resid1'*`resid2'
  }
end

cap program drop liml
program liml, eclass
  version 15.0
  // endo: 内生变量, 至多一个
  // instru: 工具变量, control: 控制变量
  // endofunc: 内生变量的函数
  syntax varlist(max=1), endo(varlist max=1) /*
    */ instru(varlist) control(varlist) [endofc(varlist)]
  tempname dep x z b0 b1 initb
  local `dep' "`varlist'"
  local `x' "`endo' `endofc' `control'"
  local `z' "`instru' `control'"
  ml model lf1 limlobj (Structual:``dep'' = ``x'') /*
    */ (First_stage:`endo' = ``z'') /rho
  quietly{ //使用最小二乘估计作为初始值
    reg ``dep'' ``x''
    mat `b0'=e(b)
    reg `endo' ``z''
    mat `b1'=e(b)
```

```
    mat 'initb'=('b0', 'b1', 0)
    ml init 'initb', copy
  }
  ml maximize
end

set obs 1000
// generate data
gen z=rnormal()
gen u=rnormal()
gen x1=z+rnormal()+u
gen x2=z+0.5*rnormal()
gen y=x1+x2+u
// 官方LIML
ivregress liml  y (x1=z) x2 // 最简单情况，与2SLS等价
// 手写LIML
liml y, endo(x1) instru(z) control(x2)
```

在以上程序中，我们设定了三个方程，分别是结构方程、第一阶段方程，以及额外的参数 ρ。为了避免初始值设置不当从而迭代次数过多的问题，我们使用两个方程的最小二乘估计作为初始值，并使用 ml init 命令进行设置。最后，产生一些模拟的数据，测试我们的代码，并与官方的有限信息极大似然估计进行比较。

注意我们在估计的选项中添加了一个可选的「endofc」选项，是由于有限信息极大似然方法可以灵活的处理内生变量 w_i 的函数（如内生变量的平方 w_i^2、内生变量与外生变量的交叉项 $w_i \times x_i$）等的问题。例如，如果数据生成过程为：

$$y_i = \beta_1 w_i + \beta_2 w_i \times x_{2i} + \tilde{x}_i' \tilde{\beta} + u_i$$

那么同样可以用控制函数法：

$$y_i = \beta_1 w_i + \beta_2 w_i \times x_{2i} + \tilde{x}_i' \tilde{\beta} + \rho (w_i - z_i' \delta) + e_i$$

来处理，比如：

```
// 带有交叉项的内生变量问题，x1内生
gen x12=x1*x2
gen z12=z*x2
gen ynew=x1+x2+x12+u
// 第一种办法，使用z*x2作为额外的工具变量
ivregress 2sls ynew (x1 x12=z z12) x2
// 控制函数法（标准误错误，不具有参考意义）
```

```
quietly: reg x1 z x2
predict x1rnew, residual
reg ynew x1 x2 x12 x1rnew
// 手写的有限信息极大似然
liml ynew, endo(x1) instru(z) control(x2) endofc(x12)
```

1.9.4 模拟

模拟（simulation）是统计研究中经常使用的研究手段，常用来试探性探索或者检验统计量在小样本、大样本下的统计特性。其过程是，在每次模拟中，根据一个数据生成过程（data generating process），给定样本量等参数，使用完全（或部分）随机生成的数据，并计算感兴趣的统计量；重复以上模拟的过程若干次（如 1000 次），这样我们就得到了统计量在给定的样本量等参数下的一个分布，从而我们可以使用这个分布观察统计量的一些统计特性。

比如，如果我们希望研究正态分布的均值的分布，我们可以每次生成 N（$=10$，比如）个正态分布的随机数，并计算均值；重复生成 M（$=1000$，比如）次，我们就得到了 M 个均值，这样我们就可以通过观察这 M 个均值的分布情况与我们的理论进行比较。

在 Stata 中，可以使用「simulate」命令完成，其语法如下：

```
simulate exp_list, reps(#): command
```

该命令是一个前缀，其中 exp_list 是表达式的列表，通常是程序的返回值（r-class 或者 e-class），reps(#) 为模拟重复的次数（即前述的 M），而 command 为要进行模拟的 Stata 命令。

比如，下面的例子我们就展示了如何对：

$$\frac{\sqrt{N}\left(\bar{x}-\mu\right)}{s} \overset{a}{\sim} N\left(0,1\right)$$

这个大样本结论进行模拟：

```
cap program drop sample_mean
program sample_mean, rclass
  version 15
  syntax [, mu(real 1) s(real 1) obs(integer 100)]
  drop _all
  set obs `obs'
  tempvar x
  gen `x'=(rchi2(2)-2)*`s'+`mu'
  su `x'
  return scalar mean=r(mean)
```

```
  return scalar sd=r(sd)
  return scalar sd_m=sqrt(`obs')*(r(mean)-`mu')/r(sd)
end

simulate m=r(sd_m), reps(5000): sample_mean, obs(5) s(2)
hist m, normal
```

在以上程序中，首先定义了一个程序「sample_mean」，其中选项 mu 和 s 为两个位置和尺度参数，而 obs 为每次模拟的样本量 N，默认为 $N=100$。在程序中，首先清除了内存中所有数据，将样本量设置为 'obs'，然后产生一些随机的数字（在这里是 $\chi^2(2)$ 分布的位置尺度族），然后使用「return」命令返回了样本均值（mean）、样本标准差（sd）以及标准化之后的统计量（$\sqrt{N}(\bar{x}-\mu)/s$）。

接下来，使用「simulate」命令进行模拟，表达式「m=r(sd_m)」表示要记录程序返回的 r(sd_m) 到一个变量「m」中，「reps(5000)」表示要重复 $M=5000$ 次模拟；冒号后面表示要模拟的程序为「sample_mean」，每次模拟的样本量为 $N=5$。最后，使用「hist」命令画直方图，「normal」选项表示在直方图中附带正态分布的密度函数图像。

我们可以不断的修改选项 obs 中的数值慢慢变大，如此每次模拟的样本量 N 也在慢慢变大。我们会发现，尽管当样本量很小时 $\sqrt{N}(\bar{x}-\mu)/s$ 的分布与正态分布相差很远，但是随着每次模拟的样本量 N 的变大，$\sqrt{N}(\bar{x}-\mu)/s$ 的分布也与正态分布越来越相似了。

或者，在 Stata16 中，随着数据框的加入，使用数据框也可以很方便的完成模拟的任务。此外，使用数据框的技巧，类似安慰剂检验、Bootstrap 等方法也可以类似的进行。比如上例中的模拟如果使用数据框可以写为：

```
clear all
// 参数设定
local obs=5
local reps=5000
local mu=0
local s=2
// 新建数据框
frame create simulation xbar sd sd_m
// 开始循环
forvalues i = 1/`reps'{
    frame change default
    clear
    quietly: set obs `obs'
    gen x = (rchi2(2)-2)*`s'+`mu'
    quietly: su x
    frame post simulation /*
```

```
    */(r(mean)) (r(sd)) (sqrt('obs')*(r(mean)-'mu')/r(sd))
}
frame simulation: hist sd_m, normal
```

上面的代码中，数据框「simulation」有三个变量，分别记录了每次模拟的均值、标准差和标准化之后的均值，接下来使用循环命令，每次产生随机数据，并进行描述性统计，将统计结果使用「frame post」命令添加到 simulation 数据框中。

1.9.5 ado 文件

1.10 外部编程接口

1.10.1 mata 语言

1.10.2 Python 接口

Python 由于其简单易用但又功能齐全的特点，在数据科学方面异军突起，成为了数据科学方面的重要语言，多数机器学习的算法，包括决策树、随机森林、支持向量机、神经网络等等，在 Python 中都有非常成熟易用的实现，而这些在 Stata 中之前难以做到。而在 Stata16 中加入了对 Python 调用的支持，从而以上机器学习算法等可以方便地通过调用 Python 实现。

为了使用 Python 接口需要首先安装 Python，Python 在各个平台的安装方法可以参考：https://github.com/sijichun/PythonTutor。安装完成后，打开 Stata，可以尝试在命令窗口中输入「python query」命令，将会显示 Stata 默认的 Python 配置，包括 Python 的安装路径及版本等。

此时输入「python」命令会进入 Python 的交互模式，结果窗口中会出现 Python 提示符:「>>>」，如果报错且 Python 已经安装成功，可能是因为 Stata 没有识别 Python 程序的路径。如果需要改变 Python 的执行路径，可以使用 set 命令，比如如下命令修改了 Python 的执行路径：

```
set python_exec /opt/intel/intelpython3/bin/python3, perm
```

其中「perm」选项代表永久修改。此外，还可以使用「set python_userpath」命令设定额外的 Python 模块搜索路径。

使用 python 命令进入 Python 的交互模式时，可以选择在 python 命令后面加一个冒号「:」，加或者不加冒号的区别是：如果不加冒号，那么在交互模式中如果遇到错误不退出；如果加冒号，那么只要遇到返回错误的 Python 代码就退出。如果希望退出交互模式，只要输入「end」命令就可以了。

此外，还可以使用「python:」前缀的形式执行单行的 Python 代码，比如在命令窗口中输入「python: import this」就会打印 Python 之禅（The Zen of Python）。

更常见的是在 do 文件中嵌入 Python 代码。在 do 文件中，同样以「python」或者「python:」开头，以「end」结尾的都是 Python 代码块，比如一个简单的使用 Python 进行大整数计算的 do 文件：

```
local a=4
local b=1000
di `a'^`b'
python
def pow(a,b):
    print(a**b)
end
python: pow(`a',`b')
```

在以上代码中，首先在 python 到 end 的代码块中定义了一个 Python 函数「pow」，接着使用 python 前缀的方式调用了之前定义的「pow」函数。在执行过程中，所有的宏都会被首先替代，然后交由 Python 进行处理，因而上述程序中最后一样首先被翻译为「python: pow(4,1000)」再被 Python 所执行。可以看到由于数字太大，Stata 已经无法计算，但是 Python 给出了正确的结果。实际上以上程序如果写为：

```
local a=4
local b=1000
di `a'^`b'
python
def pow():
    print(`a'*`b')
end
python: pow()
```

也是完全可以的，程序被执行时，「print(`a'*`b')」语句也会被先翻译为「print(4**1000)」，从而实际上以上代码定义了一个输出结果不变的（useless）函数（useless）。

最后，如果需要执行外部的 Python 文件，可以使用「python script」命令，比如我们创建一个 Python 的脚本文件「print_add.py」如下：

```
# filename: print_add.py
import sys
a,b=(sys.argv[1],sys.argv[2])
print(int(a)+int(b))
```

注意在以上的 Python 脚本中，接受两个参数，使用「sys」模块读取两个参数，并将其保存在变量「a,b」中。在 Stata 中可以使用如下命令执行这个脚本：

```
python script print_add.py, args(2 3)
```

即使用了 args 选项来指定 Python 脚本「print_add.py」所需的两个输入参数。

虽然通过脚本参数的方式或者宏的方式可以让 Stata 和 Python 之间交换信息，但是以上方式仅仅只能交换少量的数据，不适合大量数据在 Stata 和 Python 之间进行传递。为此，我们必须借助 sfi 模块在 Stata 和 Python 中交换各种类型的数据，以及使用 Python 操作 Stata 各类数据的方法等，文档可以参考：https://www.stata.com/python/api16/。在这里，我们简单介绍在 Stata 和 Python 之间如何交换宏和数据，其他细节如数据框、日期时间、矩阵、标签等读者可以自行参考文档。

一般而言，需要从 Stata 中获取数据，使用每个模块中的 get* 函数即可。比如 Macro 模块中的 getLocal 可以获得 Stata 中的 local，而 setLocal 函数可以将内容写入到 Stata 中的 local 中，比如：

```
local a=4
local b=1000
python
from sfi import Macro
a=int(Macro.getLocal("a"))
b=int(Macro.getLocal("b"))
Macro.setLocal("c",str(a**b))
end
di "`c'"
```

在以上程序中，首先在 Stata 中设定了两个局部宏 a 和 b，接下来在 Python 代码中，从 sfi 模块中调入 Macro 模块，接着使用 Macro.getLocal() 函数从 Stata 中获取名称为 a 的局部宏，并将其转换为整数，赋值给变量 a，b 同理；接下来计算 a^b 的值，将其转换为字符串，并使用 Macro.setLocal() 函数将其赋值给 c，此时 Stata 环境中就多了一个名为 c 的局部宏，但是由于现在 c 中字符串不能转换为数字（因为太大了），因而需要在 di 时在「`c'」的前后加双引号，即将 c 中的内容当成字符串现实出来。

类似的，可以使用 Data 模块中的 get 函数获取变量的值，但是注意，由于在 Stata 中缺失值「.」代表的是一个比任何数字都大的值，因而在不同机器上可能会有不同的取值，此时需要使用 Missing 模块来处理，比如 Missing 模块的 isMissing() 函数可以用来判断是否为缺失值。以下程序展示了如何在 Stata 中调用 Python 计算调和平均数：

```
use datasets/cfps_family_econ.dta
python
from sfi import Data,Macro
from sfi import Missing
data=Data.get("fincome1")
## 去掉缺失值和<=0的值
```

```
sub_data=[d for d in data\
    if not Missing.isMissing(d) and d>0]

inverse_data=[1/d for d in sub_data]
harmonic_mean=len(sub_data)/sum(inverse_data)
Macro.setLocal("harmonic_mean",str(harmonic_mean))
end
di `harmonic_mean'
```

以上代码中我们分别从 sfi 模块中调入了用于处理数据的 Data 模块、用于处理宏的 Macro 模块以及用于处理缺失值的 Missing 模块；接着，使用 Data.get("fincome1") 获取了 Stata 数据集中变量「fincome1」的所有数据，接下来使用一个列表推断去掉了所有的缺失值以及等于 0 的值；最后计算调和平均，并将其返回到 'harmonic_mean' 这个局部宏中。

类似的，也可以使用 Data 模块中的 store() 函数将一系列数据储存到 Stata 中，以下代码实现了一个简单的计算收入对数的功能，注意对于收入 ≤ 0 或者收入本来为缺失值的观测，应该返回收入对数为缺失值，我们使用 Missing.getValue() 函数获取 Stata 中缺失值在 Stata 中的对应数字，如下：

```
use datasets/cfps_family_econ.dta
python
from sfi import Data
from sfi import Missing
import math
data=Data.get("fincome1")
log_data=[]
for d in data:
    if Missing.isMissing(d) or d<=0:
        log_data.append(Missing.getValue())
    else:
        log_data.append(math.log(d))

Data.addVarDouble("log_income")
Data.store("log_income",None,log_data)
end
```

以上代码中，我们首先从 Stata 中获取 fincome1 变量的数据，接着判断收入是否为缺失值或者 ≤ 0，如果是则返回缺失值存入列表，否则就计算对数放入列表；接下来，通过 Data.addVarDouble() 函数在 Stata 数据集中添加了一个双精度型变量 log_income，最终使用 Data.store() 函数将计算好的列表 log_data 存入到 Stata 的变量 log_income 中。注意 Data.store() 函数的第二个分量为设

定添加数据到第几行，None 代表所有观测，如果只需要储存一行那么就写行号即可，如果要储存多行，则使用一个列表带入即可。

1.10.3 C 接口

练习题

练习 1.1. 请将数据集「bond_interest.xlsx」转换为 Stata 的长格式，即包含如下几个变量：日期、国债名称、国债代码、股价收益率（tips: 需灵活使用 Excel 的功能）。

练习 1.2. 请使用附带的数据「export.dta」计算显示性比较优势指数（Revealed comparative advantage），即：

$$RCA_{cp} = \frac{\frac{E_{cp}}{\sum_{p' \in P} E_{cp'}}}{\frac{\sum_{c' \in C} E_{c'p}}{\sum_{p' \in P} \sum_{c' \in C} E_{c'p'}}}$$

其中 $c \in C$ 为国家，C 为所有国家的集合；$p \in P$ 为产品，P 为产品的集合，在数据中以 HS2 位代码表示；E_{cp} 为国家 c 出口产品 p 的出口额。

练习 1.3. 使用数据集 stock_price.dta，计算所有股票的收益率，并找出所有样本偏度系数的符号与（均值-中位数）的符号不相同的股票代码。

练习 1.4. 请使用附带的数据「citydata.dta」，以描述性统计量和图表的形式，选取不同视角，比较我国东部、中部、西部的人均 GDP、GDP 增长率、人均医院个数以及人均医生数等数据。

练习 1.5. 请使用附带的数据「citydata.dta」，将数据整理为每个城市的 GDP 的时间序列数据，即每一行代表一个年份，每一列代表一个城市，并尝试使用上海市的 GDP 对浙江省、江苏省、安徽省的 GDP 做回归。

练习 1.6. 请使用附带的数据「cfps_adult.dta」，为每一个个人随机匹配一个同一省份的其他个人（不能自己匹配自己），并计算随机匹配的两个人的收入之差（绝对值）的描述性统计。

练习 1.7. 重复练习1.6中的步骤 100 次，记录每次两人收入之差的绝对值的均值，并画出均值的直方图。

练习 1.8. 尝试随机生成一组数据计算中位数，并对中位数进行模拟，观察中位数的抽样分布。如果将换成 75% 分位数，观察两者分布上的差别。