Vala 编程手册

译者注：

此手册为那些只会用 C，在不想学习其他高级语言的前提下使用面向对象编程并享受大量现代编程技术，而加快开发效率的白痴（包括本人）所译！

为了更好的学习 Vala，译者在 github 上创建了专门的仓库，并结合学习《大话设计模式》过程中的感悟，会不断地往仓库上添加练习代码，希望和大家一起交流。

仓库地址：https://github.com/Matrix-Zhang/Vala-Practice.git

原文地址：https://live.gnome.org/Vala/Tutorial

内容目录

1. 介绍（ Introduction ）.....5

免责声明：.....5

Vala 是什么？（ What is Vala ? ）.....5

本手册的阅读对象（ Who is this tutorial for ? ）.....6

排版约定 （Conventions）.....6

2. 第一个 Vala 应用程序（ A First Program ）.....6

编译运行程序 （ Compile and Run ）.....8

3. Vala 基础（ Basics ）.....8

源代码和编译（ Sources Files and Compilation ）.....8

语法概述（ Syntax Overview ）.....9

注释 (Comments).....10
数据类型 (Data Types).....10
值类型 (Value Types).....10
字符串 (Strings).....11
数组 (Arrays).....13
引用类型 (Reference Types).....15
静态类型转换 (Static Type Casting).....15
类型推断 (Type Inference).....15
继承某种类型定义一个新的类型 (Defining new Type from other).....16
操作符 (Operators).....16
控制结构 (Control Structures).....18
语言组件 (Language Elements).....20
方法 (Methods).....20
委托 (Delegates) 22
匿名方法 / 闭包 (Anonymous Methods / Closures).....23
命名空间 (NameSpaces).....24
结构体 (structs).....25
类 (Classes).....26
接口 (Interfaces).....26
代码属性 (Code Attributes).....27
4. 面向对象编程 (Object Oriented Programming) 27
基础 (Basics) 27
构造 (Construction) 29

析构（Destruction）.....30
信号（Signals）.....31
属性（Properties）.....32
继承（Inheritance）.....35
抽象类（Abstract Classes）.....37
接口 / 混合（Interfaces / Mixins）.....37
多态（Polymorphism）.....39
方法隐藏（Method Hiding）.....42
运行时类型信息（Run-Time Type Information）.....42
动态类型转换（Dynamic Type Casting）.....43
泛型（Generics）.....43
GObject 风格的构造（GObject-Style Construction）.....45
5. 高级特性（Advanced Features）.....46
断言和简化编程（Assertions and Contract Programming）.....46
错误处理（Error Handling）.....47
参数路径（Parameter Directions）.....50
容器（Collections）.....52
ArrayList<G>.....53
HashMap<K,V>.....53
HashSet<G>.....54
Read-Only Views.....54
语法支撑的方法（Methods With Syntax Support）.....54
多线程编程（Multi-Threading）.....56
Vala 中的线程（Threads in Vala）.....56
资源控制（Resource Control）.....58

主循环 (The Main Loop) 59
异步方法 (Asynchronous Methods) 60
弱引用 (Weak References) 64
所有权 (Ownership) 64
无属引用 (Unowned References) 64
所有权转让 (Ownership Transfer) 67
可变长度的参数列表 (Variable-Length Argument Lists) 67
指针 (Pointers) 69
非对象类 (Non-Object classes) 70
D-BUS 集成 (D-Bus Integration) 70
配置 (Profiles) 73
6. 实验性的功能 (Experimental Features) 73
连锁关系表达式 (Chained Relational Expressions) 74
正则表达式 (Regular Expression Literals) 74
严格的非空模式 (Strict Non-Null Mode) 76
7. 库 (Libraries) 76
库的使用 (Using Libraries) 77
库的制作 (Creating a Library) 77
使用 Autotools (Using Autotools) 77
使用命令行来汇编和链接 (Compilation and linking using Command Line) 81
例子 (Example) 82
使用 VAPI 文件构建库 (Binding Libraries with VAPI Files) 83
8. 工具 (Tools) 84

valac.....84
Vapigen.....84
Vala-gen-introspect.....84
9. 其他技术（Techniques）.....85
调试（Debugging）.....85
使用 GLib（Using Glib）.....86
文件处理（File Handling）.....86

介绍（Introduction）

免责声明：

Vala 是一个仍然在不断发展，不断演进的项目，所以很多特性随时都可能发生改变。我会尽力保持本手册内容能跟上 Vala 的演进步伐，但我并不能保证此点。另外，我也不能保证我在本手册的示范代码里所使用的技术在原则上是最好的，但是我仍然会努力的去达到这个标准。

Vala 是什么？（What is Vala ?）

Vala 是一门新兴的编程语言，为那些依赖 GNOME 平台运行时（runtime），尤其是依赖 GLib 和 GObject 库的应用程序的编写提供了大量现代的编程技术的支持。目前，GNOME 平台长期持支了一套完整的编程环境，拥有诸如动态类型系统，辅助内存管理等特性。在 Vala 诞生之前，要在 GNOME 平台上编程，只能去调用原生的 C 语言接口，但是这样的编程方法暴露了很多根本不需要知道的细节，如果用更高级的语言，诸如 Python，Mono C#，则需要相应的虚拟机环境，否则就只能调用 C++ 封装的库。

Vala 不同于以上提到的任何语言，Vala 最终会转换成 C 语言，然后再编译运行，不用依赖 GNOME 平台的其他附加的库（除了 GLib 和 GObject 之外）。

由此，造成了一些细微的但又非常重要的影响：

- 使用 Vala 编写应用程序和直接使用 C 编写应用程序，运行的效率非常相似，但是 Vala 相比 C，更加容易,快速地编写和维护。

- 使用 Vala 做不到 C 语言也做不到的事情，尽管 Vala 相对 C 语言来说，提供了很多 C 语言没有的特性，但实际上 Vala 所有的特性都是映射到 C 语言本身的，只不过如果直接用 C 语言来实现这些特性的话，非常耗时，也非常困难。

综上所述，即便 Vala 提供了全部我们所希望的现代编程特性，但其实这些都是得益于一个已经存在的强力平台，因此我们在使用 Vala 的时候，必须遵守这个平台的编程规范。

本手册的阅读对象（ Who is this tutorial for ? ）

本手册不会深入的去讲解基本的编程方法。只是大概的阐明一些面向对象的编程原则，我将会把重点放在 Vala 是如何适用这些编程原则的。如果你对其他语言有所了解，将对理解本手册的内容很有帮助。即便如此，并不需求一定要对某门语言有深入的了解。

Vala 借鉴了很多 C# 的语法，但是我不会去解释 Vala 和 C#，或者 JAVA 之间的相似或者不同，以使得本手册更加的容易理解。

理解 C 语言对学习本手册是非常有帮助的，有时你都不需要理解 Vala 本身。要知道，Vala 最终是以 C 语言的形式编译并运行，并且绝大多数是要调用 C 的库，掌握 C 语言的知识，将有助于更加容易，更加深入的理解和学习 Vala。

排版约定 （Conventions）

示例代码将使用 Ubuntu Mono 字体，命令行则使用 $ 符号开头。

我倾向于使用最简洁明了的代码去阐述他所实现的功能，我在某些地方会说明哪些代码是可以省略的，但是这并不代表我鼓励你这样去做。

第一个 Vala 应用程序 (A First Program)

没错，你猜对了，Vala 版的 Hello Word！

```
class Demo.HelloWorld : GLib.Object {
    public static int main(string[] args) {
        stdout.printf("Hello, World\n");
```

```
            return 0;
        }
    }
```

我希望你能意识到，这段代码的实现其实并不是很好，不过我会带着你一步一步的去深入分析它！

```
    class Demo.HelloWorld : GLib.Object {
```

这一行，是定义一个类的开始，可以看到 Vala 中的类定义和其他语言非常相似。一个类代表了一个对象的基本类型，基于该类创建的对象拥有相同的属性。在 Vala 中，一个类相关的维护是由 GObject 库来完成的，但是在平常运用中，我们无需关心该底层细节。

特别要注意的是，上面的 Demo.HelloWord 类是继承于 GLib.Object 的子类，这是因为即便 Vala 允许定义其他类型的类，但是绝大多数情况下，作为 GLib.Object 的子类是你所希望的，而且实际上，你只有作为 GLb.Object 的子类，才能在 Vala 编程中使用某些特性!

至于这行代码的其余部分，则展现了命名空间和一个完全展开的合法命名，这些将在后面说明。

```
    public static int main(string[] args) {
```

这行则是一个方法的定义，所谓方法，是其所属的某类对象能够调用的一个函数。而用 static 修饰的方法，则表明该方法可以被直接调用，而无需指定一个特定的实例。这里定义的方法命名为 main，并且被 static 修饰，Vala 则将其视为程序的入口点。（等同于 C 的 main 函数）

main 方法并不一定要定义在一个类中，事实上如果非要定义在一个类中的话，则必须要加上 static 修饰符，至于到底定义为 public 还是 private 型是无关紧要的。至于返回值，可以是 void 或者 int 型，

如果方法定义为 void 型，则程序的退出码始终为 0，参数中的 string 字符串数组，则是可选的。

```
    stdout.printf("Hello, World\n");
```

其中 stdout 是 GLib 命名空间中的对象，Vala 会确保开发者始终能够访问该命名空间的对象而无需显式的指明 GLib 命名空间。这行代码告诉 Vala 调用 stdout 对象的 printf 方法，参数则是字符串“Hello, World\n”。这是种很常见的语法，用来调用一个对象的方法，或者访问一个对象的数据，字符串参数中的\n 则是换行符的转义字符。

```
    return 0;
```

return 关键字的作用是，返回一个值给调用者并且结束 main 方法的执行，此时也同时结束了整个程序的运行，main 方法的返回值同样也作为整个程序的返回码。

至于剩下的两行 '}'，则简单结束了 main 方法的定义和整个类的定义。

编译运行程序 (Compile and Run)

如果你已经安装了 Vala，那么编译运行该程序只需要在命令行中输入如下命令：

```
$ valac hello.vala
$ ./hello
```

valac 是 Vala 语言的编译器，它负责将代码编译为二进制文件，此时最终编译出来的二进制文件的文件名和源代码名一致，我们可以直接在机器上运行该程序，当然，运行结果可想而知。。。

Vala 基础 (Basics)

源代码和编译 (Sources Files and Compilation)

Vala 的源代码是以.vala 扩展名保存的，Vala 并没有采取 JAVA 那样的强制性组织架构，也没有包和专门的类文件的概念。相反，Vala 代码的组织架构都是在每个代码文件里以文本描述，通过使用 namespace 这样的结构来描述代码的逻辑位置。当需要编译 Vala 代码的时候，将需要编译的代码文件列表传给编译器，编译器知道如何将他们组合在一起。

结果就是，只要你想，你可以将很多的类或者方法全部之放在一个文件当中，甚至是将许多不同的命名空间结合在一起。这不一定是个好主意。你或许想遵守一个普遍的约定。一个很好的关于如何组织架构 Vala 程序代码的例子就是 Vala 项目本身！

同一个包的所有源代码文件通过命令行参数的形式传送给 Vala 的编译器 valac。这一点和 Java 代码的编译有点类似。

如下：

```
$ valac compiler.vala --pkg libvala
```

以上命令将链接包 libvala，并编译生成名为 compiler 的一个二进制文件，事实上，valac 本身就是这样编译出来的。

如果你希望自己定义一个不同的二进制文件名或者你传入了多种源代码给编译器，可以使用 -o 选项来指定生成的二进制文件名：

```
$ valac source1.vala source2.vala -o myprogram
$ ./myprogram
```

如果你使用了 -c 选项，valac 编译器不会将你的程序编译为二进制文件，而是为每个 Vala 源代码生成中间 C 文件，在当前情形下，将生成 source1.c 和 source2.c 两个文件。如果你查看这些 C 代码文件内容，你会发现，在 Vala 中使用类来编程，其实是用 C 来实现的，但是更加简洁。你也将注意到，使用的类都是在系统中动态注册的。这是 GNOME 平台之所以强大的典型例子，但是就像我之前所说，使用 Vala 是 不需要更多的了解这些细节的。

如果你希望生成 C 风格的.h 头文件，可以使用 -H 选项：

```
$ valac hello.vala -C -H hello.h
```

语法概述（ Syntax Overview ）

Vala 的语法很大程度上借鉴了 C# 的语法。因此，许多语法对于熟悉 C-like 语言的程序员来说是非常有亲和力的，鉴于此，我将简要的介绍 Vala 的语法。

使用大括号 {} 来定义范围。一个对象或者引用只有在 { 和 } 之间的才有效。此符号也是类，方法，代码段等的分隔符，因此他们都自动拥有自己的范围。Vala 并没有严格限制变量要在哪里声明。

使用类型和名称来定义一个标识符，比如 int c，意思是定义一个名为 c 的整型。此时，也意味着创建个 int 型的值类型对象。而对于引用类型，则只是创建一个新的引用但并不初始化以指向任何一个实例。

标识符名称的定义规则和 C 的一样，必须是以字母 [a-z],[A-Z] 或者下划线其中之一开头，随后的名称可以附上数字 [0-9]。但是 UNICODE 字符是不允许作为标识符名称的。也可以使用保留字符 @ 来作为一个标识符的前缀，使得可以用关键字来命名标识符，这个字符不是名称的一部分。比如你可以这样定义个名为 foreach 的方法，@foreach，@ 是 Vala 的一个保留字符。

引用类型的实例化是使用 new 操作符和对应的构造方法名称来实现的，绝大多数情况下，构造方法的名称是和指定的类型名称是一致的，比如 Object o = new Object()，创建了一个 Object 类型对象，并用 o 来引用这个对象。

注释（ Comments ）

Vala 使用如下不同方法来注释代码：

```
// Comment continues until end of line

/* Comment lasts between delimiters */

/**
* Documentation comment
*/
```

以上风格和其他很多语言一样，所以没有必要过多解释。其中 Documentation comment 并不是特别用于 Vala 的，但是类似于 Valadoc 这样的文档生成工具，将会识别这种风格的注释。

数据类型（ Data Types ）

总的来说，Vala 有两种数据类型，分别是：引用类型和值类型，这两种类型名称分别代表着各自的实例是如何在系统中传递的。值类型在赋值给一个新的标识符的时候是自动拷贝自身值的，而引用类型则不拷贝，相反只是简单的将对象实例增加一个新的引用而已。

在一个类型前加上 const 修饰符，代表了其值是只读的，命名规则一般是全大写。

值类型（ Value Types ）

Vala 和其他语言一样，提供了一组简单的值类型：

- char, uchar; 字节，因为历史原因，也称作 char
- unichar; 32 位的 Unicode 字符
- int, uint; 整型
- long, ulong; 长整型
- short, ushort; 短整型
- int8, int16, int32, int64, uint8 uint16, uint32, uint64; 固定长度的整形，其中的数字分别代表各类型所占的位的长度。
- float, double 浮点数

- `bool` 布尔型，可能的值为 `true` 或者 `false`
- `struct` 结构体
- `enum` 枚举型，内部表述为整型，但并不像 Java 中那样是个类

以下是个例子：

```
/* atomic types */
unichar c = 'u';
float percentile = 0.75f;
const double MU_BOHR = 927.400915E-26;
bool the_box_has_crashed = false;

/* defining a struct */
struct Vector {
    public double x;
    public double y;
    public double z;
}

/* defining an enum */
enum WindowType {
    TOPLEVEL,
    POPUP
}
```

在不同平台上，这些类型的长度都是不一样的，除了固定长度整型是个例外。使用 `sizeof` 操作可以获得当前平台下相应类型的长度（以字节为单位）：

```
ulong nbytes = sizeof(int32);   //nbytes will be 4 (= 32 bits)
```

你可以对数值类型使用 `.MIN` 和 `.MAX` 方法来确定当前数值类型所能表示的最小值和最大值，比如使用 `int.MIN` 和 `int.MAX` 来获得当前 `int` 型所能表示的数值范围。

字符串（ Strings ）

使用 `string` 来定义字符串类型，Vala 的字符串都是以 UTF-8 编码的，而且是不可更改的。

```
string text = “A string literal”;
```

Vala 支持一种名为逐字字符串的字符串，使用这种字符串可以避免字符串中的字符被转义，比

如 ‘\n’字符。 若要使用这种字符串，则用三个双引号 ” 将字符串包围即可。

```
string verbatim = """This is a so-called "verbatim string".
                     Verbatim strings don't process escape sequences,
                     such as \n, \t, \\, etc.They may contain quotes and
                     may span multiple lines.""";
```

Vala 还支持用 @ 符号为前缀的字符串模版，这种字符串可以将内部那些以 $ 开头的变量或者表达式展开，然后再得到最终的字符串：

```
int a = 6, b = 7;
string s = @”$a * $b = $( a * b )”; // => “6 * 7 = 42”
```

符号 == 和 符号 != 可以用于比较两个字符串的内容，而 JAVA 则是测试两个字符串的引用是否相等。

也可以对字符串使用 [start : end] 来分割字符串，如果使用负值，则是针对字符串尾的偏移。

```
string greeting = “hello, world”;
string s1 = greeting[7 : 12]; // => “word”
string s2 = greeting[-4 : -2]; //=> “or”
```

和其他很多语言一样，Vala 的字符串索引也是从 0 开始，从版本 0.11 往后，可以使用[index]直接访问字符串的某个字节。

```
uint8 b = greeting[7]; // => 0x77
```

但是，却不能使用这个方法来给字符串中某个字节赋新的值，因为 Vala 的字符串是不能改动的！

许多基本类型都有巧妙的方法将其转换成字符串类型或者反之将字符串转换成其类型，

比如：

```
bool b = bool.parse(“false”);               // => false
int i = int.parse(“-52”);                   // => 52
double d = double.parse(“6.67428E-11”); // => 6.67428E-11
string s1 = true.to_string();               // => “true”
string s2 = 21.to_string();             // => “21”
```

有两个很有用的方法，使得很容易的从终端写入或者读取字符串，分别是 stdout.printf()和 stdin.readline():

```
stdout.printf("Hello, world\n");
stdout.printf("%d %g %s\n", 42, 3.1415, "Vala");
string input = stdin.read_line();
int number = int.parse(stdin.read_line());
```

从之前第一个 Hello World 的例子中，我们已经知道 stdout.printf() 的作用了。实际上，和 C 语言一样（C format strings），这个方法可以使用多个任意类型的参数，在这种情况下，第一个参数是格式化字符串。如果处于某种必须输出错误信息的情形下，可以使用 stderr.printf() 来代替 stdout.printf()。

此外还可以使用 in 操作符来判断一个字符串是否被另外一个字符串所包含：

```
if (“ere” in “Able was I ere I saw Elba.”)...
```

想了解字符串相关的完整信息，请参阅 the complete overview of the string class，另外还有个关于如何使用字符串的示例程序 sample program 。

数组（ Arrays ）

定义数组首先指定一个类型，然后紧跟着符号 []，最后用 new 操作符来创建。比如 int[] a = new int[10]，创建了一个长度为 10 的整型数组。数组的长度包含在数组的 length 成员中，比如 int count = a.length。注意，如果你写这样一行代码： Object[] a = new Object[10]; 这行代码没有创建任何对象，仅仅创建了一个用来存储 10 个这样对象的数组而已。

```
int[] a = new int[10];
int[] b = { 2, 4, 6, 8 };
```

和字符串类似，你也可以使用[start : end]来分割数组：

```
int[] c = b[1 : 3]; // => { 4, 6 }
```

针对分割后的数组所做的修改，是不影响原始数组的。

定义多维数组，可以这样定义：[,] 或者 [,,]等。

```
int[,] c = new int[3, 4];
int[,] d = {{ 2, 4, 6, 8},
            { 3, 5, 7, 9},
            { 1, 3, 5, 7}};
```

```
d[2,3] = 42;
```

获取多维数组各维的大小可以访问各维的 length 成员:

```
int[,] arr = new int[4, 5];
int r = arr.length[0];
int c = arr.length[1];
```

但是要注意，你不能从多维中获得一个单维数组，也不要去尝试分割一个多维数组,如下：

```
int[,] arr = {{ 1, 2 }, { 3, 4 }};
int[] b = arr[0];           // won't work
int[] c = arr[0,];          // won't work
int[] d = arr[:, 0];        // won't wrok
int[] e = arr[0:1, 0];      // won't work
int[,] f = arr[0:1, 0:1];   // won't work
```

可以使用 += 操作符来动态的追加数组元素，但是，这种方式只对局部数组变量或者私有数组有效。数组会根据需要来自动重新分配空间，出于运行效率原因，重分配的大小是按照的 2 的指数来进行的，.length 保存的是数组中实际的元素个数，而非内部空间大小。

```
int[] e = {};
e += 12;
e += 5;
e += 37;
```

也可以调用 resize() 方法来来重新改变数组大小并尽可能不改变原始数组内容。

```
int[] a = new int[5];
a.resize(12);
```

如果在定义数组的时候仅仅用 [] 符号，而没使用 new 操作符，将得到一个固定大小的数组。固定大小的数组是在栈上分配空间的（如果是局部数组变量的话），或者是基于内联分配的（如果是作为某个域的话），因此，不能对这样的数组重新分配大小。

```
int f[10];      // no 'new ...'
```

程序运行时对数组的访问，Vala 是不做边界检查的，因此为了更加安全，可以使用更复杂的数据结构，比如 ArrayList，在稍后的容器（ collections ） 章节将学习更多这方面的内容。

引用类型（ Reference Types ）

引用是指所有被声明为类（class）的类型，不论该类是否继承自 GLib 的 Object 基本类，当以引用类型来传递一个类（class）的时候，Vala 会持续跟踪对象的引用计数的增减来自动管理内存。当一个引用类型没有指向任何实例的时候，它的值为 null,更多关于类及其特性的叙述将在后面的面向对象编程的章节介绍。

```
/* defining a class */
class Track : GLib.Object {             /* subclassing 'GLib.Object' */

    public double mass;                 /* a public field */

    public double name { get; set; }    /* a public property */

    private bool terminated = false;    /* a private field */

    public void terminate() {           /* a public method */

        terminated = true;
    }
}
```

静态类型转换（ Static Type Casting ）

在 Vala 中，可以转换一个变量的类型。要使用静态类型转换，只要在变量前用括号包住想要转换的类型即可。使用静态类型转换并不会采取任何运行时的安全检查，并且此方法适用于任何 Vala 类型，比如：

```
int i = 10;
float j = (float) i;
```

Vala 也支持一种称作动态类型转换的机制，这种机制是利用运行时的类型检查来实现，将在面向对象编程的章节描述。

类型推断（ Type Inference ）

Vala 拥有一种称作类型推断的机制，具体表现为定义一个变量时只需要用 var 关键字声明一个模糊不清的类型，而不需要指定一个具体的类型，具体的类型则取决于赋值表达式右边。这种机制有助于在不牺牲静态类型功能的前提下减少不必要的冗余代码。

```
var p = new Person();       // same as: Person p = new Person();
var s = “hello”;            // same as: string s = “hello”;
var l = new List<int>();    // same as: List<int> l = new List<int>();
var i = 10;                 // same as: int i = 10;
```

不过这种机制只能对局部变量使用，尤其对那些拥有多个参数类型的方法时非常有用。

比较：

```
MyFoo<string, MyBar<string, int>> foo = new MyFoo<string, MyBar<string, int>>();
```

和：

```
var foo = new MyFoo<string, MyBar<string, int>>();
```

显然第二种风格要简洁的多！

继承某种类型定义一个新的类型 (Defining new Type from other)

继承某种类型来定义一个新的类型是个需求，这里有个例子：

```
/*
Define a new type from a container like GLib.List with elements type GLib.Value
*/

public class ValueList : GLib.List<GLib.Value> {

    [CCode (has_construct_function = false)]

    protected ValueList ();

    public static GLib.Type get_type ();
}
```

操作符 (Operators)

=

赋值操作符。操作符左边必须是个标识符，而右边必须是个适当的值或者引用。

+ , - , / , * , %

基本算术操作符，需要给之提供左右操作数。符号 + 同样适用于串联字符串。

+=, -=, /=, *=, %=

算术操作符，用在两个操作数之间，但是左边操作数必须是标识符从而能够被赋予新的运算结果。

++, --

自赠，自减操作符，这种操作符只有一个简单的数据类型操作数，操作数的值改变后，又重新赋给操作数本身，操作符可以放在操作书的前面或者操作数的后面，分别代表先运算赋值再返回值，和先返回值，再运算赋值。

|, ^, &, ~, |=, &=, ^=

位操作符：分别为按位或，按位异或，按位与，按位取反。第二组带等号的操作符和前面的算术操作符类似，这些操作符都可用于简单的数据类型，至于为什么没有 ~= 这样的操作符是因为 ~ 是单元操作符，因此要得到类似的效果则这样用 a = ~a

<<, >>

位移操作符，对操作符左边的数字按位移动操作符右边数值大小的位数

<<=, >>=

位移操作符，操作符左边必须是标识符，对该标志符的值按位移动操作符右边数值大小的位数，并将结果的值赋予操作符左边的标志符

==

相等操作符，测试操作符左右两边的值是否相等，返回结果为 bool 型的值。如果测试的是值类型，则比较他们的值是否相等，如果测试的是引用类型，则比较他们是否引用的同一个实例，但有个例外，

如果比较的是 string 类型，则比较的是 string 变量的值。

<, >, >=, <=, !=

不等操作符，根据符号代表的意思来测试操作符左右两边不等的各种形式，返回结果为 bool 型的值。对于 string 类型的比较，是按照字序来比较的。

!, &&, ||

逻辑操作符： 分别为逻辑非， 逻辑与， 逻辑或。这些操作符是用于 bool 类型的值的，第一个操作数为单个，其他两个则是两个操作数。

? :

三元条件操作符。测试一个条件表达式，返回的结果取决于表达式的真假，结果为真则返回左边的子表达式，否则返回右边的子表达式，比如：`condition ? value if true : value if false`

? ?

空否操作符，表达式 `a ?? b` 等同于 `a != null ? a : b`，这个操作符在引用一个空值的时候，需要提供一个默认值的情况下非常有用：

```
stdout.printf("Hello, %s!\n", name ?? "unknown person");
```

in

此操作符测试右操作数是否包含左操作数，适用于 `arrays, strings, collections` 等其他拥有适合的 `contains()` 方法的类型。用于 `string` 类型的时候，则代表子字符串匹配。

Vala 是不支持操作符重载的，此外还有诸如 `lambda` 表达式等用于特殊用途的操作符，在后面用到的时候再介绍。

控制结构 (Control Structures)

while (a > b) { a--; }

首先判断 `a` 是否大于 `b`， 如果条件为真 `1`，则 `a` 自减，然后继续这一过程， 否则结束该循环体。

do { a--; } while (a > b);

首先 `a` 自减，然后再判断 `a` 是否大于 `b`，如果条件为真，则继续这一过程，否则结束该循环体。

for (int a = 0; a < 10; a++) { stdout.printf(“%d\n”, a); }

首先初始化整型变量 `a` 的值为 `0`，然后判断 `a` 是否小于 `10`， 如果条件为真，则打印 `a` 的值，接着 `a` 自增，然后继续循环判断，打印的过程，否则结束该循环体。

foreach (int a in int_array) { stdout.printf(“%d\n”, a); }

逐个打印 int_array 每个元素的值，也称作迭代遍历。“迭代”的概念将在后面的章节描述。

以上四种循环的执行流程，都可以用关键字 break 和 continue 来控制。break 关键字将导致执行流立即停止，并结束本循环体的执行，而 continue 关键字则会结束本次的循环流程，从头开始下一次的循环流程。

```
if (a > 0) { stdout.printf("a is greater than 0\n"); }
else if (a < 0) { stdout.printf("a is less than 0\n"); }
else { stdout.printf("a is equal to 0\n"); }
```

以上将在一组条件测试中执行一段符合条件的代码段。第一个条件测试意味着，如果 a 大于 0 则运行后面紧跟着的大括号中的代码。如果 a 大于 0，则不会继续测试 a < 0 这个条件，否则继续测试 a 是否小于 0， 如果仍不符合，则最终执行 else 后面的代码段。

```
switch (a) {

    case 1:

        stdout.printf("one\n");
        break;

    case 2:
    case 3:

        stdout.printf("two or three\n");
        break;

    default:

        stdout.printf("unknown\n");
        break;
}
```

switch 分支根据传入的值（此处为 a），可能执行 1 个或者 0 个分支代码段。除了空的 case 分支之外，Vala 是不允许不在 case 分支后加上 break，return，throw 关键字的。另外，直接传入字符串值来选择分支也是可行的。

比如：

```
    switch (str) {
        case “a” :
            break;

        case “b” :
```

```
            break;

        case “c”:
            break;

        default:
            break;
    }
```

有一点是需要 C 程序员们注意的：条件判断必须是以布尔值为准的，举个例子来说的需要判断一个非 0 或者非 null 条件的时候，必须显式调用，比如：if (object != null) { } 或 if (number != 0) { }，而不是 C 程序员惯用的 if (!object) 和 if (!number) 形式。

语言组件（ Language Elements ）

方法（ Methods ）

Vala 语言中将函数称为方法，不论该方法是定义在类中还是在类外，从现在开始，我们将坚持使用“方法”这个术语。

```
int method_name(int arg1, Object arg2) {
    return 1;
}
```

以上代码定义了一个名为 method_name 的方法，该方法有两个参数，第一个是 int 型，传递的是其值的拷贝，第二个是 Object 型，传递的是其引用，该方法返回值为 int 型，在当前代码中，返回的是 1 。

所有 Vala 中的方法，都对应 C 中的函数，因此可以附带任意数量的参数，并且返回一个值，也可以不返回任何值，如果定义为 void 方法的话。也可以将多个返回值放在调用代码所知道的位置，关于此机制将在后面高级部分中的“参数路径”章节描述。

对于方法的命名规则是：采用全小写的字母，使用下划线 _ 来分隔单词。这和 C# 还有 Java 的驼峰规则（CamelCase）或者混合驼峰（mixedCamelCase）规则是有一点点区别的。但是使用这种规则，将可以和其他 Vala 或者 C/GObject 库保持兼容。

Vala 不支持方法重载，即不允许多个方法拥有一致的方法名。

```
void draw(string text) { }
void draw(Shape shape) { }  // not possible
```

正是由于上述的事实，为了让 C 程序员更好的使用那些由 Vala 编写的库，在 Vala 编程中欲达到上述的效果这可以样做：

```
void draw_text(string text) { }
void draw_shape(Shape shape) { }
```

通过定义不同的方法名，可以避免命名冲突。在那些提供方法重载的语言中，往往是为了使用简洁的方法定义来串联基本方法。

```
void f(int x, string s, double z) { }
void f(int x, string s) { f(x, s, 0.5); }  // not possible
void f(int x) { f(x, "hello"); }           // not possible
```

在这种情形下，我们可以使用 Vala 提供的“默认参数”的特性来完成，只需要定义一个方法，并定义参数的默认值，之后不需要显式的指明参数值就可以调用方法。

```
void f(int x, string s = "hello", double z = 0.5) { }
```

调用此方法可能的情形有：

```
f(2);
f(2, "hi");
f(2, "hi", 0.75);
```

甚至可以定义一个真正的可变参数的方法，就像 stdout.printf() 方法那样，但是我不建议使用此特性，在稍后我们将介绍此特性。

Vala 对方法的参数和返回值是否为空执行一个基本的检查，如果一个方法允许参数或者返回值为 null，则类型关键字后要以 ‘?’ 结尾。这个附加的信息，告诉 Vala 的编译器，对调用该方法相关的地方加上必要的静态和运行时的断言检查，以帮助在编程过程中避免可能会有的相关错误，比如去引用一个值为 null 的引用。

```
string? method_name(string? text, Foo? foo, Bar bar) {
    // ...
}
```

在这个例子中，text，foo 参数还有返回值可能是 null，但是 bar 则必须不为 null。

委托（Delegates）

```
delegate void DelegateType(int a);
```

委托（Delegates)类型代表一个方法，使得一段代码可以在对象之间传递。上面的例子中，定义了一个名为 DelegateType 的委托类型，代表了拥有一个 int 型参数，无返回值的方法。那些符合此特征的方法，可以赋值给这种类型的变量，或者作为参数传递给需要此类型参数的方法。

```
delegate void DelegateType(int a);

void f1(int a) {
    stdout.printf("%d\n", a);
}

void f2(DelegateType d, int a) {
    d(a);       // Calling a delegate
}

void main() {
    f2(f1, 5);  // Passing a method as delegate argument to another
                   method
}
```

以上代码在调用方法 f2 时，传入方法 f1 的引用，和数字参数 5，接着方法 f2 内部将调用方法 f1，并且将数字 5 作为参数再传入方法 f1。委托类型也可以局部的创建，然后内部的方法成员，可以赋值给这个局部委托类型：

```
class Foo {
    public void f1(int a) {
        stdout.printf("a = %d\n", a);
    }
    delegate void DelegateType(int a);
    public static int main(string[] args) {
        Foo foo = new Foo();
        DelegateType d1 = foo.f1;
```

```
        d1(10);

        return 0;

    }
}
```

匿名方法 / 闭包（ Anonymous Methods / Closures ）

```
(a) => { stdout.printf("%d\n", a); }
```

匿名方法也被称作 lambda 表达式， 或 function literal（方法显式声明），或闭包， 在 Vala 中可以使用符号 => 来定义。位于该符号左侧的是该方法的参数列表，而方法的实现代码则位于该符号的右侧。

像上面定义的匿名方法从某种角度来看没太大意义。但是在你需要将某个方法直接作为委托类型，或者想给某个以方法为参数的方法传递该参数的时候，将非常有用。

注意，在匿名方法中，不论是参数类型还是返回值类型都没有明确指出。因此只能根据和该匿名方法关联的委托类型来推断。

将匿名方法赋给某个委托类型变量：

```
delegate void PrintIntFunc(int a);

void main() {
    PrintIntFunc p1 = (a) => { stdout.printf("%d\n", a); };
    p1(10);
    // Curly braces are optional if the body contains only one
       statement:
    PrintIntFunc p2 = (a) => stdout.printf("%d\n", a);
    p2(20):
}
```

将匿名方法作为参数传递给另外一个方法：

```
delegate int Comparator(int a, int b);

void my_sorting_algorithm(int[] data, Comparator compare) {
```

```
    // ... 'compare' is called somewhere in here ...
}

void main() {
    int[] data = { 3, 9, 2, 7, 5 };
    // An anonymous method is passed as the second argument:
    my_sorting_algorithm(data, (a, b) => {
        if (a < b) return -1;
        if (a > b) return 1;
        return 0;
    });
}
```

Vala 中的匿名方法是真正的闭包（real closures）。这意味着，你可以在 lambda 表达式内部去访问表达式外部的变量。

```
delegate int IntOperation(int i);

IntOperation curried_add(int a) {
    return (b) => a + b;  // 'a' is an outer variable
}

void main() {
    stdout.printf("2 + 4 = %d\n", curried_add(2)(4));
}
```

在上面的例子中，curried_add（参见 Currying）返回一个新的方法，并且保留值 a。返回的这个新方法直接以数字 4 作为参数，最终的结果就是计算两个数字的和。

命名空间（NameSpaces）

```
namespace NameSpaceName {
    // ...
}
```

在以上两个中括号的所有元素，都是属于命名空间 NameSpaceName 的，任何需要引用其命名空

间中内容的外部代码，必须使用完整的命名，或者在文件中导入该命名空间后再直接使用，比如：

```
using NameSpaceName;
// …
```

假设当前代码用一行 “using Gtk;” 导入了个名为 Gtk 的命名空间，那么就可以简单的用代码 Window 来代替 Gtk.Window。但是在某些情况下，为了避免歧义，仍然需要用完整的命名，比如 GLib.Object 和 Gtk.Object。

在使用 Vala 编程的过程中，命名空间 GLib 是默认被导入的，就好比每个代码文件的开头都被自动添加了一行代码： using Glib。

那些并没有被放入具体命名空间的元素，被一个全局的匿名命名空间所容纳 ，如果因为歧义而需要明确的引用该匿名空间的元素，则需要使用 global:: 前缀。

命名空间是可以嵌套的，一个命名空间可以在另外一个命名空间中定义，也就是可以用 NameSpace1.NameSpace2 形式来命名。

至于其他类型也可以适用这样的命名规则来使得自己是属于某个命名空间的，比如 class NameSpace1.Test { ... }。注意，当这样定义的时候，定义的元素所属的最终命名空间是嵌套在声明中所含的命名空间中的。

结构体（structs ）

```
struct StructName {
    public int a;
}
```

以上定义一个类型为结构体的复合类型，结构体在适当的情况下可以拥有方法，或者私有成员，或者是用 public 关键字显式声明的可以直接访问的成员。（译者注：直至 Vala 0.18.1，仍没有实现对结构体中 private 成员直接访问的错误提示）。

```
struct Color {
    public double red;
    public double green;
    public double blue;
}
```

以下是对这个结构体初始化的各种方法：

```
// without type inference
Color c0 = Color();
Color c1 = new Color();
Color c2 = { 0.5, 0.5, 1.0 };

Color c3 = Color() {
    red = 0.5,
    green = 0.5,
    blue = 1.0
};

// with type inference
var c4 = Color();
var c5 = Color() {
    red = 0.5,
    green = 0.5,
    blue = 1.0
};
```

结构体的内存空间是基于栈的连续分配，在结构体赋值的时候会自动拷贝其内容。

如果想定义一个结构体数组，请参阅 FAQ。

类（Classes）

```
class ClassName : SuperClassName, InterfaceName {
}
```

以上代码定义了一个称作类（class）的引用类型，与结构体不同，基于类的实例的内存是在堆上分配的。关于类有相当多的语法要介绍，其完整介绍将在后面的面向对象编程章节叙述。

接口（Interfaces）

```
interface InterfaceName : SuperInterfaceName {
}
```

以上代码定义了一个称作接口（Interface）的非实例化类型。如果要创建一个接口的实例化类型，必须要首先在非抽象类中实现其抽象方法。事实上，Vala 中的接口还支持 *mixins*。比 Java 还有 C# 还要强大！关于接口的详细内容，在稍后的面向对象编程章节描述。

代码属性（ Code Attributes ）

代码属性指定编译器生成的代码如何在目标平台上运行的细节。语法规则是 [AttributeName] 或者[AttributeName(param1 = value1, param2 = value2, ...)]。

一般使用 vapi 文件生成语言绑定的时候采用 [CCode(...)] 这种语法来凸显需要的属性。另外用的较多的例子是导出 D-Bus 接口时候使用的 [DBus(...)] 属性。

面向对象编程（ Object Oriented Programming ）

虽然使用 Vala 编程，没有强制你要使用面向对象的风格，但是某些重要特性只能在面向对象风格下才能使用。因此，绝大多数情况下，使用面向对象风格来使用 Vala 编程是首选的。和其他许多现代语言一样，想要定义自己的对象类型，首先要定义一个类。

一个类的定义，包含了这个类拥有哪些数据类型，或者是这个类所引用的其他对象类型，还有可以被这个类的实例对象所能调用的方法。定义的类的名字可以包含另外一个类的名字，这样的类称作另外一个类的子类（subclass）。一个子类的实例化，也是其父类的实例化，包含了其父类所拥有的数据或者方法，即便可能无法直接访问这些成员。一个类也可以实现任意多个“接口”，所谓”接口”就是一组必须要在该类中实现的方法，这种包含接口的类的实例，也是该类或者父类所实现的接口的实例。

Vala 中的类可以拥有 “静态（static）” 成员，此修饰符允许数据或者方法为整个类所拥有，而不属于某个具体的实例。访问这种成员，并不一定要有一个从属于该类的具体实例。

基础（Basics）

一个简单的类可以这样定义：

```
public class TestClass : GLib.Object {

    /* Fields */
    public int first_data = 0;
    private int second_data;

    /* Constructor */
    public TestClass() {
        this.second_data = 5;
    }

    /* Method */
    public int method_1() {
        stdout.printf("private data: %d", this.second_data);
        return this.second_data;
    }
}
```

以上代码定义了一个新的类（此类型将由 gobject 库的类型系统自动注册），此类拥有 3 个成员，其中：在开头定义了 2 个整型的数据成员，还有一个返回值为整型的方法成员，名为 method_1。从类的定义上得知，该类是 GLib.Object 的子类，因此该类的所有实例也是 Object 的实例，也拥有 Object 所拥有的全部成员。由于此类继承自 Object，那么 Vala 中的某些特性，可以直接通过访问 Object 的特性来获得。

该类被声明为 public 型（类默认为 internal 型）。此声明表示该类可以被外部的代码文件直接引用，如果你是使用 glib / gobject 的 C 程序员，你会发觉，这等同于将该类声明在一个 C 风格的头文件中，使得其他代码文件可以直接包含其实现。

类的成员同样可以用 public 或者 private 关键字来声明。其中第一个成员 first_data 被声明为 public 型，因此该成员对类的使用者是可见的，可以直接访问或者修改。而第二个成员 second_data 是被声明为 private 型，因此该成员只能被类本身的代码所引用。Vala 提供以下四种不同的访问权限修饰符：

public	没有访问限制。
private	访问仅限于类的内部。在没有指明访问修饰符情况下，这是默认的。

protected	访问仅限于该类，或者继承自该类的类的内部。
internal	访问仅限于从属同个包中的类。

使用构造方法 TestClass() 来初始化每个类的实例。可以看出构造方法的名称和类的名称一模一样，构造方法可以拥有零个或者多个参数，要注意的是构造方法是没有返回值的。

代码中最后一部分则是一个名为 method_1，返回值类型为整型的方法的定义。由于该方法没有声明为 static 型，因此该方法只能被每个实例单独调用，并通过使用 this 引用来访问实例中的成员。this 始终指向当前调用此方法的实例的引用，除非可能会产生歧义，否则 this 标识符其实是可以省略的。

你可以像如下代码所示，使用一个新的类：

```
TestClass t = new TestClass();
t.first_data = 5;
t.method_1();
```

构造（Construction）

Vala 支持两种风格的构造方法：分别是 Java/C# 风格和 Gobjct 风格，我们当前主要关注前者，而后者将在此章的末尾介绍。

由于 Vala 不支持方法重载，因此构造方法重载也是不支持的，也就是不能拥有多个名称一样的构造方法。但这并没不是什么缺陷，因为 Vala 支持命名构造方法，如果需要提供多个构造方法，只需要增加不同的命名即可：

```
public class Button : Object {
    public Button() {
    }
    public Button.with_label(string label) {
    }

    public Button.from_stock(string stock_id) {
    }
}
```

那么实例化的方法如下：

```
    new Button();
    new Button.with_lable(“Click me”);
    new Button.from_stock(Gtk.STOCK_OK);
```

可以通过使用 this() 或者 this.name_extension() 来串联构造方法：

```
    public class Point : Object {
    public double x;
    public double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public Point.rectangular(double x, double y) {
        this(x, y);
    }
    public Point.polar(double radius, double angle) {
        this.rectangular(radius * Math.cos(angle), radius * Math.sin(angle));
    }
}

void main() {
    var p1 = new Point.rectangular(5.7, 1.2);
    var p2 = new Point.polar(5.7, 1.2);
}
```

析构（Destruction）

尽管 Vala 负责管理内存，但是仍然会需要添加自己的析构方法来手动管理内存，来销毁那些必须销毁的资源。定义析构方法的语法和 C# 还有 C++ 类似：

```
    class Demo : Object {
        ~Demo() {
            stdout.printf(“in destructor”);
        }
```

```
}
```

由于 Vala 的内存管理是基于引用计数，而非垃圾回收机制，因此析构方法是确定不变的，可以实现 RAII 模式来管理各种资源（诸如：关闭流，关闭数据库连接等）。

信号（Signals）

信号是 Glib 库中的 Object 类提供的一个系统，在 Vala 中，对从 Object 类派生的类使用该系统更加容易。Vala 中的信号相当于 C# 中的事件（event），相对 Java 则是对其事件监听机制的替代实现。简短的说，信号是在同一时间内调用任意次数的同一外部方法的简单手段，至于实际的方法调用是在 gobject 内部的，Vala 程序员无需关心。

信号是作为一个成员在类中定义的，并且在形式上就是个没有实现体的方法。信号的处理方法可以使用 connect() 方法来动态添加。为了更加深入的演示，以下的例子使用了 lambda 表达式，这是 Vala 编程中添加信号处理方法经常使用的风格：

```
public class Test : GLib.Object {

    public signal void sig_1(int a);

    public static int main(string[] args) {

        Test t1 = new Test();

        t1.sig_1.connect((t, a) => {
            stdout.printf("%d\n", a);
        });

        t1.sig_1(5);
        return 0;
    }
}
```

以上代码用我们已经熟悉的语法，新定义了一个名为“Test”的类。其中第一个成员是一个名为“sig_1”，需要传入一个整型参数的信号。在方法 main 中，我们首先创建了一个 Test 类的实例，这一步是必须的，因为信号必须依附于一个类的实例。接着，我们将一个信号的处理方法（handler)赋给我们实例的 “sig_1”成员，在这里使用的是 lambda 表达式方式。从该 lambda 表达式定义来看，此方法

接受 2 个参数，分别为“t”和 “a”，但是并没有指明参数类型。这里之所以可以这样的简洁，是因为 Vala 已经由前面的代码知道了信号的定义，也因此而知道了所需要的参数类型。

之所以在信号处理方法（handler）中传递 2 个参数，是因为不论信号什么时候，被谁给发出，总是将发出该信号的对象实例作为第一个参数传入给该方法（handlers），以指明发出信号的对象实例，而第二个参数则是由信号提供的用来自身用途的。

最后，我们迫不及待的发出了个信号。我们是将信号以普通方法的形式调用，来发出信号的，并将该信号委托给 gobject 来维护并转发给所有关注此信号的处理方法们（handlers），在 Vala 中使用信号系统，并不需要了解整个系统的内在机制。

注意：

当前只能使用 public 修饰符，所有信号可以被任意代码连接（connect）和发出（emit）！

自从 2010 年四月之后，可以使用任意的标记组合来标注信号：

```
[Signal (action=true, detailed=true, run=true, no_recurse=true, no_hooks=true)]
public signal void sig_1 ();
```

属性（Properties）

针对面向对象编程，一个总的原则是面对类的使用者，要将类的实现细节尽量隐藏（information hiding principle），这样即便将来对类的内部实现做修改，也不会破坏公共的 API 接口。其中一个原则就是将某些域作为私有的（private），然后提供获取，设置这些域的方法（getters and setters）。

如果你是 Java 程序员，你可能自然地联想到如下的代码：

```
class Person : Object {
    private int age = 32;
    public int get_age() {
        return this.age;
    }
    public void set_age(int age) {
        this.age = age;
    }
}
```

以上代码是可行的，但是 Vala 可以做的更好。以上代码主要问题是，使用起来略显笨拙。假设我们要将一个人的年纪增加一岁：

```
var alice = new Person();
alice.set_age(alice.get_age() + 1);
```

而这正是 Vala 属性大放光彩的时刻：

```
class Person : Object {

    private int _age = 32;  // underscore prefix to avoid name clash with
                               property

    /* Property */

    public int age {

        get { return _age; }

        set { _age = value; }
    }
}
```

此语法可能 C# 程序员相对熟悉些。某个属性拥有 get 和 set 两个代码块来获取和设置其值。value 是一个关键字，用来表示新的应当赋值给属性的值。

现在访问该属性，就好像是被声明为了 public 字段一样。其实这功能的背后是 get 和 set 代码块在支撑。

```
var alice = new Person();
alice.age = alice.age + 1;  // or even shorter:
alice.age++;
```

如果只是需要像上面那样实现一套标准的属性，可以更简洁的书写代码：

```
class Person : Object {

    /* Property with standard getter and setter and default value */

    public int age { get; set; default = 32; }
}
```

通过使用属性，我们可以改变内部的实现，而不用更改对外的公共 API 接口，比如：

```
static int current_year = 2525;
```

```
class Person : Object {

    private int year_of_birth = 2493;

    public int age {

        get { return current_year - year_of_birth; }

        set { year_of_birth = current_year - value; }

    }
}
```

这一次，age 的结果取决于 year_of_birth。注意，相比在 get 和 set 代码段中做简单的数值访问和赋值，其实可以做更多。比如可以对数据库的访问，登陆，缓存更新等。

如果你想将某个属性作为只读的来提供给类的使用者，只需将 setter 声明为私有（private）：

```
public int age { get; private set; default = 32; }
```

或者，你可以悬空 set 代码段：

```
class Person : Object {

    private int _age = 32;

    public int age {

        get { return _age; }
    }
}
```

属性不仅仅可以有正式的名称，也可以赋予个昵称（nick）用来简单的描述属性，或者赋予导语（blurb）用来详细的描述属性。你可以使用代码属性来达到此目的：

```
[Description(nick = "age in years", blurb = "This is the person's age in years")]

public int age { get; set; default = 32; }


void main() {
    Type type = typeof(Person);
    ObjectClass ocl = (ObjectClass) type.class_ref ();
    unowned ParamSpec? spec = ((ObjectClass)ocl).find_property("age");
    stdout.printf ("nick is %s\n", spec.get_nick());
}
```

属性还有其附加的描述可以动态的被查询。某些程序，比如 Glade 图形用户接口设计程序，就使用了这些信息。通过这种方式，Glade 使用人类可读的文本来描述 GTK+ 中 widgets 的属性。

所有继承自 Glib.Object 的类，都包含一个名为 notify 的信号。该信号在其对象的属性发生改变时发出。因此可以连接（connect）该信号，如果有兴趣在属性的改变时获得通知的话：

```
obj.notify.connect((s, p) => {
    stdout.printf("Property '%s' has changed!\n", p.name);
});
```

注意，当你必须使用字符串来表示属性的名称的时候，字符串中的下划线将被转换成破折号，这种转换是 Gobject 的属性名转换机制自动完成的。比如 " my_property_name" 将被自动转换为 " my-property-name"。

属性默认的改变通知行为可以在属性的声明前使用 CCode 代码属性来禁用：

```
public class MyObject : Object {
    [CCode(notify = false)]
    // notify signal is NOT emitted upon changes in the property
    public int without_notification { get; set; }
    // notify signal is emitted upon changes in the property
    public int with_notification { get; set; }
}
```

此外还有一种称作“构造属性”的属性。将在后面关于 gobject 风格构造的章节介绍。

继承（Inheritance）

在 Vala 中，一个类可以继承自一个，或着不继承自任何类。实际一般是继承自一个，但并不像 Java 语言那样提供隐式继承。

当定义一个类继承自另外一个类时，就自然而然地确立了这两个类之间的关系：这个类的实例也是其父类的实例。也就是说，针对父类实例的操作同样适用于其子类。因此，那些需要父类实例的地方，同样可以用子类实例来替代。

当定义一个类的时候，需要精确的分别控制类中方法和数据的访问权限。以下的例子，演示了可以

定义的选项范围：

```
class SuperClass : GLib.Object {

    private int data;

    public SuperClass(int data) {
        this.data = data;
    }
    protected void protected_method() {
    }
    public static void public_static_method() {
    }
}

class SubClass : SuperClass {

    public SubClass() {
        base(10);
    }
}
```

data 是父类实例的一个数据成员，基于该类的所有实例，都拥有该成员，由于被声明为私有的（private），因此只有父类中的代码才能够访问该成员。

protected_method 是父类实例中的一个方法成员，该方法成员只能被父类或者其子类中的代码来访问，这也是 protected 修饰符的作用结果。

public_static_method 拥有两个修饰符，其中 static 修饰符意味着该方法的调用可以在父类或者子类的实例之外。因此，该方法在执行的过程中不能访问 this 引用。至于 public 修饰符，则说明了该方法可以在任何代码调用，而不用管和父类或者子类的关系。

有了如上定义，子类的实例可以包含所有三个父类中的成员，但仅仅能访问非 private 修饰符声明的那些成员。外部代码则只能访问被声明为 public 的方法。

使用 base 构造方法，则可以将子类的构造方法串联上其基类的构造方法。

抽象类（Abstract Classes）

可以用称作抽象（abstract）的修饰符来声明方法。用此修饰符声明的方法不需要在当前的类中实现，但是必须在使用前在其子类中实现。这样允许在所有实例中调用同一类型的操作，却又可以是各自不同版本的实现。

一个类如果包含抽象方法，则必须也要用抽象（abstract）来声明该类，结果就是该类不能被实例化：

```
public abstract class Animal : Object {

    public void eat() {

        stdout.printf("*chomp chomp*\n");

    }

    public abstract void say_hello();
}

public class Tiger : Animal {

    public override void say_hello() {

        stdout.printf("*roar*\n");

    }
}

public class Duck : Animal {

    public override void say_hello() {

        stdout.printf("*quack*\n");

    }
}
```

抽象方法的实现必须要用 override 来标记，属性同样可以抽象的使用。

接口 / 混合（Interfaces / Mixins）

Vala 编程中，类中可以包含数个接口的实现。所谓接口，是一个类型，和类很相似，但是不能被实例化。通过实现一个或多个接口，一个类可以被声明为该类的实例同时也是这些接口的实例。因此，这个类的实例可以用在那些需要这些接口实例的那些地方。

实现接口的步骤和继承拥有抽象方法的类的步骤相似，必须在实现所有已经声明但还未实现的方法后，这个类才可以能被使用。

一个简单的接口定义如下：

```
public interface ITest : GLib.Object {
    public abstract int data_1 { get; set; }
    public abstract void method_1();
}
```

以上代码定义了一个名为”ITest“ 的接口， 首先该接口继承自 Glib.Object 并且拥有两个成员。其中 data_1 是属性，和之前介绍过的一样，只是多了个修饰符 abstract。因此，Vala 在此处并未实现该属性，取而代之，则要求那些实现该接口的类，必须要有一个名为 ” data_1“ 的属性，并且还要提供 get 和 set 访问方法。第二个成员是名为 “method_1” 的方法，此处的意思是，那些实现该接口的类，必须也要实现此方法。

完整实现此接口最简单可行的方法是：

```
public class Test1 : GLib.Object, ITest {
    public int data_1 { get; set; }
    public void method_1() {
    }
}
```

然后如下使用此类：

```
var t = new Test1();
t.method_1();
ITest i = t;
i.method_1();
```

Vala 中的接口可以不继承自其他的接口，还可以声明为以其他接口为先决条件。比如，可以定义一个类需要实现一个 “List” 接口，并且还需要实现一个 “Collection” 接口。实现此作用的语法，非常类似在一个类中实现一个接口：

```
public interface List : Collection {
}
```

以上 List 接口的定义只有在 Collection 也被实现的前提下，才能在类中实现该接口，因此 Vala 使用接下来的风格将所有必须要实现的接口（必要条件类），声明在一串需求列表中。（译者注：一定要注意声明的先后顺序，否则编译不会报错，但是运行的时候会报错，Vala 0.81 版本如此!）

```
public class ListClass : GLib.Object, Collection, List {
}
```

接口同样可以将某一个类作为必要条件。如果在条件列表中加上一个类的名字，那么这个接口只能在该类的子类中实现。通常这样限制是为了确保接口的实例同样是 GLib.Object 的子类，使得某些特性可以使用，比如属性接口。

实际上，这和一个接口继承自其他接口的说法，只是技术上的差别（这里不叫继承，叫必要条件），在这一点上 Vala 的做法和其他语言是一致的，只是该特性称作必要条件。

Vala 中的接口和 Java 或 C# 中的接口有个很大的不同，Vala 的接口中可以定义非抽象的方法！Vala 确实允许在接口中定义方法，也因此抽象方法必须要用 abstract 来声明。所以 Vala 的接口可以实现混合（mixins），这是一种受限制的多重继承。

多态（Polymorphism）

多态特性提供了同一个对象当作不同类型来使用的特性。关于此特性的几种技术前面已经描述过了，在这里先阐述 Vala 如何做到的：一个类的实例可以当作某个父类的实例或者任意在类中实现的接口的实例来使用，而不用关心该实例实际的类型。

也就是说通过此强大的特性，可以使得子类和父类在使用相同特性的时候却表现出不同的结果。这个概念用语言表述很难，因此，不如直接给个例子来阐明以下代码会发生什么，不过结果可能会出乎你的意料哦：

```
class SuperClass : GLib.Object {
    public void method_1() {
        stdout.printf("SuperClass.method_1()\n");
    }
}

class SubClass : SuperClass {
    public void method_1() {
```

```
            stdout.printf("SubClass.method_1()\n");
        }
    }
```

以上两个类都实现了名为“method_1”的方法，因此 “SubClass”类拥有 2 个名为“method_1”的方法，其中一个是继承自“SuperClass”。可以分别像以下代码所示的来调用：

```
    SubClass o1 = new SubClass();
    o1.method_1();
    SuperClass o2 = o1;
    o2.method_1();
```

这段代码的实际运行结果是，两个不同的方法被分别调用。其中第二行中的“o1”是 “SubClass”类型，调用的方法是其自身类所实现的方法。第四行中的 “o2”是 “SuperClass”类型，调用的是该类实现的方法。

这段代码有个问题，就是任何引用为 “SuperClass”的代码调用的方法都将调用 “SuperClass”内部实现的方法，即便实际的对象类型是 “SubClass”类。改变这种现象的方法是使用虚方法（virtual methods）。如下是针对上个例子重写的新示例代码：

```
    class SuperClass : GLib.Object {
        public virtual void method_1() {
            stdout.printf("SuperClass.method_1()\n");
        }
    }

    class SubClass : SuperClass {
        public override void method_1() {
            stdout.printf("SubClass.method_1()\n");
        }
    }
```

仍像之前的例子运行以上的代码，“SubClass”的 “method_1”方法将被调用两次。因为我们使用“virtual”修饰符告诉系统方法“method_1”是虚方法。如果在子类中被覆盖的话，那么无论调用者将子类对象实例当作何种类型实例，都将调用子类所实现的方法版本。

以上的区别可能对 C++ 程序员来说非常熟悉，但是事实上却和 Java 语言的风格正好相反， 在 Java

语言中则是尽力地采取措施，以防止方法被虚化。

至此，你可能意识到，当一个方法被声明为抽象（abstract）时，该方法同时肯定也是虚方法。否则，无法根据实际的类型来执行各自版本的方法。当在一个子类中实现一个抽象方法时，你也可以在声明的时候加上“override” 修饰符，从而使方法拥有虚方法特性，让子类型在你希望的情况下，可以做相同的事情。

也可以使用相同的方法在子类中更改接口方法的实现。要达到此目的，则在初始声明的地方加上“virtual”修饰符，然后如果必要的话，再在子类中覆盖此方法的实现。

当编写一个类的时候，使用从父类中继承的功能是非常常见的。但是当继承树中有多个继承的时候，方法名将会比较复杂。通常情况是，覆盖了一个虚方法并提供了附加的功能，但却又需要调用父类实现的方法版本，此时 Vala 提供“base”关键字来满足此需求：

```
public override void method_name() {
    base.method_name();
    extra_task();
}
```

Vala 也允许属性是为虚化的：

```
class SuperClass : GLib.Object {
    public virtual string prop_1 {
        get {
            return "SuperClass.prop_1";
        }
    }
}

class SubClass : SuperClass {
    public override string prop_1 {
        get {
        return "SubClass.prop_1";
        }
    }
}
```

方法隐藏（Method Hiding）

通过使用 new 修饰符你可以用同一个名称新建一个方法来隐藏被继承的方法。新建的方法可以拥有不同的特征。方法隐藏不同于方法覆盖，因为方法隐藏没有表现出多态的特性。

```
class Foo : Object {
    public void my_method() { }
}

class Bar : Foo {
    public new void my_method() { }
}
```

译者注：还可以这样新建一个不同特征的方法：

```
class Foo : Object {
    public void my_method() { }
}

class Bar : Foo {
    public new void my_method(int param) {
        stdout.printf (“param = %d\n”, param);
    }
}
```

你依然可以通过类型转换，将子类转换为基类或者原始接口类型，来调用原始的方法：

```
void main() {
    var bar = new Bar();
    bar.my_method();
    (bar as Foo).my_method();
}
```

运行时类型信息（Run-Time Type Information）

由于 Vala 中的类都是动态注册的，并且每个实例都携带了自身的类型信息，所以可以使用 is 操作

符来动态检查一个对象的类型：

```
bool b = object is SomeTypeName;
```

也可以使用 get_type() 方法来获取 object 实例的类型信息：

```
Type type = object.get_type();
stdout.printf("%s\n", type.name());
```

还可以直接使用 typeof() 操作符来获取类型信息，然后基于此类型信息调用 Object.new() 来创建新的实例。

```
Type type = typeof(Foo);
Foo foo = (Foo) Object.new(type);
```

但是到底是哪个构造方法将被调用呢？相关内容将在 gobject 风格的构造章节描述。

动态类型转换（Dynamic Type Casting）

如果要想动态转换类型，只要对相应的变量运用表达式 as DesiredTypeName 即可。Vala 将会自动使用动态类型检查机制来确保转换是否合理正确，如果是非法的转换，则将返回 null。不过此方法要求，待转换的类型和目标类型都必须是引用类型。比如：

```
Button b = widget as Button;
```

如果 widget 实例的类不是 Button 的子类，也不是在 Button 类中实现的接口，那么 b 的值最终是 null。这种转换形式与下面的形式等价：

```
Button b = (widget is Button) ? (Button) widget : null;
```

泛型（Generics）

Vala 拥有一套运行时的泛型系统，即某个类的实例类型在构造时可以为该实例指定一个或一组类型。通常是指某个对象实例拥有指定自身所需要存储的数据类型的能力。比如，可以实现一个针对特定类型对象的链表。此时 Vala 确保只有需要的类型才能被添加到该链表中。

Vala 是在程序运行时操作泛型系统。当定义了指定类型的类，即便每个实例都分别指定了不同的需

求类型，但系统中仍只存在一个类，这一点和 C++ 那样为每个需求类型都创建一个类正好相反。Vala 此时和 Java 类似。但这也造成了很多影响，最重要的是：被声明为静态的成员（static members）是所有类型所共有的，不论每个实例上的指定如何，对于在子类中使用的泛型，同样适用于其父类。

以下代码演示了如何使用泛型系统来定义一个最小化的封装类（wrapper class）：

```
public class Wrapper<G> : GLib.Object {
    private G data;
    public void set_data(G data) {
        this.data = data;
    }
    public G get_data() {
        return this.data;
    }
}
```

该“Wrapper”类对象在实例化前必须为其指定一个需求类型，其中该需求类型用“G”来代替，此类的实例将存储一个类型为“G“ 的数据，并且使用 set 和 get 方法来访问该数据。（这里使用这样的例子，是为了说明当前的泛型系统不支持属性特性，需要实现简单的 get 和 set 方法。）

为了实例化该类，必须指定一个需求类型，比如 Vala 内建的 string 类型（Vala 并不限制泛型系统中所能使用的类型）。可以简单得如下创建实例：

```
var wrapper = new Wrapper<string>();
wrapper.set_data("test");
var data = wrapper.get_data();
```

由此可以看出，当从该类中取出数据并赋值给变量时，并没有明确的指定类型，因为 Vala 已经知道每个封装中的对象类型，然后将一切转换都替你办妥。

基于 Vala 并不会因为泛型而创建许多类这个事实，你可以如下编写代码：

```
class TestClass : GLib.Object {
}

void accept_object_wrapper(Wrapper<GLib.Object> w) {
}

...
```

```
var test_wrapper = new Wrapper<TestClass>();
accept_object_wrapper(test_wrapper);
…
```

因为所有的“TestClass”实例也是 Object 的实例，因此方法“accept_object_wrapper”非常乐意接受该对象的传入，然后将他当作被封装的对象，即使该对象实际上是 GLib.Object 的实例。

GObject 风格的构造（GObject-Style Construction）

正如之前就提到过的，Vala 提供另一种和之前描述不一样的构造风格。此风格也许更能够被从 GObject 或者 Java，C# 语言转向 Vala 的程序员所接受。gobject 风格的构造提供了一些新的语法元素：构造属性，特别的 Object(...) 调用和 construct 代码段。让我们看看它是如何工作的：

```
public class Person : Object {
    /* Construction properties */
    public string name { get; construct; }
    public int age { get; construct set; }
    public Person(string name) {
        Object(name: name);
    }
    public Person.with_age(string name, int years) {
        Object(name: name, age: years);
    }

    construct {
        // do anything else
        stdout.printf("Welcome %s\n", this.name);
    }
}
```

基于 gobject 风格的构造，每个构造方法是使用 Object() 调用来设置属性的，因此这样的属性被称作构造属性（construct properties），Object 调用使用 property : value 的形式来传入一系列的参数来初始化属性，这些被初始化的属性必须在定义的时候用 construct 修饰。对于这些属性，都由给定的值来初始化，紧接着根据继承的层次， 一直从 GLib.Object 开始向下的所有子类的 construct {} 代码段将依次被调用。

construct {} 代码段在类的实例被创建时，是肯定会被调用的，即便创建的是子类型。其中并不包含任何参数，也不返回任何类型的值。需要的话，在此代码段内部，可以调用其他方法来设置成员变量。

如果某个构造属性没有用 set 来声明，也被称作仅构造属性（construct only propetry），也就意味着该属性只能在构造的时候被赋值，其他时间段是不允许赋值的。在以上的例子中，name 是仅构造属性。

以下是在 gobject-based 库文档中能看到的各种属性命名方法的总结：

```
public int a { get; private set; }    // Read
public int b { private get; set; }    // Write
public int c { get; set; }            // Read / Write
public int d { get; set construct; }  // Read / Write / Construct
public int e { get; construct; }      // Read / Write-Construct-Only
```

在某些情况下，你可能有这样的需求：不论类的实例是在何时被创建的，只是在该类由 GObject 系统动态创建时来执行某段初始化代码。用 GObject 的术语来说，就是指调用类的 class_init 方法。在 Java 中，众所周知的方法是使用静态初始化代码段（static initializer blocks）。在 Vala 中则这样使用：

```
/* This snippet of code is run when the class * is registered with the type
   system*/
static construct {
    ...
}
```

高级特性（Advanced Features）

断言和简化编程（Assertions and Contract Programming）

程序员通过使用断言（*assertions*）在程序运行时对当前运行上下文的假设条件做检查。断言的语法是 asseert(condition)。如果一个断言检查失败，则程序将终止运行并且抛出适当的错误信息。在标准的 GLib 命名空间中存在如下几种断言方法：

```
assert_not_reached()
```

```
return_if_fail(bool exp)

return_if_reached()

warn_if_fail(bool expr)

warn_if_reached()
```

可以使用断言来检查方法的各个参数，以防止参数值为 null。但是这并不是必要的步骤，因为 Vala 隐式地自动对所有没有用“?”标记的参数值做是否为 null 的检查。

```
void method_name(Foo foo, Bar bar) {
    /* Not necessary, Vala does that for you:
    return_if_fail(foo != null);
    return_if_fail(bar != null);
    */
}
```

Vala 支持基本的简化编程（contract programming）特性，使得方法可以约束一个前提条件（requires）或后置条件（ensures）来分别约束方法开始和结束的条件。

```
double method_name(int x, double d)
    requires (x > 0 && x < 10)
    requires (d >= 0.0 && d <= 1.0)
    ensures (result >= 0.0 && result <= 10.0)
{
    return d * x;
}
```

其中 result 是个特殊的变量，用来代表一个方法的返回值。

错误处理（Error Handling）

Glib 拥有一个在运行时管理程序异常的系统，称作 GError。Vala 将其转换为与现代编程语言类似的处理形式，但是实现却和 Java 或者 C# 并不一样。首先，GError 为了能在运行时捕获错误，做了非

常特别的设计，当使用这个类型来处理错误时，错误的因素并不能事先得知，直到程序运行，并且只能捕获非致命的错误。建议不要滥用 GError，比如仅仅是需要报告一个非法的参数变量。假设一个方法需要一个大于 0 的参数，可以使用前面提到的简化编程中介绍的前提条件或者断言方法使得在传入一个负值的时候让程序报错。

Vala 程序的错误也就是所谓的异常，意味着在某些地方，该错误必须要被处理。但是如果没有在代码中显式地捕获错误，那么 Vala 的编译器仅仅是发出个警告信息，却不终止编译的过程。

使用异常（以 Vala 的术语来说也叫错误）的情形有：

1）声明可以抛出异常的方法：

```
void my_method() throws IOError {
    // ...
}
```

2）在必要的情况下抛出错误：

```
if (something_went_wrong) {
    throw new IOError.FILE_NOT_FOUND("Requested file could not be found.");
}
```

3）在调用代码中捕获错误：

```
try {
    my_method();
} catch (IOError e) {
    stdout.printf("Error: %s\n", e.message);
}
```

4）使用“is”操作符来测试一个错误码：

```
IOChannel channel;

try {
    channel = new IOChannel.file("/tmp/my_lock", "w");
} catch (FileError e) {
    if(e is FileError.EXIST) {
```

```
            throw e;
        }
        Glib.error("%s", e.message);
    }
```

以上的使用形式或多或少都在其他语言中出现过，但是定义一个专门的错误类型估计是独有的。Errors 拥有三个组件，分别是”错误域“（domain），“错误码”（code） 还有“错误信息“（message）。其中”信息”（message）上面代码已经用到了，即当错误发生时，对该错误的一段文本描述。错误的域（domains）描述了错误的类型，相当于 Java 或者类似语言中的“Exception”子类。在上面的例子中，抛出了一个域（domain）为“IOError”的错误。最后，错误码精确得描述了发生的问题。每个域（domain）都含有一个或多个错误码，在以上的例子中，错误码是 “FILE_NOT_FOUND”。

定义错误的信息的是基于 glib 来实现的，为了让以上的例子能够正常运行，我们需要一个如下的错误定义：

```
    public errordomain IOError {
        FILE_NOT_FOUND
    }
```

当需要捕获错误的时候，首先指定需要捕获的错误域，之后当这个域中的错误被抛出时，错误处理中的代码将被执行。从错误对象中，可以根据需要提取错误码或者错误信息。如果需要捕获不止一个域的错误，只需要增加额外的捕获（catch）代码段就好。此外，还有个可选的方式，即在 try 或者 catch 代码段的后面附加一个称作 finally 的代码段。其中的代码始终在最后被执行，不论错误是否被抛出或者任意的捕获代码是否被执行，或者错误没有被处理，或者错误将会被再次抛出。通过这种方法，可以确保那些在 try 代码段中申请的资源在最终都会被销毁掉，不论错误是否发生。关于此特性，完整的例子如下：

```
    public errordomain ErrorType1 {
        CODE_1A
    }

    public errordomain ErrorType2 {
        CODE_2A,
        CODE_2B
    }

    public class Test : GLib.Object {
        public static void thrower() throws ErrorType1, ErrorType2 {
```

```
            throw new ErrorType1.CODE_1A("Error");
        }
        public static void catcher() throws ErrorType2 {
            try {
                thrower();
            } catch (ErrorType1 e) {
                // Deal with ErrorType1
            } finally {
                // Tidy up
            }
        }
        public static int main(string[] args) {
            try {
                catcher();
            } catch (ErrorType2 e) {
                // Deal with ErrorType2
                if (e is ErrorType2.CODE_2B) {
                    // Deal with this code
                }
            }
            return 0;
        }
    }
```

这个例子中有两个错误域，都是被名为“thrower”的方法抛出。方法“catcher”只能抛出第二种错误类型，因此必须处理被“thrower”抛出的第一种错误类型。最终，方法“main”将处理所有从“catcher”方法抛出的错误。

参数路径（Parameter Directions）

Vala 中的一个方法可以传入零个或多个参数。当一个方法被调用时，有如下的默认行为：

- 任何值类型的参数，将被拷贝到方法执行时的本地空间。

- 任何引用类型参数，不会被拷贝，取而代之的是只是引用这些参数，然后传递给方法。

以上的行为可以通过使用 'ref' 和 'out' 修饰符来改变。

'out' 以调用者的视角来看：

传递给方法的该参数是没有被初始化的，希望该方法将其初始化并返回给我。

'out' 以被调用者的视角来看：

该参数没有被初始化过的，而我必须要初始化它。

'ref' 以调用者的视角来看：

传递给方法的该参数是已经初始化的，方法可以改变或者不改变其值。

'ref' 以被调用者的视角来看：

该参数是被初始化过的，我可以改变或者不改变其值。

```
void method_1(int a, out int b, ref int c) { ... }
void method_2(Object o, out Object p, ref Object q) { ... }
```

以上的两个方法可以用如下的形式调用：

```
int a = 1;
int b;
int c = 3;
method_1(a, out b, ref c);

Object o = new Object();
Object p;
Object q = new Object();
method_2(o, out p, ref q);
```

以上代码分别如下对待每个变量：

- “a”是个值类型。其值将会被拷贝到新方法内存区域中，因此在该方法中改变该参数的值对调用者是不可见的。

- “b”也是个值类型，但是被修饰为 out 参数类型。这种情况下，不会拷贝其值，取而代之的是用指向该数据的引用传递给方法。因此在方法中对该参数值的改变，是对调用者可见的。

- “c”和 “b”一样被对待，唯一的区别是方法中修饰符不一样而已。

- “o”是引用类型。方法被传递了和调用者一致的引用类型。因此方法可以改变这个对象，但是如果该对象被重新分配了，那么之后的改变对调用者也是不可见的。

- “p”和 “o”的类型一样，但是被修饰为 out 型参数。这意味着，方法将接受该指向该对象的引用。也因此将使用新的引用来替换该引用，当从方法返回到调用者时，之前的引用将被替换。如果使用了这种类型的参数，而没有赋予新的引用，那么该参数将被置为空（null）。

- “q”依然是一样的类型。此时被和“p”一样对待，除了一个非常重要的不同点，那就是方法可以选择访问该引用却不改变该引用。Vala 会确保“q”确实引用某个对象的实例，并且不会被置空（null）。

关于如何实现方法 method_1() 有个例子：

```
void method_1(int a, out int b, ref int c) {
    b = a + c;
    c = 3;
}
```

由于在方法中将参数 “b”设为 out 型，因此 Vala 会确保方法中的 “b”不为 null，之后你可以安全地直接将 null 作为第二个参数传给该方法，如果你并不关心其值的话。

容器（Collections）

Gee 是个由 Vala 编写的一套容器类库。对于用户来说，其中的类和 Java 中的基础类很相似。Gee 由一组不同的接口和类型用不同的方式来实现。

如果你想在自己的应用程序中使用 Gee，需要在你的系统中独立地安装该库。可以从地址 http://live.gnome.org/Libgee 来获得。为了能够链接 Gee 库，需要在编译程序的时候使用 --pkg gee-0.8 选项。

其中基本的容器类型有：

- Lists ： 一组有序的同一类型元素，使用索引来访问。
- Sets ： 一组无序的任意类型元素。
- Maps ： 一组无序的任意类型元素，使用索引来访问。

其中所有的 Lists 和 Sets 都实现了 Collection 接口，而 Maps 则都实现了 Map 接口，Lists 和 Sets 也同样分别实现了 List 和 Set 接口。这些常见的接口说明不只是相似类型的容器可以交互使用，还可以使用现成的代码，同样的接口来实现新的容器。

对于容器类来说另一个普遍的接口是可迭代的接口（Iterable）。也就是可以使用一组标准的方法来遍历容器类中的各个存储对象，或者直接使用 Vala 提供的 foreach 语法。

库中所有的类和接口都使用了泛型系统。因此在初始化的时候必须指定容器需要容纳的对象类型。Vala 将确保只有符合要求的类型才能被装进容器，并且同时确保对象被检索到的时候，返回的是正确的类型。

Full Gee API documentation, Gee Examples

比较重要的 Gee 类有：

ArrayList<G>

实现了：Iterable<G>, Collection<G>, List<G>

HashMap<K,V>

实现了：Iterable<Entry<K,V>>, Map<K,V>

表中的类型 k 和元素类型 v 作 1 对 1 的映射。映射的原理是为每个 key 计算一个哈希值，其计算方法可以通过提供自己的 hash 和 key equal 方法来改变。

可以在构造的时候，如下传入自己的 hash 和 equal 方法：

```
var map = new Gee.HashMap<Foo, Object>(foo_hash, foo_equal);
```

对于字符串（strings）和整型（integers）类型，hash 和 equal 方法将自动提供一组默认的方法，至于对象类型，则默认由它们各自的引用来作为 key。可以提供自己的 hash 和 equal 方法来覆盖

其默认的行为。

HashSet<G>

实现了：Iterable<G>, Collection<G>, Set<G>

一组类型为 G 的元素。每个 key 将被计算出一个 hash 值，以检测是否有重复元素，同样可以提供自己的 hash 和 equal 方法来做特殊用途。

Read-Only Views

你可以通过容器的 read_only_view 属性来获得容器的只读视图，比如 my_map.read_only_view。此操作将返回一个和对应容器一模一样的容器，但是不允许对该容器做任何的修改。

语法支撑的方法（Methods With Syntax Support）

Vala 识别某些特别的名称和特征的方法，并对他们提供语法支持。比如，如果某个类型拥有 contains() 方法，那么这个类型的对象就可以使用 in 操作符。以下的表格列出了这些特殊的方法。

T 和 Tn 只是类型占位符，当实际使用的时候需要用具体的类型来替代。

Indexers

T2 get(T1 index)	index access: obj[index]
void set(T1 index, T2 item)	index assignment: obj[index] = item

Indexers with multiple indices

T3 get(T1 index1, T2 index2)	index access: obj[index1, index2]
void set(T1 index1, T2 index2, T3 item)	index assignment: obj[index1, index2] = item
(... and so on for more indices)	

Others

T slice(long start, long end)	slicing: obj[start:end]
bool contains(T needle)	in operator: bool b = needle in obj

`string to_string()`	support within string templates: `@"$obj"`
`Iterator iterator()`	iterable via foreach
`T2 get(T1 index)` `T1 size { get; }`	iterable via foreach

迭代类型可以拥有任何名称，但是必须实现以下两个方法中的一个：

`bool next()` `T get()`	standard iterator protocol: iterating until *`.next()`* returns false. The current element is retrieved via *`.get()`*.
`T? next_value()`	alternative iterator protocol: If the iterator object has a *`.next_value()`* function that returns a nullable type then we iterate by calling this function until it returns null.

以下的例子实现了这些方法：

```
    public class EvenNumbers {

        public int get(int index) {

            return index * 2;

        }

        public bool contains(int i) {

            return i % 2 == 0;

        }

        public string to_string() {

            return "[This object enumerates even numbers]";

        }

        public Iterator iterator() {

            return new Iterator(this);

        }


        public class Iterator {

            private int index;

            private EvenNumbers even;

            public Iterator(EvenNumbers even) {

                this.even = even;

            }
```

```
        public bool next() {
            return true;
        }
        public int get() {
            this.index++;
            return this.even[this.index - 1];
        }
    }
}

void main() {
    var even = new EvenNumbers();
    stdout.printf("%d\n", even[5]);   // get()
    if (4 in even) {                  // contains()
        stdout.printf(@"$even\n");    // to_string()
    }
    foreach (int i in even) {         // iterator()
        stdout.printf("%d\n", i);
        if (i == 20) break;
    }
}
```

多线程编程（Multi-Threading）

Vala 中的线程（Threads in Vala）

用 Vala 编写的程序可以拥有多个线程，允许在同一时间做多个事情。至于线程具体是如何管理的，超出了 Vala 的语言范围。多个线程可能共享一个处理器，这取决于具体的运行环境。

Vala 中一个线程并不是在编译的时候定义的，取而代之的是用简单的一段代码申请一个线程来执行。这是通过使用 GLib 中 Thread 类的静态方法来完成的，就像如下简单的例子所示：

```
        void* thread_func() {
            stdout.printf("Thread running.\n");
            return null;
```

```
}

int main(string[] args) {
    if (!Thread.supported()) {
        stderr.printf("Cannot run without threads.\n");
        return 1;
    }
    try {
        Thread.create(thread_func, false);
    } catch (ThreadError e) {
        return 1;
    }
    return 0;
}
```

这个简短的程序创建并运行一个新的线程。代码在 thread_func 中执行。另外要注意的是，在 main 方法的开头所执行的测试，Vala 应用程序将不能使用线程，除非程序被适当的编译。因此，如果按照惯例来编译此程序，将会显示个错误信息，并终止运行。在运行时检查线程是否支持，能够使得不论有没有线程都可以编译运行程序。为了支持线程，可以这样编译程序：

```
$ valac --thread threading-sample.vala
```

以上操作将包含所有需要的库，并在线程可用的情况下，确保线程系统被初始化。

这时运行该程序将不会产生段错误，但是仍然没有达到我们需要的效果。如果没有任何形式的事件循环，当主线程（由“main”方法隐式创建）执行完的时候。Vala 程序将被终止，为了控制这种行为，可以允许线程间的协作。可以使用强大的事件循环或者异步队列来达到这个效果，但是目前只介绍基本的线程概念。

一个线程可以告知系统当前不需要执行，转而建议先让另外的线程运行，这是通过使用静态方法 Thread.yield() 来实现的。如果将此代码放在上面 main 方法的末尾，系统将暂停主线程的运行，并检查是否存在另外可以运行的线程，以上的例子就会转向运行新创建的线程，直到该线程运行完毕，整个程序才终止，这正是我们希望得到的效果。但是这并不能确保以上的执行流程，系统可以决定何时运行线程，可能的情况是新创建的线程可能还没有结束，主线程就被重新调度并运行结束并终止整个程序了。

为了能够等待一个线程的完成，可以用 join() 方法。对一个线程对象调用该方法，将导致调用

的线程等待直到该线程对象代表的线程结束运行。另外也允许一个线程去获得另外一个线程的返回值，如果这个返回值是有用的话。为了实现 join 线程：

```
try {
    unowned Thread thread = Thread.create(thread_func, true);
    thread.join();

} catch (ThreadError e) {
    return 1;
}
```

这一次，我们创建线程的时候为最后一个参数传递了 true。这样标记，使得该线程是可以被等待的（joinable）。我们也保存了创建所返回的 unowned 型 Theard 对象的引用（关于 unowned 引用将在后面的章节解释）。通过这个引用，我们可以在主线程中等待（join）创建的线程完成。这个版本的程序将确保新创建的线程在主线程继续运行并终止程序前得到充分运行。

以上所有的例子都存在一个潜在的问题，就是新创建的线程并不知道它该运行在怎么样的上下文里。在 C 编程中，可以为线程创建提供一些数据，而在 Vala 编程中经常是将一个实例的方法传递给 Thread.create，而不是用静态的方法。

资源控制（Resource Control）

当多个线程同时运行的时候，就有可能同时访问同一个数据。因此这种情况下，多个线程之间存在竞争条件，结果则取决于系统何时切换线程。

为了控制这种情况。可以使用 lock 关键字来确保一段访问相同数据的代码在执行时，不被其他线程所打断。给个例子来说明是最好不过了：

```
public class Test : GLib.Object {
    private int a { get; set; }
    public void action_1() {
        lock (a) {
            int tmp = a;
            tmp++;
            a = tmp;
        }
```

```
        }

        public void action_2() {
            lock (a) {
                int tmp = a;
                tmp--;
                a = tmp;
            }
        }
    }
```

以上的类定义了两个方法，都是用于改变 “a” 的值。如果这里不用 “lock” 关键字来括住这两个方法的代码段，将会允许这两个方法交替运行，而对 “a” 的改变结果将会是随机的。不过这里使用了 lock 代码段，因此 Vala 保证在一个线程锁住 “a” 的时候，另外一个需要得到该锁的线程将持续等待，直到该锁被销毁。

在 Vala 中能够在代码执行时锁定的只能是对象中的成员。这也是主要的使用限制，但是实际上，使用这种技术的标准方法是，所有在类中涉及到锁定资源的代码，都需要在类的内部用 lock 来锁定。就像上面的例子中所有对 “a” 的访问。

主循环（The Main Loop）

GLib 拥有一个被称作事件循环的系统，也就是 Vala 中的 MianLoop 类。这个系统的目的是允许编写一个能够等待事件发生并对相应事件（event）作出回应的程序，而无需不停地检查条件。这是被 GTK+ 所使用的事件驱动模型，因此程序可以等待用户的交互而不依赖当前运行的任何代码。

以下的程序创建并运行了一个 MainLoop，然后使之关联了一个事件源。此时，事件源只是一个简单的定时器，并且在 2000 毫秒后执行指定的方法。在这个例子中，该方法只是停止主循环，并终止程序的运行。

```
    void main() {
        var loop = new MainLoop();
        var time = new TimeoutSource(2000);
        time.set_callback(() => {
            stdout.printf("Time!\n");
```

```
            loop.quit();
            return false;
        });

        time.attach(loop.get_context());
        loop.run();
    }
```

当使用 GTK+ 编程时， 将会自动创建一个主循环（main loop），并且在调用 'Gtk.main()' 的时候会启动该循环。这也代表着此刻程序已经准备好运行，并且乐意接受用户或者其他地方发起的事件。使用 GTK+ 的代码和以上的例子类似，因此可以使用类似的方法来添加事件源（event sources），尽管必须使用 GTK+ 的方法来控制主循环（main loop）。

```
    void main(string[] args) {
        Gtk.init(ref args);
        var time = new TimeoutSource(2000);
        time.set_callback(() => {
            stdout.printf("Time!\n");
            Gtk.main_quit();
            return false;
        });
        time.attach(null);
        Gtk.main();
    }
```

通常，GUI 编程中的一个普遍需求是，在 GUI 中执行的动作要能被尽快的响应，同时又不会影响用户当前的使用。为了达到这个要求，可以使用 IdleSource，它发送事件到程序的主循环中，但是只有在没有更重要的事需要处理的时候才会响应这些请求（译者注：空闲状态）。

更多的关于事件循环的信息，请参阅 GLib 和 GTK+ 的文档。

异步方法（Asynchronous Methods）

所谓的异步方法是指那些在程序员的控制下，可以暂停并且恢复的方法。通常用于程序的主线程中

那些需要等待外部的慢任务完成的情况，但是又必须确保不会终止其他正在进行的处理（比如，一个很慢的操作不能冻结整个 GUI 的响应）。当这些方法需要等待某些数据的时候，将会主动交出 CPU 时间给调用者（比如 yields），当其等待的数据准备好时，将重新被调度。

异步方法可能需要等待的外部慢任务有：等待远端服务器的数据，或者等待另外一个线程的计算结果，或者等待从一个驱动器中读取数据完毕。

异步方法通常被用在 GLib 的主循环的中，比如通常用空闲回调（idle callbacks）来处理一些内部的回调。但是在特定的情况下，可以脱离 GLib 主循环来使用，比如该异步方法经常被 yield 并且从未调用 Idle.add() 方法。

异步方法被设计为使得多个不同的长时效操作可以一个线程里交错地运行。他们没有将自己的负载分散在不同的线程上。然而，异步方法却可以用来控制一个后台线程，并等待其完成，或者将后台线程操作序列化。

Vala 使用的异步方法是由 GIO 库来实现的，因此编译的时候必须使用 --pkg gio-2.0 选项。

使用 async 关键字来定义一个异步方法，比如：

```
async void display_jpeg(string fnam) {
    // Load JPEG in a background thread and display it when loaded
    [...]
}
```

或者：

```
async int fetch_webpage(string url, out string text) throws IOError {
    // Fetch a webpage asynchronously and when ready return the
    // HTTP status code and put the page contents in 'text'
    [...]
    text = result;
    return status;
}
```

可以看出，异步方法和普通的方法一样，可以拥有参数列表和返回值。也能用 yield 语句来显式地交出 CPU 时间给其调用者。

调用异步方法有如下三种形式：

```
display_jpeg("test.jpg");

display_jpeg.begin("test.jpg");

display_jpeg.begin("test.jpg", (obj, res) => {

    display_jpeg.end(res);

});
```

前两种形式是完全对等的，使用对应的参数来调用异步方法。第三种形式，则注册了一个 AsyncReadyCallback 类型的方法，该方法将在异步方法完成时被自动调用，其中 异步方法的 .end() 方法将包含异步方法的返回值及其异常等结束状态，可以利用此特性使用 .end() 方法来捕获异步方法抛出的错误或者接收异步方法的返回值。如果异步方法拥有被 out 修饰的参数。则可以在 .begin() 中省略，而在 .end() 调用中再使用该参数。

比如：

```
fetch_webpage.begin("http://www.example.com/", (obj, res) => {

    try {

        string text;

        var status = fetch_webpage.end(res, out text);

        // Result of call is in 'text' and 'status' ...

    } catch (IOError e) {

        // Problem ...

    }

});
```

当异步方法开始运行时，它将占用 CPU 时间直到运行到第一个 yield 语句，然后将 CPU 时间返给其调用者。之后当该方法被恢复时，将接着在 yield 后的代码运行。使用 yield 多种常见的途径有：

如下形式将放弃 CPU 时间，将在 GLib 主循环中没有其他事件需要处理的时候恢复方法的运行。

```
Idle.add(fetch_webpage.callback);

yield;
```

如下形式将放弃 CPU 控制权， 同时存储回调的详细内容，提供给那些想恢复该方法执行的代码使用。

```
SourceFunc callback = fetch_webpage.callback;
[... store 'callback' somewhere ...]
yield;
```

现下其他的代码只能调用被存储的 SourceFunc 来恢复该方法的执行。可以通过 Glib 的主循环调用来运行:

```
Idle.add((owned) callback);
```

或者在主线程中直接调用：

```
callback();
```

如果使用以上的形式直接调用， 被恢复的异步方法将立即接管 CPU 的控制权，在返回到 callback() 调用者前，一直运行到下一个 yield 语句。使用 Idle.add() 是非常有效的，如果回调必须是在后台线程中被调用，比如在一些后台的处理完成后再恢复异步方法。（ 其中 (owned) 转换不是必须的，在这里只是为了避免委托拷贝的编译警告信息。 ）

第三种形式一般多见于，一个异步方法使用 yield 来调用另外一个异步方法，比如：

```
yield display_jpeg(fnam);
```

或者：

```
var status = yield fetch_webpage(url, out text);
```

以上两种情况，均会主动放弃 CPU 控制权，并直到被调用的方法执行完成后才被恢复执行。其中 yield 语句将自动为被调用的方法注册回调，以确保调用者能够被正确恢复。该自动回调同样能够接收被调用方法的返回值。

当这样的 yield 语句被执行时，首先将 CPU 控制权交付给被调用的方法，直到被调用的方法运行到第一个 yield，接着放弃 CPU 控制权返回到调用者，然后调用者完成其自身的 yield 语句， 最终放弃 CPU 控制并返回到该调用者的调用者。

可以参阅 Async Method Samples 来深入学习异步方法的不同用法。

弱引用（Weak References）

Vala 的内存管理是基于自动的引用计数机制。每当一个对象被赋值给一个变量的时候，该对象内部的引用计数将自动加一，而每当引用一个对象的变量的生存周期结束的时候，该对象内部的引用计数则自动减一。当对象的引用计数值为零时，该对象将被系统自动销毁（delete）。

不过，此机制可能会使数据结构表现出循环引用的情况。比如，一个树数据结构中的孩子节点可能会保持一个其父亲节点的引用，反之亦然，父亲节点需要保持一个其孩子节点的引用，或者一个双向链表数据结构，其中每个元素都保持其前置元素的一个引用，其前置元素也保持对该元素的一个引用。

在这些情形下，由于相互引用的关系，对象的生存周期将始终被保持，即便该对象在某时应该被销毁了。为了打断这样的引用循环形式，可以使用 weak 修饰符来修饰其引用：

```
class Node {
    public weak Node prev;
    public Node next;
}
```

有关此主题的细节在 Vala's Memory Management 中有阐述。

所有权（Ownership）

无属引用（Unowned References）

通常，在 Vala 中创建一个对象，返回的是对该对象的引用。这也就意味着传递了一个指向该对象在内存中地址的指针，该指针会被该对象所自动登记。同样，不论何时，其他对该对象创建的引用，也会被对象所登记。正因为对象能够明确知道对自身的所有引用，所以可以在需要的时候自动移除这些引用，以上是 Vala 内存管理的基本概念。

反之，无属引用（Unowned references）将不被该对象所登记。这样做允许在逻辑需要的时候将直接对象移除，不论此时是否仍存在对该对象的引用。通常这样做的方式是，定义一个返回无属引用的方法：

```
class Test {
    private Object o;
```

```
    public unowned Object get_unowned_ref() {
        this.o = new Object();
        return this.o;
    }
}
```

当调用此方法时，为了能够接收该方法返回的引用，那么你一定希望用无属引用类型来接收该无属引用：

```
unowned Object o = get_unowned_ref();
```

但是要注意某些情况可能并非如你所愿，使得以上例子过于复杂的原因是所有权的概念：

- 如果对象 " o" 没有在类中存储，那么当方法 "get_unowned_ref" 返回后，“o”将变为无属的（将没有任何对该对象的引用）。如果是这样的情况，该对象将被销毁，而该方法将永远不能返回一个有效的引用。

- 如果返回值没有被定义为无属型，对象所有权将被传递给调用者代码。但是调用者却希望接受一个无属型的引用，因此也无法获得所有权。

如果以如下形式调用方法：

```
Object o = get_unowned_ref();
```

此时 Vala 将尝试获得一个强引用或者拷贝该无属引用指向的实例。

对比普通的方法，属性的 get 方法始终返回无属的返回值。这意味着，你不能在 get 方法中返回在该方法内部新创建的对象，同样也不能在方法调用中返回有属的（owned）返回值。事实上，属性的所有权是被拥有该属性的对象所拥有。一个在对象侧获取属性值的调用不能窃取（steal）或者转交所有权（reproduce）（通过使用拷贝或者增加其引用计数方式）该属性值。

因此，下面的例子将导致编译错误：

```
public Object property {
    get {
        return new Object();
        // WRONG: property returns an unowned reference,
```

```
        // the newly created object will be deleted when
        // the getter scope ends the caller of the getter ends up
        // receiving an invalid reference to a deleted object.
    }
}
```

同样也不能这样做：

```
public string property {
    get {
        return getter_method();   // WRONG: for the same reason above.
    }
}

public string getter_method() {
    return "some text"; // "some text" is duplicated and returned at this
                        // point.
}
```

如下，是最好的方式：

```
public string property {
    get {
        return getter_method(); // GOOD: getter_method returns an unowned value
    }
}

public unowned string getter_method() {
    return "some text";
    // Don't be alarmed that the text is not assigned to any strong variable.
    // Literal strings in Vala are always owned by the program module itself,
    // and exist as long as the module is in memory
}
```

无属修饰符可以用于自动的无属属性存储，这意味着：

```
public unowned Object property { get; private set; }
```

等同于：

```
private unowned Object _property;

public Object property {
    get { return _property; }
}
```

关键字 owned 可以明确要求某个属性返回有属引用，由此将导致属性值所有权在对象侧被转交。试想将之前的例子中的 unowned 换成两个 owned，那么还是一般的属性或者简单的 get_xxx 方法么？这肯定会对你的程序设计带来问题，但不管怎么样，以下的代码段是正确的：

```
public Object property { owned get { return new Object(); } }
```

无属引用和后面将要描述的指针发挥着类似的作用。然而，它们比指针更加容易使用，因为他们可以简单地转换为普通的有属引用。但是在编程过程中，通常它们不被广泛地使用，除非你明确得知道你在做什么。

所有权转让（Ownership Transfer）

关键字 owned 是用于所有权转让的。

- 作为一个参数类型的前缀，意味着该对象的所有权被转让给该方法代码上下文。
- 作为类型转换操作符，可以避免拷贝在 Vala 中经常遇到的非引用计数的类，比如：

```
Foo foo = (owned) bar;
```

这意味着，bar 将被置为 null， 而 foo 将继承其引用/所有权。

可变长度的参数列表（Variable-Length Argument Lists）

Vala 的方法支持 C 风格的可变长度参数列表（“varargs”）。在方法中使用省略号（“...”）来声明这种类型的参数，在提供可变长的参数列表时，要求至少有一个固定的参数：

```
void method_with_varargs(int x, ...) {
```

```
        var l = va_list();
        string s = l.arg();
        int i = l.arg();
        stdout.printf("%s: %d\n", s, i);
    }
```

在这个例子中，参数 x 是个固定参数，使得符合以上提到的要求。通过方法 va_list() 来获得参数列表，然后通过调用该列表的方法 arg<T> 来顺序地一个接一个从该参数列表中获得参数，其中 T 是列表中的参数将被转换为的类型。如果参数类型在上下文中是显而易见的（比如我们的例子中），参数类型将被自动推断，因此只需要调用 arg() 方法，而无需使用泛型参数。

以下的例子将解析任意数量的字符串（成对的字符串参数）：

```
    void method_with_varargs(int fixed, ...) {
        var l = va_list();
        while (true) {
            string? key = l.arg();
            if (key == null) {
                break;  // end of the list
            }
            double val = l.arg();
            stdout.printf("%s: %g\n", key, val);
        }
    }

    void main() {
        method_with_varargs(42, "foo", 0.75, "bar", 0.25, "baz", 0.32);
    }
```

例子中检查返回 null 来作为是否到达参数列表末尾的标记。Vala 始终隐式地将 null 作为参数列表中的最后一个参数。

可变长参数列表（Varargs）有个需要关注的严重缺陷：它不是类型安全的。编译器并不能告知你传递的参数类型是否合法。这也是为什么除非有特别理由的情况下，才建议使用该特性。比如：将你的 Vala 库提供给 C 程序员， 绑定 C 函数。通常，一个参数数组，是更好的选择。

可变长参数列表（varargs）的一个常见模式是，预期的是一个参数值配对的交替字符串，通常是 gobject 的“属性 – 值”字符串形式。在这种情况下，你可以用 property: value 形式来取代之。

比如：

```
    actor.animate (AnimationMode.EASE_OUT_BOUNCE, 3000, x: 100.0, y: 200.0,
rotation_angle_z: 500.0, opacity: 0);
```

以上等同于：

```
    actor.animate (AnimationMode.EASE_OUT_BOUNCE, 3000, "x", 100.0, "y", 200.0,
"rotation-angle-z", 500.0, "opacity", 0);
```

指针（Pointers）

指针是在 Vala 编程中允许手动管理内存的途径。通常，当你创建了一个类型的实例，并接收了该实例的引用，Vala 将在该实例没有任何引用的时候销毁它。通过请求获得一个实例的指针，则有责任在该实例不再需要的时候销毁它，并且因此对于内存的使用获得了更大权限的控制。

在绝大多数时候并没必要使用此功能，现代的计算机系统对于处理引用计数来说已经足够快了，并且也拥有足够大的内存，因而细微的效率影响并不重要。

你可能采取手动管理内存的情形有：

- 当特别需要优化一段程序代码的时候。
- 当处理一个外部的库，而该库不是基于 gobject 实现，没有引用计数机制的时候

为了在创建一个实例的时候，并获得其指针：

```
    Object* o = new Object();
```

为了访问指针指向的实例成员：

```
    o->method_1();
    o->data_1;
```

为了销毁指针指向的实例：

```
delete o;
```

Vala 同样支持 C 语言中取地址（&）和间接引用（*）操作符：

```
int i = 42;
int* i_ptr = &i;    // address-of
int j = *i_ptr;     // indirection
```

其行为和引用类型有一个细微的差别，对于引用类型，在赋值的时候可以忽略取地址和间接引用操作符：

```
Foo f = new Foo();
Foo* f_ptr = f;    // address-of
Foo g = f_ptr;     // indirection

unowned Foo f_weak = f;  // equivalent to the second line
```

而对于无属引用类型，引用型或者指针的用法是没有区别的。

非对象类（Non-Object classes）

对待那些不是继承自 Glib.Object 的类，是个特别情况，它们直接衍生自 GLib 的类型系统，因此是非常轻量级的类。在新版本的 Vala 编译器下，这种类也可以实现接口，信号还有属性。

很明显，在 GLib 的绑定中使用了这种非对象类，因为 GLib 的层次比 GObject 要低，许多在 GLib 绑定中定义的类都是此种类型。正如前面提到的，这种轻量级的类在许多实际情况下非常有用（比如 Vala 编译器本身）。不过，此种类的详细用法已经超出了本手册的范围。只要注意到，这种类和结构体相比，本质上是不同的。

D-BUS 集成（D-Bus Integration）

D-Bus 以非常紧凑的形式集成在 Vala 语言中，没有比在 Vala 中更加容易使用 D-Bus 的方式了。

如果想导出一个自己的类作为 D-Bus 服务，你只需要用 DBus 代码属性来修饰此类，然后在你本地的 D-Bus 会话中注册一个类的实例。

```
        [DBus(name = "org.example.DemoService")]

        public class DemoService : Object {

            /* Private field, not exported via D-Bus */

            int counter;


            /* Public field, not exported via D-Bus */

            public int status;


            /* Public property, exported via D-Bus */

            public int something { get; set; }


            /* Public signal, exported via D-Bus

             * Can be emitted on the server side and can be connected to on the client
side.

            */

            public signal void sig1();


            /* Public method, exported via D-Bus */

            public void some_method() {

                counter++;

                stdout.printf("heureka! counter = %d\n", counter);

                sig1();  // emit signal

            }


            /* Public method, exported via D-Bus and showing the sender who is

            is calling the method (not exported in the D-Bus inteface) */

            public void some_method_sender(string message, GLib.BusName sender) {

                counter++;

                stdout.printf("heureka! counter = %d, '%s' message from sender %s\n",

                counter, message, sender);

            }

        }
```

注册一个服务的实例然后运行主循环：

```
void on_bus_aquired (DBusConnection conn) {
    try {
        // start service and register it as dbus object
        var service = new DemoService();
        conn.register_object ("/org/example/demo", service);
    } catch (IOError e) {
        stderr.printf ("Could not register service: %s\n", e.message);
    }
}

void main () {
    // try to register service name in session bus
    Bus.own_name (BusType.SESSION, "org.example.DemoService",
    /* name to register */
    BusNameOwnerFlags.NONE, /* flags */
    on_bus_aquired, /* callback function on registration succeded */
    () => {}, /* callback on name register succeded */
    /* callback on name lost */
    () => stderr.printf ("Could not aquire name\n"));

    // start main loop
    new MainLoop ().run ();
}
```

编译此程序必须要附加上 gio-2.0 包：

```
$ valac --pkg gio-2.0 dbus-demo-service.vala
$ ./dbus-demo-service
```

所有小写的字和下划线交替的成员名称将被自动转换为 D-Bus 的驼峰风格名称。在这个例子中被导出的 D-Bus 接口有个一名为 Something 的属性， 一个名为 Sig1 的信号还有一个名为 SomeMethod 的方法。你可以打开一个新的控制台窗口，输入如下命令调用该方法：

```
$ dbus-send --type=method_call                    \
```

```
                --dest=org.example.DemoService       \
                /org/example/demo                    \
                org.example.DemoService.SomeMethod
```

或者

```
    $ dbus-send --type=method_call                   \
                --dest=org.example.DemoService       \
                /org/example/demo                    \
                org.example.DemoService.SomeMethodSender \
                string:'hello world'
```

你同样可以使用图形化的 D-Bus 调试器，比如 D-Feet 来浏览你的 D-Bus 接口并调用其中的方法。

更加全面的例子，参见：Vala/DBusClientSamples 和 Vala/DBusServerSample

配置（Profiles）

Vala 支持几种不同的配置：

- gobject（默认）
- posix
- dova

配置决定了可用的语言特性以及生成的 C 代码依赖的库文件。

想使用一个不同的配置，可以使用 valac 编译器的 --profile 选项，比如：

```
    valac --profile=posix somecode.vala
```

实验性的功能（Experimental Features）

某些 Vala 的功能仍处在实验阶段。也就是说这些功能没有经过完成的测试并且可能在将来的版本做出修改。

连锁关系表达式（Chained Relational Expressions）

这种功能使得你编写如下复杂关系的代码时：

```
if (1 < a && a < 5) { }

if (0 < a && a < b && b < c && c < d && d < 255) {
    // do something
}
```

可以用更自然的形式编写：

```
if (1 < a < 5) {}
if (0 < a < b < c < d < 255) {
    // do something
}
```

正则表达式（Regular Expression Literals）

Regular expressions 是在用模式匹配字符串文字时使用的一种强大技术。Vala 目前为正则表达式提供了实验性的支持（/regex/）：

```
string email = "tux@kernel.org";
if (/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.match(email)) {
    stdout.printf("Valid email address\n");
}
```

其中尾随的 i 使得该表达式是不区分大小写的，可以将一个正则表达式保存在一个类型为 Regex 的变量中：

```
Regex regex = /foo/;
```

关于用正则表达式替换字符串的例子：

```
var r = /(foo|bar|cow)/;
```

```
var o = r.replace ("this foo is great", -1, 0, "thing");
print ("%s\n", o);
```

有以下几种尾随字符可用：

- i，使得模式匹配不区分字母的大小写
- m，使得模式中的行开始或者行结束位置元字符对字符串中每个行开始和行结束都立即生效，而不是只对整个字符串做行开始和行结束的鉴定。
- s，使得模式中的点元字符（.）将匹配包括换行符在内的所有字符。如果不用此尾随字符，换行符默认是被自动排除的。
- x，模式中所有的空白字符将被忽略，除了被转义的或者在字符类中的字符。

针对以上几种尾随字符的用法，本人添加了几个例子（原文中没有）：

```
string str_test = “the first line!\nthe second line!”;

if(/the First line!/i.match(str_test)) {
    // it should be run here
}

var result = /the first line!$/.replace(str_test, -1, 0, "replacement");

stdout.printf (“result is %s\n”); // is original string

var result = /the first line!$/m.replace(str_test, -1, 0, "replacement");

stdout.printf (“result is %s\n”); // result is “replacement\nthe second
                                  line!”
if (/.+the second line!/.match(str_test)) {
    // it should not be run here
}

if (/.+the second line!/s.match(str_test)) {
    //it should be run here
}

if (/the second line!/x.match(str_test)) {
    //it just to match thesecondline!
```

```
}
```

严格的非空模式（Strict Non-Null Mode）

如果在编译 Vala 代码的时候使用了 --enable-experimental-non-null 选项，编译器将 运行在严格的非空检查模式下，编译器将认为所有的类型都是非空的，除非某类型被明确的用问号（？）标记来声明。

```
Object o1 = new Object();     // not nullable
Object? o2 = new Object();    // nullable
```

编译器将执行一个运行时的静态分析，以确保没有可为空的引用被赋值给不可为空的引用，比如以下的赋值是不允许的：

```
o1 = o2;
```

其中 o2 是可以为 null 的，但是 o1 被声明为不可为 null 的，所以这样的赋值是被禁止的。不过，可以使用明确的非空转换来改变这样的行为，如果可以确认 o2 是不为空的话：

```
o1 = (!) o2;
```

严格非空模式可以帮助编程过程避免引用空引用（null）的错误，这个功能非常有潜力，如果所有的绑定都对返回值空的可能性做了正确标记的话，遗憾的是这种约定目前还没完全被贯彻执行。

库（Libraries）

在系统级，Vala 的库本质上就是 C 库，因此都使用相同的工具。为了使过程更加简单，有个额外的具体信息用来让 Vala 编译器理解这个过程。

因此，Vala 库是系统的一部分：

- A system library (e.g. libgee.so)
- A pkg-config entry (e.g. gee-1.0.pc)

可见都是安装在本地的标准目录中。而 Vala 的具体文件则是：

- A VAPI file (e.g. gee-1.0.vapi)
- An optional dependency file (e.g. gee-1.0.deps)

这些文件将在本章节的后面解释。不过需要注意的是：库文件名，Vala 的具体文件名还有 pkg-config 文件名都是一致的。

库的使用（Using Libraries）

如果你使用 Valac 编译器，那么使用一个库基本是自动化的。Vala 的具体库文件涵盖了关于一个包的所有信息。像如下代码所示，告诉编译器你所需要的包：

```
$ valac --pkg gee-1.0 test.vala
```

以上命令意味着你可以使用 gee-1.0.vapi 文件中定义的一切类型，并且包括了一切 gee-1.0 所依赖的类型。如果确实存在依赖，那么可以在 gee-1.0.deps 文件中罗列出来。在这个例子中，valac 是编译二进制文件的一组方法，并且因此会从相关的 pkg-config 文件中合并信息来链接正确的库。这也是为什么 pkg-config 文件名和 Vala 库文件名一致的原因。

包一般也使用了命名空间，但是和 Vala 中的命名空间无技术上的关联。也就是说，即便你的应用程序编译的时候引用了某个包，但是代码中依然要在每个文件中每个合适的地方加上 using 代码段，或者为所有符号使用完全扩展的名称。

同样也可以将本地的库（指没有安装在系统中的库）作为一个包来使用。比如，Vala 本身就使用了内部版本的 Gee 库。当 valac 构造 Gee 并创建 VAPI 文件的时候，是大致这样使用的：

```
$ valac --vapidir ../gee --pkg gee ...
```

关于如何生成这个库的细节，请参阅下一章节或例子。

库的制作（Creating a Library）

使用 Autotools（Using Autotools）

可以使用 Autotools 来创建那些由 Vala 编写的库。库是由 Vala 编译器生成的 C 代码来创建的，然后像其他库一样的链接和安装。接着你需要告知哪些 C 文件是必须被用作创建库的，哪些又是必须被分派的（distributable），从而允许其他人编译成一个压缩包。使用标准的 Autotools 命

令：configure，make 和 make instal 。

例子（Example）

该例子源于新增的 GXml 项目。 GXmlDom 是一个基于 GObject 实现的库，旨在替代 libxml2；是由 Vala 所编写，并且最初是用 WAF 来构建的。

使用 valac 来生成对应 Vala 源码文件的 C 代码文件和头文件。目前，也已经可以由 Vala 源码来生成 GobjectIntrospection 和 VAPI 文件。

gxml.vala.stamp 用来作为我们库中的源代码文件。

为了给 valac 设置所有编译和链接需要的 CFLAGS 和 LIBS 选项并使得最终能够编译成功，增加 --pkg 选项是非常重要的。

```
NULL =

AM_CPPFLAGS = \
            -DPACKAGE_LOCALE_DIR=\""$(prefix)/$(DATADIRNAME)/locale"\" \
            -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
            -DPACKAGE_DATA_DIR=\""$(datadir)"\"

BUILT_SOURCES = gxml.vala.stamp
CLEANFILES = gxml.vala.stamp

AM_CFLAGS =\
        -Wall\
        -g \
        $(GLIB_CFLAGS) \
        $(LIBXML_CFLAGS) \
        $(GIO_CFLAGS) \
        $(GEE_CFLAGS) \
        $(VALA_CFLAGS) \
        $(NULL)

lib_LTLIBRARIES = libgxml.la
```

```
VALAFLAGS = \
        $(top_srcdir)/vapi/config.vapi \
        --vapidir=$(top_srcdir)/vapi \
        --pkg libxml-2.0 \
        --pkg gee-1.0 \
        --pkg gobject-2.0 \
        --pkg gio-2.0 \
        $(NULL)

libgxml_la_VALASOURCES = \
                    Attr.vala \
                    BackedNode.vala \
                    CDATASection.vala \
                    CharacterData.vala \
                    Comment.vala \
                    Document.vala \
                    DocumentFragment.vala \
                    DocumentType.vala \
                    DomError.vala \
                    Element.vala \
                    Entity.vala \
                    EntityReference.vala \
                    Implementation.vala \
                    NamespaceAttr.vala \
                    NodeList.vala \
                    NodeType.vala \
                    Notation.vala \
                    ProcessingInstruction.vala \
                    Text.vala \
                    XNode.vala \
```

```
                    $(NULL)

libgxml_la_SOURCES = \
                gxml.vala.stamp \
                $(libgxml_la_VALASOURCES:.vala=.c) \
                $(NULL)

# Generate C code and headers, including GObject Introspection GIR files and
  VAPI file

gxml-1.0.vapi gxml.vala.stamp GXml-1.0.gir: $(libgxml_la_VALASOURCES)
$(VALA_COMPILER) $(VALAFLAGS) -C -H $(top_builddir)/gxml/gxml-dom.h \
                    --gir=GXmlDom-1.0.gir  --library gxmldom-1.0 $$
@touch $@


# Library configuration
libgxml_la_LDFLAGS =

libgxml_la_LIBADD = \
                $(GLIB_LIBS) \
                $(LIBXML_LIBS) \
                $(GIO_LIBS) \
                $(GEE_LIBS) \
                $(VALA_LIBS) \
                $(NULL)

include_HEADERS = \
                gxml.h \
                $(NULL)

pkgconfigdir = $(libdir)/pkgconfig
pkgconfig_DATA = libgxml-1.0.pc

gxmlincludedir=$(includedir)/libgxml-1.0/gxml
gxmlinclude_HEADERS= gxml-dom.h

# GObject Introspection
```

```
if ENABLE_GI_SYSTEM_INSTALL
    girdir = $(INTROSPECTION_GIRDIR)
    typelibsdir = $(INTROSPECTION_TYPELIBDIR)
else
    girdir = $(datadir)/gir-1.0
    typelibsdir = $(libdir)/girepository-1.0
endif

# GIR files are generated automatically by Valac so is not necessary to scan
  source code to generate it

INTROSPECTION_GIRS =
INTROSPECTION_GIRS += GXmlDom-1.0.gir
INTROSPECTION_COMPILER_ARGS = \
                            --includedir=. \
                            --includedir=$(top_builddir)/gxml

GXmlDom-1.0.typelib: $(INTROSPECTION_GIRS) \
    $(INTROSPECTION_COMPILER) $(INTROSPECTION_COMPILER_ARGS)  $< -o $@

gir_DATA = $(INTROSPECTION_GIRS)
typelibs_DATA = GXmlDom-1.0.typelib

vapidir = $(VALA_VAPIDIR)
vapi_DATA=gxmldom-1.0.vapi

CLEANFILES += $(INTROSPECTION_GIRS) $(typelibs_DATA) gxml-1.0.vapi

EXTRA_DIST = \
            libgxml-1.0.pc.in \
            $(libgxml_la_VALASOURCES) \
            $(typelibs_DATA) \
            $(INTROSPECTION_GIRS) \
            gxml.vala.stamp
```

使用命令行来汇编和链接（Compilation and linking using Command Line）

Vala 目前还不能直接地创建动态或静态库。为了创建一个库，通过使用 -c （仅编译）选项然后使用自己偏好的连接器来链接目标文件，比如 libtool 或者 ar 。

```
$ valac -c ...(source files)
$ ar cx ...(object files)
```

或者使用 gcc 来汇编 C 中间代码：

```
$ valac -C ...(source files)
$ gcc -o my-best-library.so --shared -fPIC ...(compiled C code files)...
```

例子（Example）

以下的例子是如何用 Vala 编写一个简单的库，并且在没有安装到系统的前提下，在本地编译和测试：

保存以下代码到一个名为 test.vala 的文件中。这是一个活生生的库源码， 包含了在我们主程序中 需要调用的功能：

```
public class MyLib : Object {
    public void hello() {
        stdout.printf("Hello World, MyLib\n");
    }
    public int sum(int x, int y) {
        return x + y;
    }
}
```

使用以下的命令来生成 test.c test.h 和 test.vapi 文件。其中前两个是编译 C 版本库需要的文件，而 VAPI 文件则描述了库提供的公共接口（public interface）。

```
$ valac -C -H test.h --library test test.vala --basedir ./
```

现在，让我们编译生成库：

```
$ gcc --shared -fPIC -o libtest.so $(pkg-config --cflags --libs gobject-2.0) \
test.c
```

保存以下代码到一个名为 hello.vala 的文件中，这段代码将使用我们已经创建的库：

```
void main() {
```

```
    var test = new MyLib();
    test.hello();
    int x = 4, y = 5;
    stdout.printf("The sum of %d and %d is %d\n", x, y, test.sum(x, \
                  y));
}
```

现在让我们编译应用程序代码，并告诉编译器链接由我们刚刚创建的库：

```
$ valac -X -I. -X -L. -X -ltest -o hello hello.vala test.vapi --basedir ./
```

现在我们可以运行该程序了。如下命令行指出了在本地目录中寻找任意需要的库：

```
$ LD_LIBRARY_PATH=$PWD ./hello
```

程序的输出结果如下：

```
Hello World, MyLib
The sum of 4 and 5 is 9
```

你也可以使用 `--gir` 选项来为你的库创建一个 GObjectIntrospection GIR 文件：

```
$ valac -C test.vala --library test --gir Test-1.0.gir
```

GIR 文件则是相关 API 的 XML 形式描述。

使用 VAPI 文件构建库（Binding Libraries with VAPI Files）

VAPI 文件是对外部 Vala 库中的公共接口（public interface）的描述。当使用 Vala 编写库的时候，该文件由编译器创建，基本上是所有 Vala 源码中公共定义的集合。当用于由 C 编写的库时， VAPI 文件将有点复杂，尤其是该库中的命名没有参照 GLib 的命名规则。在这种情况下，VAPI 文件包含了许多如何将标准的 Vala 接口定义映射到 C 版本的描述注解。

此过程一般主要有三个步骤：

- 运行 vala-gen-introspect 从 C 库中提取元数据。
- 添加额外的元数据来标准化接口或者各种其他变化。

- 使用 vapigen 从以上的源码中生成对应的 VAPI 文件。

关于如何生成绑定的具体说明，请参阅 *Vala Bindings Tutorial*。

工具（Tools）

Vala 包含了一些程序来帮助你构建 Vala 应用程序。关于每个工具的更加详细的介绍，请参阅 man pages。

valac

valac 是 Vala 的编译器。其主要的功能就是将 Vala 代码转换为编译的 C 代码，虽然在简单的情况下，它也可以编译和链接整个项目。

这样使用的简单例子是：

```
$ valac -o appname --pkg gee-1.0 file_name_1.vala file_name_2.vala
```

其中 -o 选项请求生成目标文件，而不是输出 C 源码文件。--pkg 选项则说明构建需要 gee-1.0 包中的信息。你不需要指定关于需要链接的库的细节，包的内部将包含此细节。最后，给出了所有源码的列表。如果你需要更加复杂的构建过程，使用 -C 选项来生成 C 代码文件替代生成二进制文件， 然后手动或者依托脚本来继续处理。

Vapigen

vapigen 是一个用于生成绑定的工具。其根据库的元数据和其他所需要的外部信息来生成 VAPI 文件。同样请参阅 *Vala Bindings Tutorial*。

Vala-gen-introspect

Vala-gen-introspect 是一个用于从基于 Gobject 库中提取元信息的工具。如今，执行该过程的首选方法是使用 GobjectIntrospection 来代替，而作为 vapigen 可以直接使用 GIR 文件。同样请参阅 *Vala Bindings Tutorial*。

其他技术（Techniques）

调试（Debugging）

出于演示的目的，我们将故意创建一个含有 Bug 的程序（buggy program），该程序将引用一个空的引用，因此将会导致一个段错误：

```
class Foo : Object {
    public int field;
}

void main() {
    Foo? foo = null;
    stdout.printf("%d\n", foo.field);
}
```

```
$ valac debug-demo.vala
$ ./debug-demo
Segmentation fault
```

那么我们如何调试该程序呢？ -g 命令行选项将告诉 Vala 编译器在最终的编译生成的二进制文件中包含源代码的行信息，--save-temps 则将指示编译器保存临时的 C 源码文件：

```
$ valac -g --save-temps debug-demo.vala
```

Vala 程序员可以使用 GNU 调试器 gdb 或者其图形化的一个前端工具 Nemiver 来调试该程序。

```
$ nemiver debug-demo
```

一个 GDB 会话的示例：

```
$ gdb debug-demo
(gdb) run
Starting program: /home/valacoder/debug-demo

Program received signal SIGSEGV, Segmentation fault.
0x0804881f in _main () at debug-demo.vala:7
7           stdout.printf("%d\n", foo.field);
(gdb)
```

使用 GLib（Using Glib）

Glib 包含了一组大量的工具，包含了绝大多数标准 C 函数实现的封装。这些工具可以用于所有 Vala 的平台上，即便是那些不兼容 POSIX 的。可以参阅 *GLib Reference Manual* 来获得对 GLib 功能的完整描述。其中的参考是基于 GLib 的 C API 形式，但是关于如何转换为 Vala 的工作方式是非常简单的。

通过以下命名规则来使用 GLib 的函数：

```
C API             Vala                 Example

g_topic_foobar()  GLib.Topic.foobar()  GLib.Path.get_basename()
```

类似的，可以如下使用 GLib 中的类型：

```
Instantiate with                Call an object member with

GLib.Foo foo = new GLib.Foo();  foo.bar();
```

C 和 Vala 之间的 API 并不相同，但是这样的命名规则意味着你可以在 Glib 的 VAPI 文件中找到你可以用于 Vala 的函数，以及对应的参数。希望这能够满足目前的需要直到更多关于 Vala 的文档被制作出来。

文件处理（File Handling）

关于如何灵活地使用文件 I/O 并处理文件，请参阅 Vala/GIOSamples 。

你也可以使用 FileUtils.get_contents 来读取一个文件内容并将其转换为一个字符串类型（string）。

```
string content;
FileUtils.get_contents("file.vala", out content);
```