Vala 内存管理

Vala 的内存管理基于自动的引用计数而非垃圾回收机制。

采用引用计数机制，好处和坏处是并存的，具体可以参考维基。引用计数通常情况是确定的，但是在某些情况下，你可能会造成引用循环。此时，你必须使用弱引用来打断引用循环。在 Vala 的语法里，修饰弱引用的关键字是 weak。

下面将介绍引用计数是如何工作的：

每当一个对象的引用赋值给一个变量时，该对象的引用计数将自动加 1(g_object_ref)，而当该变量的生存周期结束时，该变量引用的对象的引用计数将自动减 1(g_object_unref)。

当某个对象的引用计数为 0 时，将自动销毁该对象，如果该对象的某些成员也引用了其他对象，则对应对象的引用计数也将减 1，当对应的对象引用计数为 0 时，该对象也将被销毁。

那么什么是引用循环，引用循环存在什么问题呢？让我们看一个简单的双向链表例子：

```
class Node : Object {
    public Node prev;
    public Node next;
    public Node (Node? prev = null) {
        this.prev = prev;      // ref
        if (prev != null) {
            prev.next = this;  // ref
        }
    }
}
```

```
void main () {
    var n1 = new Node ();    // ref
    var n2 = new Node (n1);  // ref
    // print the reference count of both objects
    stdout.printf ("%u, %u\n", n1.ref_count, n2.ref_count);
}   // unref, unref
```

运行程序，然后打开任务管理器，你将看到，该程序将会消耗系统的所有内存（内存泄漏），在该程序将你的系统拖慢之前用任务管理器结束该程序的运行，消耗的内存被立即销毁。

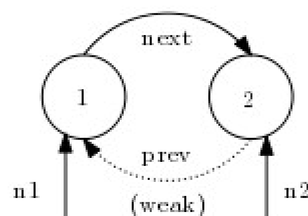
如果用 C# 或者 JAVA，一般不会有类似的问题，因为其包含的垃圾回收器在程序运行时能够检测到类似的引用循环。但是如果是使用 Vala，则必须使用一些策略，来避免这样的引用循环。

我们可以标记一个弱引用来打断这样的引用循环：

```
public weak Node prev;
public Node next;
```

这意味着，如果一个对象被赋值给用 weak 修饰符修饰的变量的时候，该对象的引用计数将不会自动加 1。

这时情况将会如下图所示：

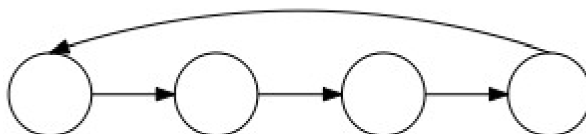


其中 n1 的引用计数是 1 了，不再像之前那样是 2 了。当 n1 和 n2 的生存周期结束的时候，n1 和 n2 的引用计数将自动减 1。接着，n1 的引用计数为 0，n1 将被自动销毁。而此时

n2 的引用计数为 1，由于 n1 中的成员包含了对 n2 的引用，但由于 n1 被销毁了，所以 n2 的引用计数将再自动减 1，所以接下来 n2 也被自动销毁了。

继续编译程序运行，你将观察到，该程序的内存使用将保持在一个稳定的数值。

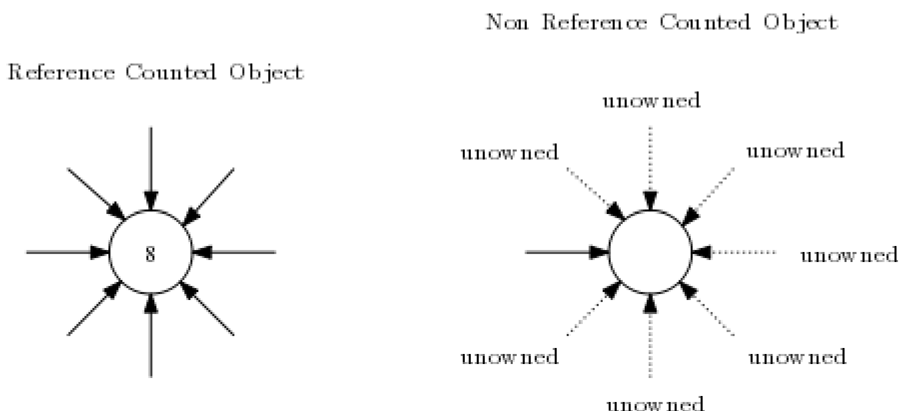
注意：引用循环并不一定是以直接引用的形式存在，比如如下的引用循环：



无属引用（Unowned References）

所有 Vala 中的对象和基于 gobject 实现的库中的对象都是使用引用计数来管理内存的。不过 Vala 同样允许你使用那些没有基于 gobject 同时也没有引用计数机制的 C 库中的类，这些类被称作 compact 类（被 [Compact] 属性所修饰）。

没有引用计数的对象只拥有一个强引用(strong reference，可以想象为 "owning" 引用)。当拥有该对象强引用的变量生存周期结束的时候，该对象将自动被销毁。因此引用其对象的其他引用，必须是无属引用类型，从而确保这些类型的引用变量生存周期结束时，其引用的对象不会被销毁。



因此，当你调用一个方法（比如没有基于 gobject 库）并返回一个无属引用型（被用 unownd 关键字标记的方法返回值）对象的时候，你有两种选择：比如新建一个对象，并拷贝返回的对象，假如有拷贝方法的话，这样你将独自有这个新对象的强引用；或者将返回的对象赋值给用 unowned 关键字声明的变量，这样 Vala 将知道在你的相关代码中不用销毁该对象。

Vala 会阻止你将一个无属引用赋值给强引用（比如非无属引用）。但是，你可以通过 owned 转换将无属引用对象的所有权转交给另外的一个引用：

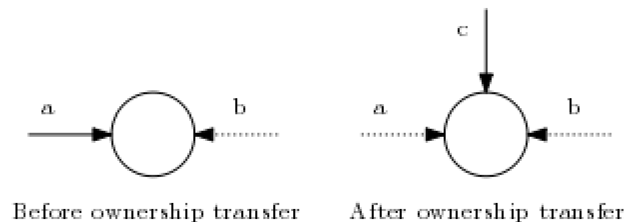
```
[Compact]
class Foo { }

void main () {

    Foo a = new Foo ();

    unowned Foo b = a;

    Foo c = (owned) a;   // 'c' is now the new "owning" reference
}
```



顺便提一下，Vala 的字符串类型（strings）也不是基于引用计数的。因为它是基于 C 的 char * 类型。不过，在需要的时候，Vala 会很小心的对字符串做自动拷贝的动作，所以完全不必多虑。只有语言绑定的编写者才需要关心在 API 中的字符串是否被合适的标记为 unowned 型。

当前，许多 Vala API 的绑定都是用 weak 标记的，即便实际上的类型是 unowned。这是因为在早些时候， weak 关键字兼具了两者的功能，所以不要被迷惑。目前 weak 和 unowned 可以互换使用，不过最好只是使用 weak 来打断引用循环，而用 unowned 来解决上面描述的所有权问题。

内存管理的几个例子（**Memory Management by Example**）

基于一般的引用计数机制的内存管理

```
public class Foo {
      public void method () { }
}

void main () {

      Foo foo = new Foo ();    // allocate, ref
      foo.method ();
      Foo bar = foo;           // ref

}  // unref, unref => free
```

以上一切的过程都是自动进行的。

基于指针语法的手动内存管理

你可以坚持选择自己来手动管理内存，如果你觉得你必须拥有完全的控制的话。指针语法借鉴了 C++：

```
void main () {

      Foo* foo = new Foo ();   // allocate
      foo->method ();
      Foo* bar = foo;
      delete foo;              // free
}
```

基于 Compact 类的非引用计数机制的内存管理

Compact 类是指那些没有在 Vala 的类型系统中注册的类。这些类通常来自那些不是基于 gobject 的 C 库中。但是你也可以定义自己的 Compact 类，但是你要认识到这种类只有非常有限的特性，比如：不能继承，不能实现接口，没有私有域等，它们是非常轻量级的类。

Compact 类默认是不支持引用计数的。因此这种类的对象只有一个强引用，该引用生存周期结束时对象会被自动销毁，其他对于该对象的引用只能是无属引用。

```
[Compact]
public class Foo {
    public void method ();
}

void main () {
    Foo foo = new Foo ();    // allocate
    foo.method ();
    unowned Foo bar = foo;
    Foo baz = (owned) foo;   /* ownership transfer: now 'baz' is the
                               "owning" reference for the object */
    unowned Foo bam = baz;
} // free ("owning" reference 'baz' got out of scope)
```

基于 Compact 类的引用计数机制的内存管理

可以手动为 Compact 类增加引用计数的功能。此时，你必须使用代码属性来告诉 Vala 哪些方法是用于引用计数的。

```
[Compact]
[CCode (ref_function = "foo_up", unref_function = "foo_down")]
```

```
public class Foo {
      public int ref_count = 1;

      public unowned Foo up () {
            ++ref_count;
            return this;
      }
      public void down () {
            if (--ref_count == 0) {
                  delete (void*) this;
            }
      }
      public void method () { }
}

void main () {
      Foo foo = new Foo ();     // allocate, ref
      foo.method ();
      Foo bar = foo;            // ref

} // unref, unref => free
```

你将发现，以上一切的过程又重新被自动管理了。

基于 immutable compact 类的非引用计数机制但拷贝的内存管理

如果一个 Compact 类不支持引用计数机制，但是这个类是不变的（immutable，内部状态不会改变）并且拥有一个用于在该类的对象在赋值给强引用的时候自动拷贝该对象的拷贝函数。

```
[Compact]
[Immutable]
```

```
[CCode (copy_function = "foo_copy")]

public class Foo {
      public void method () { }
      public Foo copy () {
            return new Foo ();
      }
}

void main () {
      Foo foo = new Foo ();   // allocate
      foo.method ();
      Foo bar = foo;          // copy

} // free, free
```

如果出于需要细微控制的理由，仍然可以使用无属引用来阻止拷贝：

```
void main () {

      Foo foo = new Foo ();   // allocate
      foo.method ();
      unowned Foo bar = foo;

} // free
```