

---

# Fong Documentation

*Release alpha*

**fong**

May 16, 2019



## 目录

1	C/C++	3
2	Python	77
3	Linux/Shell	109
4	Git	117
5	机器学习	125
6	深度学习	135
7	资源链接	151
8	实用软件	153
9	Tech	155
10	其他	157



---

**Note:** 文中可能存在错误，欢迎 PR。

<https://github.com/fongyk>

<https://github.com/fongyq>

---



## 1.1 main 函数

### 1.1.1 返回值

C++ main 函数的返回值必须是 `int`，即整型类型。在大多数系统中，main 的返回值被用来指示状态，返回值 0 表示执行成功，非 0 的返回值含义由系统定义，通常用来指出错误类型。

Windows 系统下运行可执行文件（如 `launch.exe`）可以直接忽略其扩展名 `.exe`：

```
launch
```

Unix 系统下需要使用全文件名，包括扩展名：

```
./a.out
```

访问 main 函数返回之后的方法依赖于系统。在 Windows 和 Unix 系统中，执行完一个程序之后，都可以通过 `echo` 命令来获取返回值。

Windows:

```
echo %ERRORLEVEL%
```

Unix:

```
echo $?
```

### 1.1.2 处理命令行选项

main 函数的形参列表有两种形式：

```
int main(int argc, char *argv[]){ ... }

int main(int argc, char **argv){ ... }
```

第一种形参 `*argv[]` 中, `argv` 是一个数组, 它的元素是指向 C 风格的字符串的指针; 第二种形参 `**argv` 中, `argv` 指向 `char*`。参数 `argc` 表示数组中字符串的数量。

当实参传给 `main` 函数之后, `argv` 的第一个元素指向程序的名字或者一个空字符串, 接下来的元素依次传递命令行提供的实参。最后一个指针之后的元素值保证为 0。例如, 执行:

```
launch -d -o ofile data
```

`launch` 是可执行文件。那么, `argc=5`, `argv` 包含如下的 C 风格字符串:

```
1 argv[0] = "launch";
2 argv[1] = "-d";
3 argv[2] = "-o";
4 argv[3] = "ofile";
5 argv[4] = "data";
6 argv[5] = "0";
```

---

**Note:** 当使用 `argv` 中的实参时, 实参是从 `argv[1]` 开始的; `argv[0]` 保存的是程序名, 而非用户输入。

---

### 1.1.3 参考资料

《C++ Primer 第 5 版中文版》Page 2, Page 197。

## 1.2 数组

### 1.2.1 指针

```
1 #include <ctime>
2 using namespace std;
3
4 struct TreeNode
5 {
6     int val;
7     TreeNode* left;
8     TreeNode* right;
9     TreeNode(int x) :val(x), left(nullptr), right(nullptr){} /* 唯一的构造函数, 必须给的参数 x */
```

(continues on next page)



(continued from previous page)

```

10 };
11
12 int main(int argc, char ** argv)
13 {
14     //int* p = new int(1); /* 这两行与下面三行等效 */
15     //cout << *p << endl;
16     int* p = new int;
17     *p = 1;          /* p 已经申请了内存空间，可以直接赋值 */
18     cout << *p << endl;
19
20     TreeNode* q = new TreeNode(10);
21     cout << q->val << endl;
22
23     TreeNode node(100);
24     TreeNode* r = &node; /* r 不能 delete */
25     cout << r->val << endl;
26
27     delete p;
28     delete q;
29
30     return 0;
31 }

```

**Note:** 两个指针变量的值相同，则这两个指针 **指向同一内存单元地址**或都为 **空指针**。不存在多个变量占用同一内存单元的情形。

### 1.2.2 指向函数的指针

定义形式:

函数返回值类型 (\*指针变量名)(参数列表)

指向函数的指针是让函数的入口地址赋给指针变量，类似于指向数组的指针是把数组首地址赋给指针变量。

```

1  #include<iostream>
2  using namespace std;
3
4  double square(double x)
5  {
6      return x * x;
7  }

```

(continues on next page)

(continued from previous page)

```
8
9 int main()
10 {
11     double (*p)(double x);
12     p = square; // 用函数名 square 初始化指针
13     cout << square(1.6) << endl;
14     cout << p(1.6) << endl;
15     cout << (*p)(1.6) << endl; // 三者等效
16     return 0;
17 }
```

还可以定义指针数组，

```
1 // 定义一个指向函数的指针类型，名为 MenuFood。函数参数列表为空，返回值为空。
2 typedef void (* MenuFood)();
3
4 void food1();
5 void food2();
6 void food3();
7 void food4();
8
9 MenuFood p[] = {food1, food2, food3, food4}; // 该数组的每一个元素都是指向函数的指针。
```

---

**Note:** 指针数组：类型 \* 数组名 [长度]

```
char *name[] = {"allen", "martin", "clark"};
```

指向行向量的指针变量：类型 (\* 变量名)[长度]

```
int (*pa)[10];
int *p;
int a[3][10];

p = a[0]; // 或 &a[0][0]
pa = a; // 使用：*(*(pa + i) + j)
```

---

### 1.2.3 动态数组

声明与定义一个动态数组的格式一般如下：

```

1  int** da = new int*[r];
2  for(int i = 0; i < r; ++i)
3  {
4      da[i] = new int[c];
5  }

```

内存释放:

```

1  for(int i = 0; i < r; ++i)
2  {
3      delete[] da[i]; // 释放指针指向的内存空间
4      da[i] = nullptr; // 置为空指针，防止出现‘野指针’
5  }
6  delete[] da;
7  da = nullptr;

```

内存组织形式:

动态数组在堆 (heap) 区分配内存，静态数组在栈 (stack) 区分配内存。

假如我们已经得到一个 3x4 的动态数组 da，其指针关系如下:

```

      da
      ↓
da[0] → da[0][0] da[0][1] da[0][2] da[0][3]
da[1] → da[1][0] da[1][1] da[1][2] da[1][3]
da[2] → da[2][0] da[2][1] da[2][2] da[2][3]

```

其中，da[0]、da[1]、da[2] 的地址是连续的，依次相差 sizeof(da[0]) (一个指针的大小，32 位编译器下为 4，64 位编译器下为 8)，比如:

```
&da[0] + sizeof(da[0]) == &da[1]
```

如果把 da 看作 3 行 4 列的二维数组，那么 da 的每一行元素的地址是连续的，依次相差 sizeof(da[0][0])；但是行与行之间的地址是不连续的，比如:

da[0][0], da[0][1], da[0][2], da[0][3] 地址连续;  
da[1][0], da[1][1], da[1][2], da[1][3] 地址连续;  
da[0][3] 与 da[1][0] 地址不连续;

另外:

&da, &da[0], &da[0][0] 三者的数值是不相等的。  
如果数组是静态数组, 则&da, &da[0], &da[0][0] 三者的数值是相等的;  
且静态数组的行与行之间的地址连续。

---

**Note:** 对于动态数组, 指针的地址和指针的值不能混淆, 我们讲 da[0]、da[1]、da[2] 的地址是连续的, 但是他们本身的价值没有关系, 即 `da[0] + sizeof(da[0]) != da[1]`。注意有没有 & 的区别。

---

如果想要定义连续内存空间的动态数组, 可以按如下方式进行:

```
1 // int** f; // f 的声明
2 template<typename T>
3 void Init2DArray(T** &f, const int row, const int col)
4 {
5     f = new T*[row];
6     f[0] = new T[row * col];
7     for(int i = 1; i < col; ++i)
8     {
9         f[i] = f[0] + col * i;
10    }
11 }
```

内存释放方式如下:

```
1 template<typename T>
2 void Delete2DArray(T** &f)
3 {
4     if(f != nullptr)
5     {
6         if(f[0] != nullptr)
7         {
8             delete[] f[0];
9             f[0] = nullptr;
10        }
11        delete[] f;
12        f = nullptr;
13    }
14 }
```

上面的 `Init2DArray` 在申请内存的时候，建立了 `row x col` 的二维动态数组。实际上，二维动态数组不强求列对齐，即各行的长度可以不一样，因此可以下面像这样定义：

```
f[i] = f[0] + offset_i; // offset_i 是第 i 行首地址相对于第 0 行首地址的偏移量
```

### 1.2.4 另类的数组表达

有如下程序：

```
1 int a[10];
2 int b[7][5];
3
4 0[a] = 5;
5 9[a] = 7;
6 0[b][0] = 1;
7 0[b][1] = 2;
8 0[b][2] = 3;
```

这些表达式能够正常编译和执行，是因为对于 C/C++ 而言：

```
a[0] 等价于 *a 等价于 *(a+0) 等价于 *(0+a) ==> 等价于 0[a];
```

所以可以推出二维表达式：

```
b[0][1] = *(b[0] + 1) = *(1 + b[0]) = 1[b[0]]
b[0][1] = *(*b[0] + 1) = *((0+b) + 1) = *(0[b] + 1) = 0[b][1]
b[0][1] = *(*b[0] + 1) = *(1 + *(0+b)) = 1[0[b]]
```

**Note:** `b`: 相当于 `&b[0]`，第 0 行的首地址。

`*b`: 相当于 `b[0]`，`&b[0][0]`，第 0 行第 0 列的首地址。

`b[0][1]` 等价于 `*(b[0] + 1)`，`*(*(b+0)+1)`，`*(b+0)[1]` (`[]` 优先级高于 `*`)。

### 1.2.5 数组实参

以 **非引用** 类型传递数组实参时，数组会退化为指针，形参复制的是这个指针的值（指向数组的第一个元素）。通过指针形参做的任何改变，都是在修改数组元素本身。

如果以 **引用** 形式传递数组实参，那么编译器不会将数组实参转化为指针，而是传递数组的引用本身。

编译器会检查数组实参的大小与形参大小是否匹配。

非引用形式：

```

1 void func1(int *arr); // 函数可能会改变数组
2 void func2(const int *arr); // 不能改变数组
3
4 void func3(int arr[100])
5 {
6     cout << sizeof(arr) << endl; // 4
7 }
8 // int arr[100];
9 // func3(arr); // 调用 func3

```

引用形式：

```

1 void func4(int (&arr)[100])
2 {
3     cout << sizeof(arr) << endl; // 100
4 }
5 // int arr[100];
6 // func4(arr); // 调用 func4

```

## 1.2.6 数组地址与加法

假设我们已经定义了一个数组：

```
Type a[L][M][N] = {...};
```

1.  $a$  ,  $a[0]$  ,  $a[0][0]$  ,  $\&a$  ,  $\&a[0]$  ,  $\&a[0][0]$  ,  $\&a[0][0][0]$  的数值都是一样的。
2.  $\&a$  是 4 级指针, 类型是  $\text{int} (*)[L][M][N]$  , 指向  $a$  这个数组。
3.  $a$  是 3 级指针, 类型是  $\text{int} (*)[M][N]$  , 三维数组的数组名, 是数组  $a$  的首地址。
4. 大小计算如下:

```

sizeof(&a) = 4; // 指针的大小, 32 位编译器
sizeof(a[0][0][0]) = sizeof(Type);
sizeof(a[0][0]) = N * sizeof(Type);
sizeof(a[0]) = M * N * sizeof(Type);
sizeof(a) = L * M * N * sizeof(Type); // 整个数组的大小

```

5. 加法运算:

```

a + i = a + i * sizeof(a[0]);
&a + i = a + i * sizeof(a);
这里 (&a + 1) 就已经跳过了整个数组。

```

6. 定义指针  $\text{int} *ptr = (\text{int} *)(&a + 1)$  , 则  $(ptr - 1)$  指向数组  $a$  的最后一个元素。

## 1.2.7 malloc/free 与 new/delete

### 相同点

都可用于申请动态内存和释放内存。

### 不同点

#### 1. 属性

malloc/free 是 C/C++ 的 **库函数**，在头文件 `stdlib.h` 中声明。

```
void *malloc(size_t size);
void free(void *pointer);
```

因为 `malloc()` 函数的返回值类型为 `void*`，所以需要在函数前面进行相应的强制类型转换。

`new/delete` 是 C++ 的 **运算符**。

#### 2. 参数

使用 `new` 操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算；

```
int *pi=new int[n]; //指针 pi 指向长度为 n 的数组，未初始化
```

而 `malloc` 则需要显式地指出所需内存的尺寸。

```
int *p=(int *)malloc(25*sizeof(int)); //指向整型的指针 p 指向一个 25 个 int 整型空间的地址
int *p=(int *)malloc(99); //指向整型的指针 p 指向一个大小为 99 字节的内存的地址
```

`malloc` 可以分配任意字节，`new` 只能分配实例所占内存的整数倍数大小。

#### 3. 分配失败

`new` 内存分配失败时，会抛出 `bad_alloc` 异常；`malloc` 分配内存失败时返回 `NULL`。

#### 4. 功能

`new` 做两件事，先分配内存，再调用类的构造函数；同样，`delete` 会调用类的析构函数和释放内存。而 `malloc` 和 `free` 只是分配和释放内存。

对于内部数据类型（如 `int`，`char` 等）的对象，没有构造和析构的过程，对它们而言，`malloc/free` 和 `new/delete` 等价。

对于非内部数据类型的对象而言，`malloc/free` 无法满足动态对象的要求。

#### 5. 重载 (overload)

`new/delete` 可以重载成为函数，可以自定义申请过程，比如记录申请内存的长度以及跟踪每个对象的指针。

malloc/free 不能重载。

**Warning:**

- new 和 delete 一定要配对使用。
- 对空指针使用 delete 是安全的。
- 不能使用 delete 释放绑定到对象的指针。

```
int val = 5;
int* p = &val;
delete p; // error, memory not allocated by new
```

## 1.2.8 参考资料

1. 静态数组与动态数组：

<https://blog.csdn.net/liupeng900605/article/details/7526753>

2. 浅谈 new/delete 和 malloc/free 的用法与区别：

<https://www.cnblogs.com/maluning/p/7944231.html>

3. malloc/free 与 new/delete 的区别：

<https://blog.csdn.net/hackbuteer1/article/details/6789164>

## 1.3 数据类型

### 1.3.1 常用内置数据类型的大小

以下结果若非特别指出，均在 Windows 系统下由编译器 Visual Studio 测试得到。



Table 1: 类型大小

类型	size/32 位编译器	size/64 位编译器
char	1	1
char*	4	8
int	4	4
int*	4	8
short	2	2
long	4	4 (8/linux)
long*	4	8
long long	8	8
float	4	4
double	8	8
size_t	4	8
size_type	4	8
bool	1	1
string	28	40

### 1.3.2 sizeof 与 strlen

#### 1. sizeof

`sizeof()` 是 **运算符**，计算的是分配的内存空间大小 (单位为字节)，编译时就会计算，不受里面存储内容的影响。

`sizeof()` 可以用数据类型、数组、字符串等做参数。

#### 2. strlen

`strlen()` 是 **函数**，计算的是字符串的实际长度 (字符的个数)，以 `'\0'` 结束但长度 **不包括** `'\0'`，程序执行时才计算结果。`strlen()` 只能用 `char*` 类型做参数。

#### 3. 实例

定义以下变量：

```

1 char *strA = "abcdef";
2 char strB[] = "abcdef";
3 char strC[5] = {'a'};
4 char strD[3] = {'a', 'b', 'c'};
5 char strE[] = {'a', 'b', 'c'};
6 char strF[] = {'a', 'b', 'c', '\0'};
7 int y[] = {1,2,3};

```

结果如下：

```

sizeof(strA) = 4 : 指针的大小
sizeof(strB) = 7 : 该字符数组用字符串初始化, 因此 strB 就是一个字符串, 字符串以 '\0' 结尾, 则大小为 6+1=7
sizeof(strC) = 5 : 字符数组所占内存为 5 字节
sizeof(strD) = 3 : 字符数组所占内存为 3 字节
sizeof(strE) = 3 : 字符数组中有 3 个字符
sizeof(strF) = 4 : 字符数组中有 4 个字符, 包括 '\0'
sizeof(y) = 12 : 4 * 3 = 12 字节

strlen(strA) = 6 : 字符串长度为 6, 不包括 '\0'
strlen(strB) = 6 : 字符串长度为 6, 不包括 '\0'
strlen(strC) = 1 : 字符数组中只有 1 个字符
strlen(strD) 不定, 因为数组 strD 末尾没有人为补 '\0', 因此 strD 是一个普通的字符数组, 而不是字符串
strlen(strE) 不定, 因为数组 strE 末尾没有人为补 '\0', 因此 strD 是一个普通的字符数组, 而不是字符串
strlen(strF) = 3 : 字符串长度为 3, 不包括 '\0'

```

**Warning:** 如果字符数组以字符常量进行初始化且字符个数大于 1, 如上例中的 `strD` 和 `strE`, 如果不在末尾人为添加 `'\0'`, 则该字符数组不是字符串, 使用函数 `strlen` 求得的大小不定, 且该字符数组的内容也是未知的。虽然 `strD` 只有 3 个字节空间且刚好包含 3 个字符, 但是 `cout<<strD` 的结果也是不定的。正确的定义应该是 `strF`。

### 1.3.3 float 和 double

单精度浮点型 `float` 的精度为 6 – 7 位有效数字, 双精度浮点型 `double` 的精度为 15 – 16 位有效数字。

```

1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char ** argv)
5  {
6      int i = 200000003 / 100000002; // 1.9999999900000003
7
8      float f_i = 200000003 / (float)100000002; // 浮点型常数默认为 const double, 或用 200000003.0f 指定为 float。
9      float f_f = (float)200000003.0 / (float)100000002.0; // 若不进行强制类型转换, 会有 warning: ↪
10     float f_d = (float)200000003.0 / (double)100000002; // warning: truncation from 'double' to 'float' ↪
11     double d_d = 200000003 / (double)100000002;

```

(continues on next page)

(continued from previous page)

```

12
13 cout.setf(ios::fixed); // 浮点数定点输出
14 cout.setf(ios::showpoint); // 显示小数位
15 cout.precision(10); // 固定为 10 位精度 (四舍五入)
16 cout << i << endl; // 1
17 cout << f_i << ends << static_cast<int>(f_i) << endl; // 2.0000000000 2
18 cout << f_f << ends << static_cast<int>(f_f) << endl; // 2.0000000000 2
19 cout << f_d << ends << static_cast<int>(f_d) << endl; // 2.0000000000 2
20 cout << d_d << ends << static_cast<int>(d_d) << endl; // 1.9999999900 1
21 cout.precision(2);
22 cout << d_d << ends << static_cast<int>(d_d) << endl; // 2.00 1
23
24 cout << boolalpha; // 设置布尔型输出格式
25 cout << (i == static_cast<int>(f_f)) << endl; // false
26 cout << (i == static_cast<int>(f_d)) << endl; // false
27 cout << (i == static_cast<int>(d_d)) << endl; // true (只有 double 转换到 int 的结果与 i 一致)
28
29 return 0;
30 }

```

### 1.3.4 参考资料

1. 数据类型的数值范围

<https://blog.csdn.net/qianbitou000/article/details/51939055/>

2. 关于 strlen 与 sizeof 的区别

[https://blog.csdn.net/zhengqijun\\_/article/details/51815081](https://blog.csdn.net/zhengqijun_/article/details/51815081)

3. C++ 中的 cout.setf() 函数

<https://blog.csdn.net/baishuiniyaonulia/article/details/79144033>

## 1.4 类的大小

计算类的大小遵循以下原则：

- 内存对齐。
- 类的大小与普通数据成员有关，与成员函数、静态成员无关。即普通成员函数、静态成员函数、静态数据成员均对类的大小无影响。
- 虚函数对类的大小的影响体现在 **虚函数表指针** 的大小。
- 虚继承对类的大小的影响体现在 **虚基类表指针** 的大小。

以下结果均是在 32 位 Visual Studio 2013 下编译得到。

### 1.4.1 空类

C++ 标准指出，不允许一个对象（当然包括类对象）的大小为 0，不同的对象不能具有相同的地址。这是由于：

- new 需要分配不同的内存地址，不能分配内存大小为 0 的空间。
- 避免除以 `sizeof(*)` 时得到除以 0 错误

每个类在内存中都有唯一的标识，因此空类被实例化时，编译器会隐含地为其添加一个字节，以作区分。

```
1 class Empty
2 {
3 };
4 // sizeof(Empty) = 1
```

### 1.4.2 普通数据成员

遵循内存对齐原则。

```
1 class A
2 {
3     int a;
4     char c;
5 };
6 // sizeof(A) = 4 + 4 = 8
7
8 class B
9 {
10    double a;
11    char c;
12 };
13 // sizeof(B) = 8 + 8 = 16
```

### 1.4.3 普通继承

普通类的继承，派生类的大小 = 派生类数据成员大小 + 基类数据成员大小。

```
1 class A
2 {
3     int a;
4 };
```

(continues on next page)

(continued from previous page)

```

5
6 class B: public A
7 {
8     char ch;
9     double b;
10 }
11 // sizeof(B) = 8 + 8 = 16 (对齐: 4+1 -> 8)
12
13 class C: public A
14 {
15     double c;
16     char ch;
17 }
18 // sizeof(C) = 8 + 8 + 8 = 24 (对齐: 4 -> 8, 1 -> 8)

```

## 继承空类

派生类继承空类后，派生类如果有自己的数据成员，而空基类的一个字节并不会加到派生类中去。

```

1 class Empty
2 {
3 };
4
5 class A: public Empty
6 {
7     int b;
8 }
9 // sizeof(A) = 4

```

## 类包含空类对象数据成员

空类的 1 字节是会被计算进去的。

```

1 class Empty
2 {
3 };
4
5 class A
6 {
7     int b;
8     Empty e;

```

(continues on next page)

(continued from previous page)

```

9  }
10 // sizeof(A) = 4 + 4 = 8

```

### 1.4.4 虚函数与继承

虚函数 (Virtual Function) 是通过一张 **虚函数表 (Virtual Table, vtable)** 来实现的。每当 **创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时**，编译器就会为这个类创建一个虚函数表保存该类 **所有虚函数的地址**。

当一个类中包含虚函数时，会有一个指向其虚函数表的指针 `vptr`，系统为类指针分配大小为 4 个字节 (即使有多个虚函数)。当构造该派生类对象时，其成员 `vptr` 被初始化指向该派生类的 `vtable`。所以可以认为 `vtable` 是该类的所有对象共有的，在定义该类时被初始化；而 `vptr` 则是每个类对象都有独立一份的，且在该类对象被构造时被初始化。

#### 单继承

派生类的大小 = 派生类的普通数据成员的大小 + 1 个 `vptr` 指针的大小

```

1  class Base
2  {
3      virtual void f1();
4      virtual void f2();
5  };
6  // sizeof(Base) = 4
7
8  class Derived: public Base
9  {
10     virtual void f1(); // 覆盖了基类中的 f1(), ** 多态 **
11     virtual void f3();
12 };
13 // sizeof(Derived) = 4 + 4 = 8

```

#### 多继承

每个基类都有自己的虚表 (`vtable`)。

派生类的成员函数被放到了第一个基类的表中。

派生类的大小 = 派生类的普通数据成员的大小 + 基类的普通数据成员的大小 + `n` 个 `vptr` 指针的大小。`n` 是继承的基类的个数。

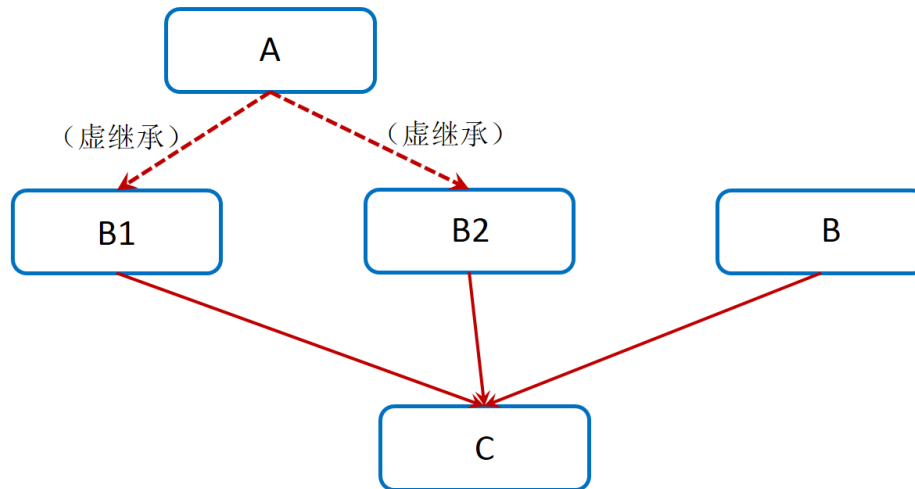
```
1  class A
2  {
3  };
4  // sizeof(A) = 1
5
6  class B
7  {
8      char ch;
9      virtual void f0();
10 };
11 // sizeof(B) = 4 + 4 = 8
12
13 class C
14 {
15     char ch1;
16     char ch2;
17     virtual void f1();
18     virtual void f2();
19 };
20 // sizeof(C) = 4 + 4 = 8
21
22 class D: public A, public C
23 {
24     int d;
25     virtual void f1();
26     virtual void f2();
27 };
28 // sizeof(D) = 4 + 4 + 1*4 = 12
29
30 class E: public B, public C
31 {
32     int e;
33     virtual void f0();
34     virtual void f1();
35 };
36 // sizeof(E) = 4 + 4 + 4 + 2*4 = 20
```

### 1.4.5 虚继承

尽管派生列表中同一个基类只能出现一次，但实际上派生类可以多次继承同一个类。派生类可以通过它的两个直接基类分别继承同一个间接基类，也可以直接继承某个基类，然后通过另一个基类再一次间接继承该类。

在默认情况下，派生类中含有继承链上每个类对应的子部分。如果某个类在派生过程中出现多次，则派生类中将包含该类的多个子对象。这会导致两个问题：第一，浪费存储空间；第二，存在二义性。

虚继承可以解决上述问题。



虚继承的目的是令某个类做出声明，承诺共享它的基类。其中，共享的基类子对象称为 **虚基类** (上图中的类 **A**)。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含唯一一个共享的虚基类子对象。

虚继承底层实现原理与编译器相关，一般通过虚基类指针和虚基类表实现。每个虚继承的子类都有一个 **虚基类指针 (Virtual Base Table Pointer, vbptr)**，占用一个指针的存储空间) 和虚基类表 (不占用类对象的存储空间) (需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了)。

实际上，vbptr 指向了一个 **虚基类表 (Virtual Table)**，虚基类表中记录了虚基类与本类的偏移地址。通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类 (虚基类) 的两份同样的拷贝，节省了存储空间。

当虚继承的子类被当做父类继承时，虚基类指针也会被继承。上图中，C 继承了 B1, B2，也就继承了 2 个虚基类指针。

总体需要考虑：数据成员的大小、虚基类指针的大小、虚函数指针的大小。

```

1  class A {
2      int a;
3  };
4  // sizeof(A) = 4
5
6  class B :virtual public A
7  {
8      int b;
9      virtual void myfunB();
10 };
11 // sizeof(B) = 16

```

(continues on next page)



(continued from previous page)

```
12
13 class C :virtual public A
14 {
15     double c;
16     virtual void myfunC();
17 };
18 // sizeof(C) = 28
19
20 class D :public B, public C
21 {
22     int d;
23     virtual void myfunD();
24 };
25 // sizeof(D) = 52
```

## 1.4.6 内存对齐

内存对齐，是为了让内存存取更有效率而采用的一种编译阶段优化内存存取的手段。

- 内存对齐是指首地址对齐，而不是说每个变量大小对齐。
- 分配内存的顺序是按照声明的顺序。
- 每个变量相对于起始位置的偏移量必须是该变量类型大小的整数倍，如果不是整数倍则空出内存，直到偏移量是整数倍为止。
- 结构体内存对齐要求结构体内每一个成员变量都是内存对齐的。
- 整个结构体的大小必须是结构体里面变量类型最大值的整数倍。
- 如果一个结构里有某些结构体成员，则结构体成员要从其内部”最宽基本类型成员”的整数倍地址开始存储。比如，struct A 里存有 struct B, B 里有 char, int, double 等类型的成员变量，那 B 应该从 8 的整数倍开始存储。

内存对齐的作用：

- 平台原因 (移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
- 性能原因：经过内存对齐后，CPU 的内存访问速度大大提升。

```
1 class Data
2 {
3     char c;
4     int a;
5     char d;
```

(continues on next page)

(continued from previous page)

```

6 };
7 // sizeof(Data) = 1 + (3) + 4 + 1 + (3) = 12, 括号内表示补齐的字节数。
8
9 class Data
10 {
11     char c;
12     char d;
13     int a;
14 };
15 // sizeof(Data) = 1 + 1 + (2) + 4 = 8
16
17 class BigData
18 {
19     char array[33];
20 };
21 // sizeof(BigData) = 33
22
23 class Data
24 {
25     BigData bd;
26     //int integer; // 不管有没有注释这一行, sizeof(Data) 结果一样
27     double d; // d 的起始偏移量必须为 8 的倍数, 且大于 33, 则其偏移量为 40
28 };
29 // sizeof(Data) = 48

```

### 1.4.7 位域

C/C++ 中以一定区域内的位 (bit) 为单位来表示的数据成为位域, 位域必须指明具体的数目。位域的作用主要是节省内存资源, 使数据结构更紧凑。

1. 一个位域必须存储在同一个字节中, 不能跨两个字节, 故位域的长度不能大于一个字节的长度

```

1 struct BitField
2 {
3     unsigned int a:4; //占用 4 个二进制位
4     unsigned int :0; //空位域, 自动置 0, 此时占满 1 个 int 存储单元, 即 4 字节
5     unsigned int b:4; //占用 4 个二进制位, 从第二个字节存储单元开始存放
6     unsigned int c:4; //占用 4 个二进制位
7     unsigned int d:5; //占用 5 个二进制位, 剩余的 3 个 bit 不够存储 4 个 bit 的数据, 从下一个
    存储单元开始存放
8     unsigned int :0; //空位域, 自动置 0, 此时占满 2 个 int 存储单元, 即 8 字节
9     unsigned int e:4; //占用 4 个二进制位, 从第三个 int 存储单元开始存放
10 };

```

(continues on next page)

(continued from previous page)

```
11 // sizeof(BitField) = 3 * 4 = 12
```

2. 取地址操作符 & 不能应用在位域字段上
3. 位域字段不能是类的静态成员
4. 位域字段在内存中的位置是按照从低位向高位的顺序放置的

```
1 struct BitField
2 {
3     unsigned char a:2; //最低位;
4     unsigned char b:3;
5     unsigned char c:3; //最高位;
6 };
```

#### 5. 位域的对齐

```
1 struct BFA
2 {
3     unsigned char a:2;
4     unsigned char b:3;
5     unsigned char c:3;
6 };
7 // sizeof(BFA) = 1
8
9 struct BFB
10 {
11     unsigned char a:2;
12     unsigned char b:3;
13     unsigned char c:3;
14     unsigned int d:4;
15 };
16 // sizeof(BFA) = 1 + (3) + 4 = 8
```

#### 6. 一个例子

```
1 struct num
2 {
3     int a:3;
4     int b:2;
5     int c:1;
6 };
7
8 int main()
9 {
```

(continues on next page)

(continued from previous page)

```
10     struct num n = {8, -6, 5};
11     cout << n.a << endl;
12     cout << n.b << endl;
13     cout << n.c << endl;
14     return 0;
15 }
```

- $8 = (00001000)_2$  , 8 在计算机中的补码也是 00001000 ,  $a$  用 3 位表示, 取低位为 000 ( 00000000 ), 原码也是 00000000 , 则  $a = 0$  。
- $-6 = (10000110)_2$  ,  $-6$  在计算机中的补码是 11111010 ,  $b$  用 2 位表示, 取低位为 10 ( 11111110 ), 原码是 10000010 , 则  $b = -2$  。
- $5 = (00000101)_2$  , 5 在计算机中的补码也是 00000101 ,  $c$  用 1 位表示, 取低位为 1 ( 11111111 ), 原码是 10000001 , 则  $c = -1$  。

---

**Note:** C++ 标准库提供了一个 **bitset** 类模板, 它可以辅助操纵位的集合。

---

### 1.4.8 参考资料

1. c++ 类大小问题

<https://www.cnblogs.com/sz-leez/p/7119232.html>

2. c++ 类的大小计算

<https://blog.csdn.net/fengxinlinux/article/details/72836199>

3. 虚继承

《C++ Primer 第 5 版中文版》Page 717-718。

4. C++ 中虚继承的作用及底层实现原理

<https://blog.csdn.net/bxw1992/article/details/77726390>

5. c++ 中的内存对齐

<https://www.cnblogs.com/suntp/p/MemAlignment.html>

6. C/C++ 内存对齐原则及作用

<https://blog.csdn.net/chy19911123/article/details/48894579>

7. C/C++ 位域知识小结

<https://www.cnblogs.com/pure/archive/2013/04/22/3034818.html>

## 1.5 交换函数 swap

1. 库函数，包含在头文件 `<algorithm>` 中。

```
std:: swap(a, b);
```

2. 指针。

```
1  template<class T>
2  void swap(T *x, T *y)
3  {
4      T tmp = *x;
5      *x = *y;
6      *y = tmp;
7  }
```

3. 引用。

```
1  template<class T>
2  void swap(T &x, T &y)
3  {
4      T tmp = x;
5      x = y;
6      y = tmp;
7  }
```

4. 异或。适用于整型/字符/枚举类型，浮点型不适用。`swap(a, a)` 会导致 `a=0` 。

```
#define swap(a, b) a^=b^=a^=b;
```

4. 赋值。受编译器影响，先执行 `a+b` 还是先执行 `b=a` 。

```
#define swap(a, b) a=a+b-(b=a);
```

5. 加减。无需申请额外空间。

```
1  template<class T>
2  void swap(T &x, T &y)
3  {
4      x = x + y;
5      y = x - y;
6      x = x - y;
7  }
```

## 1.6 数组形参

### 1.6.1 非引用

当数组以 **非引用** 类型传递，数组会悄悄退化为指针，形参复制的是这个指针的值（指向数组第一个元素）。通过该形参做的任何改变都是在修改数组元素本身。

```
1 void func1(int arr[100])
2 {
3     cout << sizeof(arr) << endl; // 指针的大小为 4 (32 位编译器)
4     /*
5         function body
6     */
7 }
8
9 void func2(int *arr)
10 {
11     cout << sizeof(arr) << endl; // 指针的大小为 4 (32 位编译器)
12     /*
13         function body
14     */
15 }
16
17 int a[10] = {1,2,3};
18 func2(a);
```

### 1.6.2 引用

如果形参是数组的 **引用**，编译器不会将数组实参转化为指针，而是传递数组的引用本身。编译器会检查数组实参的大小与形参是否匹配。

```
1 void func1(int (&arr)[10])
2 {
3     cout << sizeof(arr) << endl; // 大小为 4*10=40 (32 位编译器)
4     /*
5         function body
6     */
7 }
8
9 int a[10] = {1,2,3};
10 func2(a);
```

## 1.7 指针与引用

1. 指针是一个对象，有存储的 **值**和 **地址**，存储的数据类型是数据的地址；非常量指针可以被重新赋值，指向另一个对象。引用是对象的别名，必须初始化并总是指向（代表）最初绑定的那个对象，对对象及其引用进行取地址操作得到的结果相同。

```

1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char ** argv)
5  {
6      int k = 1;
7      int* pk = &k;
8      int& rk = k;
9
10     cout << "&k:" << &k << endl;    // 0029FC44
11     cout << "k:" << k << endl;      // 1
12
13     cout << "&pk:" << &pk << endl; // 0029FC68
14     cout << "pk:" << pk << endl;    // 0029FC44 (pk = &k)
15     cout << "*pk" << *pk << endl;  // 1
16
17     cout << "&rk:" << &rk << endl; // 0029FC44 (&rk = &k)
18     cout << "rk:" << rk << endl;   // 1
19
20     return 0;
21 }
```

2. 指针可以有多级，但是引用只能是一级（不存在引用的引用）。

3. 有 null pointer，没有 null reference，故使用前无需检查是非为空。

```

1  void rValue(const int &x)
2  {
3      cout << x << endl;
4  }
5
6  void pValue(const int* p)
7  {
8      if(p) cout << *p << endl;
9  }
```

4. 例子。

```
1 string s1("nancy");
2 string s2("candy");
3 string& rs = s1;
4 string* ps = &s2;
5 rs = s2; // rs 仍指向 s1, 但是 s1 值变为 "candy".
6 ps = &s2; // ps 指向 s2, s1 无变化
```

## 1.8 重载、覆盖、隐藏

### 1.8.1 重载 (Overloading)

同一可访问区内被声明的几个具有不同参数列表（参数个数，参数类型，参数顺序）的同名函数。不关心函数返回类型。

### 1.8.2 覆盖 (Overriding)

基类中被重写的函数，用 `virtual` 修饰。派生类重写的函数与被重写的函数保持同样的函数名、参数列表、返回类型。

使用 `virtual` 的同时，配合使用 `override` 关键字来说明派生类中的虚函数。这么做的好处是是程序员意图更加清晰（即：希望覆盖基类中的虚函数），同时让编译器发现错误。因为只有虚函数才能被覆盖。编译器会检查两个对应函数的声明是否匹配。

通过把某个函数指定为 `final`，拒绝对该函数进行覆盖。也在类名后面接 `final`，以禁止该类被继承。

```
1 class Base
2 {
3     virtual void f(int) const final;
4     virtual void f1(int) const;
5     virtual void f2();
6     void f3();
7 };
8
9 class Derived: Base
10 {
11     void f(int) const; // 错误: f 禁止覆盖
12     void f1(int) const override; // 正确
13     void f2(int) override; // 错误: 基类中没有形如 f2(int) 的函数
14     void f3() override; // 错误: f3 不是虚函数
15     void f4() override; // 错误: 基类中没有名为 f4 的函数
16 };
```

(continues on next page)



(continued from previous page)

```

17
18 class NoDerived final { /* */ }; // NoDerived 不能作为基类

```

**多态性** 我们把具有继承关系的多个类型成为多态类型，因为我们能够使用这些类型的“多种形式”而无须在意它们的差异。引用或指针的静态类型与动态类型不同，这正是 C++ 支持多态性的根本所在。

对非虚函数的调用在 **编译时**进行绑定。类似地，通过 **对象**本身进行的函数（虚函数或非虚函数）调用也在 **编译时**绑定。因为 **对象**的类型是确定不变的，通过对象进行的函数调用将在编译时绑定到该对象所属类中的函数版本。

当且仅当通过 **指针或引用**调用虚函数时，才在 **运行时**解析该调用，也只有在这种情况下对象的动态类型才有可能与静态类型不同（**动态绑定**）。

```

1  #include <iostream>
2  using namespace std;
3
4  class Base
5  {
6  public:
7      virtual void f(){ cout << "base" << endl; }
8  };
9
10 class Derived : public Base // 注意：这里必须为 public 继承
11 {
12 public:
13     void f(){ cout << "derived" << endl; }
14 };
15
16 int main(int argc, char ** argv)
17 {
18     Derived d = Derived(); // 派生类对象
19     Base* pb = &d; // 基类指针
20     pb->f(); // derived
21
22     Base b = Base(); // 基类对象
23     Base& rb = b; // 基类引用
24     rb.f(); // base
25
26     return 0;
27 }

```

### 抽象基类与纯虚函数

- 纯虚函数无须定义（非要定义的话，必须发生在类外部），在该函数的声明语句中（分号之前）加入 `= 0` 就可以将一个虚函数声明为纯虚函数。

- 含有纯虚函数的类是抽象基类。抽象基类负责声明接口，派生类负责覆盖该接口。如果派生类不给出对应基类中纯虚函数的定义，该派生类也是一个抽象基类。
- 不能直接创建抽象基类的对象。

---

**Note:** 基类中的虚函数在派生类中隐含地也是一个虚函数。当派生类覆盖了某个虚函数时，该函数在基类中的形参必须与派生类中的形参严格匹配。

我们可以将 **基类的指针或引用** 绑定到派生类的对象上。因此，当我们使用基类指针或引用时，实际上并不清楚该指针或引用所绑定的对象的真实类型。

---

---

**Note:** 构造函数 **不能声明** 为虚函数：一方面，创建一个对象时总要明确指定对象的类型。另一方面，虚函数对应一个指向虚函数表的指针（vptr），在创建对象之前，vptr 不存在，不可能完成动态绑定。

析构函数 **可以声明** 为虚函数：当基类指针指向派生类，使用基类指针删除对象时，如果析构函数不定义成虚函数，派生类中派生的部分无法完成析构。

构造函数 **不要调用** 虚函数。在基类构造的时候，虚函数是非虚，不会走到派生类中，即采用的静态绑定。显然，当我们构造一个子类的对象时，先调用基类的构造函数去构造子类中基类部分，此时子类部分还没有构造、初始化。如果在构造中调用虚函数，可能会调用一个还没有被初始化的对象，这是很危险的。

析构函数 **不要调用** 虚函数。析构的时候，首先调用子类的析构函数，析构掉对象的子类部分，然后调用基类的析构函数析构基类部分。如果在基类的析构函数里面调用虚函数，会导致其调用已经析构了的子类对象里面的函数，这是非常危险的。

总而言之：在运行构造函数或者析构函数时，对象都是不完整的，这种情况下的虚函数调用不会调用到外层派生类的虚函数。

---

**Warning:** error C2243: ‘type cast’: conversion from ‘Derived \*’ to ‘Base \*’ exists, but is inaccessible.

基类的指针和引用不能指向继承方式为 `protected` 与 `private` 的派生类对象，只能通过 `public` 继承。

### 1.8.3 隐藏 (Hiding)

派生类中的函数屏蔽了基类中的同名函数，不管参数列表是否相同。当参数不同时，无论基类中的函数是否被 `virtual` 修饰，基类函数都是被隐藏，而不是被覆盖。

### 1.8.4 参考资料

1. C++ 中重载、重写（覆盖）和隐藏的区别

<https://blog.csdn.net/zx3517288/article/details/48976097>

2. 《C++ Primer 第 5 版中文版》Page 538 – 540。

## 1.9 strcpy 函数

函数定义：

```
1 char* strcpy(char* dst, const char* src)
2 {
3     char* cp = dst;
4     while( *cp++ = *src++ ); /* Copy src over dst */
5     return( dst );
6 }
7
8 char src[10] = "abcd";
9 char dst[10];
10 char* copy = strcpy(dst, src);
```

### 1.9.1 形参 src

形参 src 定义为 const，防止函数对其进行修改。

### 1.9.2 额外指针 cp

cp++ 导致复制结束时，cp 指向的是 dst 绑定的字符串的尾部，因此不能直接返回 cp。

### 1.9.3 返回值

为了实现链式操作，将目的地址返回。

```
int length = strlen(strcpy(str, "Hello World"));
```

### 1.9.4 参考资料

1. 标准的 strcpy 函数

<https://www.cnblogs.com/elisha-blogs/p/4125799.html>

## 1.10 强制类型转换

### 1.10.1 `static_cast<type> (expr)`

1. `static_cast` 作用和 C 语言风格强制转换的效果基本一样，由于没有运行时类型检查来保证转换的安全性，所以这类型的强制转换和 C 语言风格的强制转换都有安全隐患。
2. 用于基本数据类型之间的转换，如把 `int` 转换成 `char`，把 `int` 转换成 `enum`。这种转换的安全性需要开发者来维护。
3. C++ 的任何的隐式转换都是使用 `static_cast` 来实现。
4. 基类和子类之间转换：其中子类指针转换成父类指针是安全的；但父类指针转换成子类指针是不安全的。（基类和子类之间的动态类型转换建议用 `dynamic_cast`）
5. 把空指针转换成目标类型的空指针。
6. 把任何类型的表达式转换成 `void` 类型。

### 1.10.2 `dynamic_cast<type> (expr)`

有条件转换，动态类型转换，运行时类型安全检查（转换失败返回 `NULL`）：

1. 安全的基类和子类之间转换。
2. 必须要有虚函数。
3. 相同基类不同子类之间的交叉转换，结果是 `NULL`。

```
1  // dynamic_cast
2  #include <iostream>
3  #include <exception>
4  using namespace std;
5
6  class Base { virtual void dummy() {} };
7  class Derived: public Base { int a; };
8
9  int main ()
10 {
11     try
12     {
13         Base * pba = new Derived;
14         Base * pbb = new Base;
15         Derived * pd;
16
17         pd = dynamic_cast<Derived*>(pba);
18         if (pd==0) cout << "Null pointer on first type-cast.\n";
```

(continues on next page)

(continued from previous page)

```

19
20     pd = dynamic_cast<Derived*>(pbb);
21     if (pd==0) cout << "Null pointer on second type-cast.\n";
22
23     } catch (exception& e) {cout << "Exception: " << e.what();}
24     return 0;
25 }
26
27 // 输出结果: Null pointer on second type-cast.

```

### 1.10.3 const\_cast<type> (expr)

1. 去掉或加上类型的 `const`、`volatile` 属性;
2. 常量指针被转化成非常量的指针，并且仍然指向原来的对象;
3. 常量引用被转换成非常量的引用，并且仍然指向原来的对象;
4. `const_cast` 一般用于修改指针。如 `const char *p` 形式。如果有一个函数，它的形参是 `non-const` 类型变量，而且函数不会对实参的值进行改动，这时我们可以使用类型为 `const` 的变量来调用函数，此时 `const_cast` 就派上用场了。

```

1 // const_cast
2 #include <iostream>
3 using namespace std;
4
5 void print (char * str)
6 {
7     cout << str << '\n';
8 }
9
10 int main ()
11 {
12     const char * c = "sample text";
13     print ( const_cast<char *> (c) );
14     return 0;
15 }
16
17 // 输出结果: sample text

```

### 1.10.4 reinterpret\_cast<type> (expr)

1. `reinterpret_cast` 是从底层对数据进行重新解释，依赖具体的平台，可移植性差。

2. `reinterpret_cast` 可以将整型转换为指针，也可以把指针转换为数组。

3. `reinterpret_cast` 可以在指针和引用里进行肆无忌惮的转换。

### 1.10.5 使用 `stringstream` 转换类型

```
#include <sstream>
```

`sstream` 头文件定义了三个类型来支持内存 IO: `istringstream`, `ostringstream`, `stringstream`。这些类型可以向 `string` 写入数据，或从 `string` 读取数据。

`stringstream` 的一些操作：

- `stringstream strm;` // `strm` 是一个未绑定的 `stringstream` 类型
- `stringstream strm(s);` // `strm` 是一个 `stringstream` 对象，保存 `string s` 的一个拷贝
- `strm.str();` // 返回 `strm` 保存的拷贝
- `strm(s);` // 将 `string s` 拷贝到 `strm` 中，返回 `void`

### 强制类型转换

```
1  #include <iostream>
2  #include <sstream>
3  using namespace std;
4
5  template <class output_type, class input_type>
6  output_type Convert(const input_type &input)
7  {
8      stringstream strm;
9      strm << input;
10     output_type result;
11     strm >> result;
12     strm.clear();
13     return result;
14 }
15
16
17 int main(int argc, char ** argv)
18 {
19     string strNum = "-22.22";
20     float f = Convert<float>(strNum);
21     cout << f << endl; // -22.22
22
23     float n = 22.22;
```

(continues on next page)

(continued from previous page)

```
24 string str = Convert<string>(n);
25 cout << str << endl; // 22.22
26
27 return 0;
28 }
```

**Note:** strm 调用 **成对的** << 和 >> 之后, 状态为 `end-of-file`, 必须进行 `clear` 才能进行下一次 << 操作。  
`strm.clear()` 重置了 strm 的状态标识, 并没有清空数据。如果没有调用 << 之后没有使用 >>, 可以使用 `strm.str("")` 清空数据。

### 1.10.6 参考资料

1. C++ 中四种强制类型转换区别详解  
<https://www.cnblogs.com/cauchy007/p/4968707.html>
2. c++ 四种强制类型转换介绍  
<https://blog.csdn.net/ydar95/article/details/69822540>
3. C++ 中使用 stringstream 简化类型转换  
<https://www.cnblogs.com/Mr-Zhong/p/5312478.html>
4. c++ reference  
<http://www.cplusplus.com/reference/sstream/stringstream>  
<http://www.cplusplus.com/doc/tutorial/typecasting/>
5. C++ 强制类型转换操作符 `const_cast`  
<https://www.cnblogs.com/QG-whz/p/4513136.html>

## 1.11 堆、栈

堆 (Heap) 与栈 (Stack) 有两层含义: (1) 程序内存布局场景下, 堆与栈表示的是两种内存管理方式; (2) 数据结构场景下, 堆与栈表示两种常用的数据结构。

栈由操作系统自动分配释放, 用于存放函数的参数值、局部变量等, 其操作方式类似于数据结构中的栈。

堆由程序员分配释放, 若程序员不释放, 程序结束时由系统回收。

### 1.11.1 管理方式

栈由操作系统自动分配释放，无需我们手动控制；

堆的申请和释放工作由程序员控制，容易产生内存泄漏。

### 1.11.2 空间大小

每个进程拥有的栈的大小要远远小于堆的大小。理论上，程序员可申请的堆大小为虚拟内存的大小，进程栈的大小 64bits 的 Windows 默认 1MB，64bits 的 Linux 默认 10MB；

### 1.11.3 分配方式

堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是由操作系统完成的，比如局部变量的分配。

动态分配由 `alloca` 函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由操作系统进行释放，无需我们手工实现。

### 1.11.4 生长方式

堆的生长方向向上，内存地址由低到高。

栈的生长方向向下，内存地址由高到低。

### 1.11.5 分配效率

栈由操作系统自动分配，会在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。

堆则是由 C/C++ 提供的库函数或运算符来完成申请与管理，实现机制较为复杂，频繁的内存申请容易产生内存碎片。显然，堆的效率比栈要低得多。

### 1.11.6 存放内容

栈存放函数返回地址、相关参数、局部变量和寄存器内容等。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

### 1.11.7 附：内存分区

在 C++ 中，内存主要分为堆、栈、全局/静态存储区和常量存储区。



- **栈**：就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的存储区。里面的变量通常是局部变量、函数参数等。
- **堆**：就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。
- **全局/静态存储区**：全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量又分为初始化的和未初始化的，在 C++ 里面没有这个区分了，他们共同占用同一块内存区。
- **常量存储区**：这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。

### 1.11.8 参考资料

1. 堆与栈的区别

<https://blog.csdn.net/K346K346/article/details/80849966>

2. C/C++——堆栈的讲解

<https://blog.csdn.net/lovejay7/article/details/80662390>

3. C++ 自由存储区是否等价于堆？

<https://www.cnblogs.com/QG-whz/p/5060894.html>

## 1.12 参数传递

当形参是引用类型时，称对应实参被 **引用传递** (passed by reference) 或者函数被 **传引用调用** (called by reference)。

当实参的值被 **拷贝** 给形参时，形参和实参时两个相互独立的对象。这样的实参被 **值传递** (passed by value) 或者函数被 **传值调用** (called by value)。

### 1.12.1 传值参数

当初始化一个非引用类型的变量时，初始化被拷贝给变量。此时，对变量的改动 **不会** 影响初始值。

**指针形参** 指针的行为和其他 **非引用类型** 一样。当执行指针拷贝操作时，拷贝的是指针的值。拷贝之后，**两个指针是不同的指针**。因为指针使我们可以间接地访问它所指向的对象，所以通过指针 **可以修改它所指向的对象的值**。

```

1 void reset(int* p)
2 {
3     *p = 0; // 改变了指针 p 所指对象的值
4     p = 0; // 只改变了 p 的局部拷贝，实参未被改变
5 }
```

(continues on next page)

(continued from previous page)

```
6
7 template<class T>
8 void swap(T* x, T* y)
9 {
10     T* tmp = x;
11     x = y;
12     y = tmp;
13 }
14 // 只交换了拷贝指针的值，实际指针并未改变，因此无法达到交换的目的。
```

为了改变实参指针的值，可以使用指针的引用或者使用指向指针的指针。

```
int val = 1;
int* p = &val;

// 调用: reset(p)
void reset(int* &p)
{
    *p = 0; // 改变了指针 p 所指对象的值
    p = 0; // 改变了指针 p 的值
}

// 调用: reset(&p)
void reset(int** p)
{
    **p = 0; // 改变了指针 *p 所指对象的值
    *p = 0; // 改变了指针 *p 的值
}
```

### 1.12.2 传引用参数

通过使用引用形参，允许函数改变实参的值。

使用引用避免拷贝：拷贝大的类类型对象或者容器对象比较低效，甚至有的类类型（比如 IO 类型）根本不支持拷贝操作。当某种类型不支持拷贝操作时，函数只能通过引用形参访问该类型的对象。例如，需要比较两个 string 对象，而这样的对象可能会很长，为了避免拷贝且不改变对象的值，可以将形参声明为常量引用（const &）。

使用引用形参返回额外信息：通过给函数传入一个额外的引用形参，让其保存需要的值，而不需要作为函数返回值返回（避免函数返回值太多）。

### 1.12.3 参考资料

1. 《C++ Primer 第 5 版中文版》Page 187 – 190。

## 1.13 空类指针

类的成员函数并不与特定对象绑定，所有成员函数共用一份成员函数体，当程序编译后，成员函数的地址即已经确定。那为什么同一个类的不同对象调用对应成员函数可以出现不同的结果呢？答案就是 `this` 指针。共有的成员函数体之所以能够把不同对象的数据区分开来，靠的是隐式传递给成员函数的 `this` 指针，成员函数中对成员变量的访问都是转化成 “`this->数据成员`” 的方式。因此，从这一角度说，成员函数与普通函数一样，只是多了一个隐式参数，即指向对象的 `this` 指针。

### 1.13.1 空类指针调用成员函数

```
1  class TestNullPtr {
2  public:
3      void print() {
4          cout << "print" << endl;
5      }
6
7      void getA() {
8          cout << a << endl;
9      }
10
11     void setA(int x) {
12         a = x;
13     }
14
15     virtual test() {
16         cout << "test" << endl;
17     }
18
19 private:
20     int a = 100;
21 };
22
23 TestNullPtr* ptr = nullptr;
24 ptr->print(); // 运行成功
25 ptr->getA(); // 编译成功，运行失败
26 ptr->setA(); // 编译成功，运行失败
27 ptr->test(); // 编译成功，运行失败
```

上例中，`ptr->getA()` 和 `ptr->setA()` 都试图访问成员变量，然而 `this` 指针为空，导致运行失败。另外，虚函数的特性是动态绑定，运行时根据指针或引用绑定的对象是基类对象还是派生类对象调用相关函数，空指针显然会导致错误。

### 1.13.2 参考资料

1. C++ 空指针调用成员函数

<https://www.jianshu.com/p/45cf10150e6b>

## 1.14 static 和 extern

### 1.14.1 static: 静态全局变量

在全局变量前，加上关键字 `static`，该变量就被定义成为一个静态全局变量。特点：

- 该变量在 **全局数据区**分配内存；
- 未经初始化的静态全局变量会被程序自动初始化为 0；（自动变量的值是随机的，除非它被显式初始化）
- 静态全局变量在声明它的整个文件都是可见的，而在 **文件之外是不可见的**，其它文件中可以定义相同名字的变量，不会发生冲突。

### 1.14.2 static: 静态函数

在函数的返回类型前加上 `static` 关键字，函数即被定义为静态函数。静态函数与普通函数不同，它 **只能在声明它的文件当中可见**，不能被其它文件使用。其它文件中可以定义相同名字的函数，不会发生冲突。这点与静态全局变量相似。

### 1.14.3 static: 静态局部变量

在局部变量前，加上关键字 `static`，该变量就被定义成为一个静态局部变量。特点：

- 该变量在全局数据区分配内存；
- 静态局部变量在程序执行到该对象的声明处时被首次初始化，即 **以后的函数调用不再进行初始化**；
- 静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；
- 它始终驻留在全局数据区，其生命周期一直持续到整个程序执行结束。但其作用域仍为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束。一般情况下，对于局部变量是存放在栈区的，并且局部变量的生命周期在该语句块执行结束时便结束了。

```

1 void func()
2 {
3     static int a = 1; // 初次调用 func() 时才会执行初始化
4     cout << a << endl;
5     a ++;
6 }
7
8 int main()
9 {
10    func(); // 1
11    func(); // 2
12    return 0;
13 }

```

#### 1.14.4 static: 静态成员变量

在类内数据成员的声明前加上关键字 `static`，该数据成员就是类内的静态数据成员。特点：

- 对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当作是类的成员。无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份拷贝，由该类型的所有对象共享访问。在没有产生类的实例时，我们就可以操作它；
- 静态数据成员存储在全局数据区。静态数据成员定义时要分配空间，所以不能在类声明中定义；
- 静态数据成员和普通数据成员一样遵从 `public`，`protected`，`private` 访问规则；
- (类定义体外部) 静态数据成员初始化与一般数据成员初始化不同。静态数据成员初始化的格式为：<数据类型> <类名>:: <静态数据成员名> = <值>
- 类的静态数据成员有两种访问形式：<类对象名> . <静态数据成员名> 或 <类类型名>:: <静态数据成员名>

#### 1.14.5 static: 静态成员函数

普通的成员函数一般都隐含了一个 `this` 指针，`this` 指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，`this` 是缺省的。如函数 `fn()` 实际上是 `this->fn()`。但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它 **不具有 `this` 指针**。从这个意义上讲，它 **无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数**。非静态成员函数可以任意地访问静态成员函数和静态数据成员。

```

1 class MyClass
2 {
3     private:
4         int a , b , c;

```

(continues on next page)

(continued from previous page)

```

5     static int sum; //声明静态数据成员
6 public:
7     MyClass(int a , int b , int c);
8     void GetSum();
9 };
10
11 int MyClass::sum = 0; //定义并初始化静态数据成员
12
13 MyClass::MyClass(int a , int b , int c)
14 {
15     this->a = a;
16     this->b = b;
17     this->c = c;
18     sum += a+b+c;
19 }
20 void MyClass::GetSum()
21 {
22     cout<<"sum="<<sum<<endl;
23 }

```

### 1.14.6 extern: 修饰函数、变量

修饰符 `extern` 用在变量或者函数的声明前，用来说明 “此变量/函数是在别处定义的，要在此处引用” 。在别的文件中如果想调用 `file1.c` 中的变量 `a`，只须用 `extern` 进行声明即可调用 `a`：

```

extern int a; // file2.c
extern "C" int a; // file3.cpp

```

在这里要注意 `extern` 声明的位置对其作用域也有关系，如果是在 `main` 函数中进行声明的，则只能在 `main` 函数中调用，在其它函数中不能调用。其实要调用其它文件中的函数和变量，只需把该文件用 `#include` 包含进来即可，但是用 `extern` 会加速程序的编译过程，这样能节省时间。

### 1.14.7 extern “C” {}

#### 例子

```

1  #ifndef HEADER_INCLUDED // 条件编译，避免重复包含头文件
2  #define HEADER_INCLUDED
3
4  #ifndef __cplusplus // extern "C" 只用在 c++ 文件中
5  extern "C" {

```

(continues on next page)

(continued from previous page)

```

6  #endif /* __cplusplus */
7
8  #include "c.h"
9
10 char* strcpy(char*, const char*);
11
12 /*.....
13  * do something else
14  *.....
15  */
16
17 #ifdef __cplusplus
18 }
19 #endif /* __cplusplus */
20
21 #endif /* HEADER_INCLUDED */

```

`extern "C"` 中的 `C`，表示的一种编译和连接规约，表明它按照类 `C` 的编译和连接规约来编译和连接，而不是一种语言。`C` 表示符合 `C` 语言的编译和连接规约的任何语言，如 Fortran、assembler 等。`extern "C"` 的真实目的是实现类 `C` 和 `C++` 的混合编程。

### 1.14.8 参考资料

1. C/C++ 中的 static 关键字详解

<https://www.cnblogs.com/qintangtao/p/3285937.html>

2. C++ 项目中的 `extern "C" {}`

<https://www.cnblogs.com/skynet/archive/2010/07/10/1774964.html>

3. 浅谈 C/C++ 中的 static 和 extern 关键字

<https://www.cnblogs.com/dolphin0520/archive/2011/04/20/2022701.html>

## 1.15 public、protected、private

Table 2: 类成员访问权限（可访问：✓）

权限	类成员	类对象	派生类成员	友元函数
public	✓	✓	✓	✓
private	✓	×	×	✓
protected	✓	×	✓	✓

### 1.15.1 继承

Table 3: public 继承下对基类成员的访问权限

内部权限	权限变化 (相对于派生类)	派生类成员	派生类对象
public	-> public	✓	✓
private	-> private	×	×
protected	-> protected	✓	×

Table 4: private 继承下对基类成员的访问权限

内部权限	权限变化 (相对于派生类)	派生类成员	派生类对象
public	-> private	✓	×
private	-> private	×	×
protected	-> private	✓	×

Table 5: protected 继承下对基类成员的访问权限

内部权限	权限变化 (相对于派生类)	派生类成员	派生类对象
public	-> protected	✓	×
private	-> private	×	×
protected	-> protected	✓	×

### 1.15.2 class 与 struct

class 不写权限修饰符, 成员默认是 `private`, 而 `struct` 的成员默认是 `public`。

class 的继承默认是 `private` 的, 而 `struct` 默认是 `public` 的。

### 1.15.3 参考资料

1. C++ 中关于 `public`、`protect`、`private` 的访问权限控制

<https://blog.csdn.net/ycf74514/article/details/49053041>

2. C++ 的关键字 `public`、`private` 和 `protected`

<https://www.jianshu.com/p/943c0d2fe292>

3. C++ 中 `public`、`protected`、`private` 访问

<https://www.cnblogs.com/jiudianren/p/5668438.html>



## 1.16 类的 static const 成员

### 1.16.1 const

const 数据成员的初始化只能在类的构造函数的初始化列表中进行。声明 const 变量是不能初始化。

**Note:** 必须使用初始化列表的情形

- **常量成员**，因为常量只能初始化不能赋值，所以必须放在初始化列表里面。
- **引用类型**，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面。
- 没有默认构造函数的 **类类型**，因为使用初始化列表不调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化。

**Warning:** 成员是按照他们在类中 **声明**的顺序进行初始化的，而不是按照他们在初始化列表出现的顺序初始化的。

### 1.16.2 static

不能在定义对象时对变量进行 **定义和初始化**，即不能用构造函数进行初始化。初始化在类体外进行，前面不加 static 修饰符。

**Note:** const 全局变量和类 static 成员变量都可以在 **定义类的头文件**中初始化。不会因为头文件重复包含而导致变量重复定义的编译错误。

但是在定义指针时要注意，`const char* p = "name"` 定义的指针不是 const，可能导致错误；而 `char* const p = "name"` 不会。

### 1.16.3 static const

静态常量成员，可以直接初始化（static const 和 const static 含义相同）。

```

1 // header.h
2 class Solution {
3 public:
4     static void print()
5     {
6         cout << var << endl; // 100

```

(continues on next page)

(continued from previous page)

```
7         cout << (mapping.begin()->second)[0] << endl; // 'a'
8     }
9
10 private:
11     static const map<char, vector<char> > mapping; // 常量声明式
12     static const int var = 100; // 常量声明式（直接初始化）
13 };
14
15 // source.cpp
16 const map<char, vector<char> > Solution:: mapping = {{'2', {'a', 'b', 'c'}},
17                                                     {'3', {'d', 'e', 'f'}},
18                                                     {'4', {'g', 'h', 'i'}}}; // mapping 的定义
19
20 /* 注: const map 只能通过迭代器 const_iterator 访问元素 (it->first, it->second)
21    , 不能通过下标 [] 的方式访问。*/
22
23 // 对应类的 static const int/char/bool 成员常量, 如果不取他们的地址, 则可以直接声明并使用, 而无需提供以
24 // 下的定义式。
25 const int Solution:: var; // var 的定义。由于 常量 var 在类内声明时已经获得了初始值, 因此定义时不可以
26 // 再设初始值。
```

**Note: 初始化:** 变量还没有值, 现在赋予它一个值。

**赋值:** 变量已经有一个值, 现在擦除它之前的值, 赋予一个新的值。

## 1.16.4 参考资料

1. C++ static、const 和 static const 类型成员变量声明以及初始化

<https://www.cnblogs.com/hustfeiji/articles/5168529.html>

2. 头文件中定义 const 全局变量应注意的问题

<https://blog.csdn.net/mafuli007/article/details/8499585>

3. C++ 初始化列表

<https://www.cnblogs.com/graphics/archive/2010/07/04/1770900.html>

4. C++ 的一大误区——深入解释直接初始化与复制初始化的区别

<https://blog.csdn.net/ljianhui/article/details/9245661>

5. C++ 构造函数初始化列表与赋值

<https://www.cnblogs.com/sz-leez/p/7082865.html>

<http://www.cnblogs.com/BlueTzar/articles/1223169.html>

## 1.17 枚举类型与共用体

### 1.17.1 枚举类型

枚举类型 (enumeration) 使我们可以将一组 **整型常量** 组织在一起。格式

```
enum <类型名> {<枚举成员>;
```

枚举成员不能是数值，即不能是类似于 {1,2,3}。

#### 初始化

默认情况下，每个枚举变量的值就是其序号，从 0 开始，依次加 1。

```
enum Week {Sun, Mon, Tue, Wed, Thu, Fri, Sat};  
//Sun=0, Mon=1, Tue=2, Wed=3, Thu=4, Fri=5, Sat=6
```

显式提供初始值。

```
enum Week {Sun=1, Mon, Tue, Wed, Thu, Fri=100, Sat};  
//Sun=1, Mon=2, Tue=3, Wed=4, Thu=5, Fri=100, Sat=101
```

#### 枚举变量

定义枚举类型之后，就可以定义该枚举类型的变量，或者与枚举类型同时定义。枚举变量的值只能取枚举常量表中所列的值。

```
1  enum Week {Sun=1, Mon, Tue, Wed, Thu, Fri=100, Sat} day_1;  
2  // 全局变量 day_1 默认初始化为 0。  
3  
4  void ff(int num)  
5  {  
6      cout << -10 * num << endl;  
7  }  
8  void ff(Week day)  
9  {  
10     cout << static_cast<float>(10 * day) << endl;  
11 }  
12  
13 int main(int argc, char** argv)
```

(continues on next page)

(continued from previous page)

```

14 {
15     Week day_2, day_3;
16
17     day_1 = Sun; // 或者 day_1 = Week::Sun (不限定作用域的枚举类型)
18     day_2 = day_1;
19     int i = day_1; // i = 1
20     int j = Mon; // j = 2
21
22     Week day_4(Week(100)); // day_4 = Fri
23     bool equal = (day_4 == Fri); // true
24
25     Week day_5(Week(-1)); // 越界，但是不报错，day_5 = -1 (VS 2013)
26
27     ff(1); // 匹配 ff(int)，输出 -10
28     ff(day_1); // 匹配 ff(Week)，输出 10
29     ff(Fri); // 匹配 ff(Week)，输出 1000
30
31     return 0;
32 }

```

### 1.17.2 共用体

共用体（union）及其变量的定义形式与结构体类似。共用体成员访问方式也是使用运算符 `.` 或 `->`。

```

union ifcd
{
    int i;
    float f;
    char c;
    double d;
} x1, x2[5], *pu;
// 同结构体一样，分号不能丢

```

与结构体的异同：

- 存储分配方式
  - 结构体每个成员各自占有自己的存储单元、各自的地址，结构体占有的内存空间大小是所有成员所占存储单元的总和。
  - 共用体各个成员占用共同的存储单元，具有 **相同的首地址**，占用存储单元最多的成员的长度就是共用体的长度。一个共用体可以包含多个不同类型的成员，但是每一时刻只有 **一个成员有效**，即最后一次存放的数据成员起作用。虽然仍然可以访问无效的成员，但是结果是未知的。

```
x1.i = 256;
x1.c = 'A';
x1.f = 1.23;
// 三步操作之后, 只有 x1.f 有效。
```

- 初始化
  - 结构变量或数组可以为所有成员初始化。
  - 共用体变量或数组在初始化时, 只能对初始化它的 **第一个成员**, 对多个成员初始化是不允许的。

```
union ifcd x3 = {256}, x4[3] = {256, 256, 256}; // 对成员 i 初始化
```

- 结构体和共用体可以相互出现在对方的类型定义中。

### 1.17.3 参考资料

1. 《C++ Primer 第 5 版中文版》Page 736 – 739。
2. C++ 枚举类型详解  
<http://www.runoob.com/w3cnote/cpp-enum-intro.html>
3. C++ 枚举 (enum) 的优雅用法  
<https://blog.csdn.net/daizhiyan1/article/details/82428023>

## 1.18 常用函数

### 1.18.1 lower\_bound, upper\_bound

```
#include <algorithm>
```

**lower\_bound** 从排好序的数组区间 `[first,last)` 中, 采用二分查找, 返回 **大于或等于** `val` 的 **第一个元素** 位置。如果所有元素都小于 `val`, 则返回 `last`。

```
template <class ForwardIterator, class T>
ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& val);
```

**upper\_bound** 从排好序的数组区间 `[first,last)` 中, 采用二分查找, 返回 **大于** `val` 的 **第一个元素** 位置。如果所有元素都不大于 `val`, 则返回 `last`。

```
template <class ForwardIterator, class T>
ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last, const T& val);
```

求有序数组中 val 的个数:

```
upper_bound(first, last, val) - lower_bound(first, last, val);
```

*Example*

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  int main(int argc, char ** argv)
7  {
8      int a[11] = {1,2,3,4,5,5,5,5,6,7,8};
9      cout << lower_bound(a, a + 11, 5) - a << endl; // 4
10     cout << upper_bound(a, a + 11, 5) - a << endl; // 8
11     vector<int> v(a, a + 11); // 用 a 对 v 初始化
12     cout << lower_bound(v.begin(), v.end(), 5) - v.begin() << endl; // 4
13     cout << upper_bound(v.begin(), v.end(), 5) - v.begin() << endl; // 8
14
15     *lower_bound(a, a + 11, 5) = 500;
16     for (auto ai : a) cout << ai << ends; // 1 2 3 4 500 5 5 5 6 7 8
17     cout << endl;
18
19     *lower_bound(v.begin(), v.end(), 5) = 500;
20     for (auto vi : v) cout << vi << ends; // 1 2 3 4 500 5 5 5 6 7 8
21     cout << endl;
22
23     return 0;
24 }
```

### 1.18.2 fill, fill\_n, for\_each

```
#include <algorithm>
```

fill 函数将一个区间 [first,last) 的每个元素都赋予 val 值。

```
template <class ForwardIterator, class T>
void fill (ForwardIterator first, ForwardIterator last, const T& val);
```

fill\_n 函数从 first 开始依次赋予 n 个元素 val 值。

```
template <class OutputIterator, class Size, class T>
void fill_n (OutputIterator first, Size n, const T& val);
```

`for_each` 把函数 `fn` 应用于区间 `[first,last)` 的每个元素。

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function fn);
```

*Example*

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5
6  template<class T>
7  void print(T elem){ cout << elem << " "; }
8
9  int main(int argc, char ** argv)
10 {
11
12     float a[4] = { 0.0 }; // {0.0, 0.0, 0.0, 0.0}
13     vector<int> v(4, 0); // {0, 0, 0, 0}
14
15     fill(a, a+4, 3.3); // {3.3, 3.3, 3.3, 3.3}
16     fill_n(a, 2, 6.6); // {6.6, 6.6, 3.3, 3.3}
17     fill_n(v.begin(), 4, 9); // {9, 9, 9, 9}
18
19     for_each(a, a + 4, print<float>); // 6.6 6.6 3.3 3.3
20     cout << endl;
21     for_each(v.begin(), v.end(), print<int>); // 9 9 9 9
22     cout << endl;
23
24     int b[10][20];
25     fill(b[0], b[0] + 200, 2); // b 所有元素为 2
26
27     return 0;
28 }
```

最长上升子序列：

```
1  /* https://leetcode.com/problems/longest-increasing-subsequence/ */
2  /* O(nlogn) in time.*/
3
4  class Solution {
5  public:
6      int lengthOfLIS(vector<int>& nums) {
7          if(nums.size()<=1) return nums.size();
8          int inf = INT_MAX;
9          int len = nums.size();
10         int* dp = new int[len];
11         fill(dp, dp+len, inf);
12         for(int k = 0; k < len; ++k) *lower_bound(dp, dp+len, nums[k]) = nums[k];
13         int length = lower_bound(dp, dp+len, inf) - dp;
14         delete[] dp;
15         return length;
16     }
17 };
```

### 1.18.3 sort

```
#include <algorithm>

// default
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);

// custom
template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

#### Example

```
1  #include <iostream>
2  #include <vector>
3  #include <functional>
4  #include <algorithm>
5  using namespace std;
6
7  bool comparator(int i, int j)
8  {
9      return (i < j);
10 }
11
```

(continues on next page)



(continued from previous page)

```

12 struct myclass
13 {
14     bool operator() (int i, int j)
15     {
16         return (i < j);
17     }
18 } myobject;
19
20 int main(int argc, char ** argv)
21 {
22
23     int a[] = { 32, 71, 12, 45, 26, 80, 53, 33 };
24     vector<int> v(a, a + 8);           // 32 71 12 45 26 80 53 33
25
26     // using default comparison (operator <):
27     sort(v.begin(), v.begin() + 4);    //(12 32 45 71)26 80 53 33
28
29     // using comparator as comp
30     sort(v.begin() + 4, v.end(), comparator); // 12 32 45 71(26 33 53 80)
31
32     // using object as comp
33     sort(v.begin(), v.end(), myobject);  //(12 26 32 33 45 53 71 80)
34
35     // using build-in comp: greater
36     sort(v.begin(), v.end(), greater<int>()); // (80 71 53 45 33 32 26 12)
37
38     // using build-in comp: less
39     sort(v.begin(), v.end(), less<int>());  //(12 26 32 33 45 53 71 80)
40
41     // using reverse_iterator
42     sort(v.rbegin(), v.rend()); // (80 71 53 45 33 32 26 12)
43
44     // sort array
45     sort(a, a + 8, greater<int>()); // (80 71 53 45 33 32 26 12)
46     sort(a, a + 8);                // (12 26 32 33 45 53 71 80), 可使用 comparator、myobject、less
47     ↪<int>()
48
49     return 0;
50 }

```

`string` 类也是可以排序的，如

```
string str;
sort(str.begin(), str.end());
```

**Warning:** 如果把自定义的 `comparator` 函数封装为类的成员函数，应该定义为 **静态成员函数 (static)**。

#### 1.18.4 `min_element`, `max_element`, `minmax_element`

```
#include <algorithm>
```

- `min_element` : 会返回指向输入序列的最小元素的迭代器；
- `max_element` : 会返回指向最大元素的迭代器；
- `minmax_element` : 会以 `pair` 对象的形式返回这两个迭代器。`first` 指向最小元素。`second` 指向最大元素。

`min_element`:

```
// default
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

// custom
template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last, Compare comp); // [first,
↪last)
```

*Example*

```
1  #include <iostream>
2  #include <algorithm>
3  using namespace std;
4
5  int main(int argc, char ** argv)
6  {
7
8      int a[10] = { 1, 2, 3, 4, 5, 6 };
9      cout << a[9] << endl; // 0
10
11     cout << *min_element(a, a + 10) << endl; // 0
```

(continues on next page)

(continued from previous page)

```

12
13     cout << *max_element(a, a + 10) << endl; // 6
14
15     auto p = minmax_element(a, a + 10);
16
17     cout << *p.first << ends << *p.second << endl; // 0 6
18
19     return 0;
20 }

```

### 1.18.5 accumulate

```

#include <numeric>

// sum
template <class InputIterator, class T>
T accumulate (InputIterator first, InputIterator last, T init);

// custom
template <class InputIterator, class T, class BinaryOperation>
T accumulate (InputIterator first, InputIterator last, T init, BinaryOperation binary_op);

```

累加区间 `[first,last)` 的元素，并加上 `init`。

*Example*

```

1  #include <iostream>      // std::cout
2  #include <functional>    // std::minus
3  #include <numeric>      // std::accumulate
4
5  int myfunction (int x, int y) {return x+2*y;}
6
7  struct myclass
8  {
9      int operator()(int x, int y) {return x+3*y;}
10 } myobject;
11
12 int main ()
13 {

```

(continues on next page)

(continued from previous page)

```

14  int init = 100;
15  int numbers[] = {10,20,30};
16
17  std::cout << "using default accumulate: ";
18  std::cout << std::accumulate(numbers,numbers+3,init); // 100 + (10 + 20 + 30)
19  std::cout << '\n';
20
21  std::cout << "using functional's minus: ";
22  std::cout << std::accumulate (numbers, numbers+3, init, std::minus<int>()); // 100 - (10 + 20 + 30)
23  std::cout << '\n';
24
25  std::cout << "using custom function: ";
26  std::cout << std::accumulate (numbers, numbers+3, init, myfunction); // 100 + 2 * (10 + 20 + 30)
27  std::cout << '\n';
28
29  std::cout << "using custom object: ";
30  std::cout << std::accumulate (numbers, numbers+3, init, myobject); // 100 + 3 * (10 + 20 + 30)
31  std::cout << '\n';
32
33  return 0;
34 }

```

### 1.18.6 partial\_sum

```
#include <numeric>
```

累加，并把结果存到序列（数组、向量）**result** 中。

```

// sum
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum (InputIterator first, InputIterator last, OutputIterator result);

// custom
template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum (InputIterator first, InputIterator last,
                           OutputIterator result,
                           BinaryOperation binary_op);

```

(continues on next page)

(continued from previous page)

```
// y0 = x0
// y1 = x0 + x1
// y2 = x0 + x1 + x2
// y3 = x0 + x1 + x2 + x3
// y4 = x0 + x1 + x2 + x3 + x4
// ... ..
```

*Example*

```
1  #include <iostream>      // std::cout
2  #include <functional>    // std::multiplies
3  #include <numeric>       // std::partial_sum
4  #include <vector>
5
6  int myop (int x, int y) {return x+y+1;}
7
8  int main ()
9  {
10     int val[] = {1,2,3,4,5};
11     int result[5];
12
13     std::partial_sum (val, val+5, result);
14     std::cout << "using default partial_sum: ";
15     for (int i=0; i<5; i++) std::cout << result[i] << ' '; // 1 3 6 10 15
16     std::cout << '\n';
17
18     std::vector<int> result_vec(6, 0); // 0 0 0 0 0 0
19     std::partial_sum (val, val+5, result_vec.begin()); // 1 3 6 10 15 0
20
21     std::partial_sum (val, val+5, result, std::multiplies<int>()); // 1 2 6 24 120
22     std::cout << "using functional operation multiplies: ";
23     for (int i=0; i<5; i++) std::cout << result[i] << ' ';
24     std::cout << '\n';
25
26     std::partial_sum (val, val+5, result, myop); // 1 4 8 13 19
27     std::cout << "using custom function: ";
28     for (int i=0; i<5; i++) std::cout << result[i] << ' ';
29     std::cout << '\n';
30     return 0;
31 }
```

### 1.18.7 iota

```
#include <numeric>

template <class ForwardIterator, class T>
void iota (ForwardIterator first, ForwardIterator last, T val);
```

采用递增的形式，将 val 开始的等差数列赋值给区间 [first,last) 的元素。

*Example*

```
1  #include <iostream>      // std::cout
2  #include <numeric>      // std::iota
3
4  int main () {
5      float numbers[10];
6
7      std::iota (numbers,numbers+10,3.5);
8
9      std::cout << "numbers:";
10     for (float& i:numbers) std::cout << ' ' << i; // 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.5 11.5 12.5
11     std::cout << '\n';
12
13     return 0;
14 }
```

### 1.18.8 inner\_product

```
#include <numeric>

// sum/multiply
template <class InputIterator1, class InputIterator2, class T>
T inner_product (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init);

// custom
template <class InputIterator1, class InputIterator2, class T, class BinaryOperation1, class
↳ BinaryOperation2>
T inner_product (InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2,
                 T init,
```

(continues on next page)

(continued from previous page)

```
BinaryOperation1 binary_op1,
BinaryOperation2 binary_op2);
```

内积运算，再与 `init` 做运算:

```
while (first1!=last1)
{
    init = init + (*first1)*(*first2);
    // or: init = binary_op1 (init, binary_op2(*first1,*first2));
    ++first1; ++first2;
}
return init;
```

### Example

```
1  #include <iostream>      // std::cout
2  #include <functional>    // std::minus, std::divides
3  #include <numeric>       // std::inner_product
4
5  int myaccumulator (int x, int y) {return x-y;}
6  int myproduct (int x, int y) {return x+y;}
7
8  int main () {
9      int init = 100;
10     int series1[] = {10,20,30};
11     int series2[] = {1,2,3};
12
13     std::cout << "using default inner_product: ";
14     std::cout << std::inner_product(series1,series1+3,series2,init); // 100 + (10*1 + 20*2 + 30*3)
15     std::cout << '\n';
16
17     std::cout << "using functional operations: ";
18     std::cout << std::inner_product(series1,series1+3,series2,init,
19                                     std::minus<int>(),std::divides<int>()); // 100 - (10/1 + 20/2 +
↪30/3)
20     std::cout << '\n';
21
22     std::cout << "using custom functions: ";
23     std::cout << std::inner_product(series1,series1+3,series2,init,
24                                     myaccumulator,myproduct); // 100 - (10+1 + 20+2 + 30+3)
25     std::cout << '\n';
26
27     return 0;
28 }
```

## 1.18.9 memset

```
#include <cstring>

void * memset ( void * ptr, int value, size_t num );
```

memset 按 字节赋值, fill 按 元素赋值。

如果用 memset 给 int 型变量赋值, 只能是 0 或-1。

*Example*

```
1  #include <iostream>
2  #include <cstring>
3
4  int main()
5  {
6      char str[] = "almost every programmer should know memset!";
7      memset (str, '-',6);
8      cout << str << endl; // ----- every programmer should know memset!
9
10     int a[10][10];
11     memset(a, -1, sizeof(a)); // 或者 10*10*sizeof(int), 全部赋值为-1
12     for(int e:a) cout << bitset<sizeof(int)*8>(e) << endl; // 11111111 11111111 11111111 11111111
(补码)
13
14     int b[5];
15     memset(b, 1, sizeof(b)); // 或者 5*sizeof(int), 全部赋值为 16843009
16     for(int e:b) cout << bitset<sizeof(int)*8>(e) << endl; // 00000001 00000001 00000001 00000001
(int 型占 4 字节, 每个字节都赋值为 1)
17
18     return 0;
19 }
```

### 1.18.10 附: 头文件

- cmath
  - pow()



- `sqrt()`
  - `floor()`
  - `ceil()`
  - `round()`
- `cstdlib`
  - `abs()`
  - `fabs()`
- `limits`
  - `INT_MIN`
  - `INT_MAX`
- `algorithm`
  - `min()`
  - `max()`
- `utility`
  - `pair`
- `functional`
  - `less< TYPE >()`
  - `greater< TYPE >()`
- `cassert`
  - `assert()`

### 1.18.11 参考资料

#### 1. C++ reference

[http://www.cplusplus.com/reference/algorithm/lower\\_bound](http://www.cplusplus.com/reference/algorithm/lower_bound)

[http://www.cplusplus.com/reference/algorithm/upper\\_bound](http://www.cplusplus.com/reference/algorithm/upper_bound)

<http://www.cplusplus.com/reference/algorithm/fill>

[http://www.cplusplus.com/reference/algorithm/for\\_each](http://www.cplusplus.com/reference/algorithm/for_each)

<http://www.cplusplus.com/reference/algorithm/sort>

[http://www.cplusplus.com/reference/algorithm/min\\_element](http://www.cplusplus.com/reference/algorithm/min_element)

<http://www.cplusplus.com/reference/numeric/accumulate>

[http://www.cplusplus.com/reference/numeric/partial\\_sum](http://www.cplusplus.com/reference/numeric/partial_sum)

<http://www.cplusplus.com/reference/numeric/iota>

[http://www.cplusplus.com/reference/numeric/inner\\_product](http://www.cplusplus.com/reference/numeric/inner_product)

<http://www.cplusplus.com/reference/cstring/memset>

2. C/C++-STL 中 lower\_bound 与 upper\_bound 的用法

<https://blog.csdn.net/jadeyansir/article/details/77015626>

3. c++sort 函数的使用总结

<https://www.cnblogs.com/TX980502/p/8528840.html>

4. C++ sort 排序函数用法

[https://blog.csdn.net/w\\_linux/article/details/76222112](https://blog.csdn.net/w_linux/article/details/76222112)

## 1.19 常用 STL 类及容器

**STL: Standard Template Library** , 标准模板库。

### 顺序容器

- array
- vector
- list
- deque
- string
- ...

### 关联容器

- set
- map
- multiset
- multimap
- ...

### 容器适配器

- stack
- queue
- priority\_queue

### 1.19.1 string

```
#include<string>
```

- 长度: length(), size(), empty()
- 访问: [pos], at(pos)。at() 返回位置 pos 处元素的引用, 越界则抛出 out\_of\_range 异常。
- 字典序比较: ==, !=, <, <=, >, >=
- 串接: +
- c\_str(): 返回指向 C 类型字符串的指针。如果需要使用 C 的字符串函数如 strcmp、strcpy 等, 需要使用 c\_str()。

```
const char* c_str() const noexcept;
```

- 子串

```
string substr(size_t pos = 0, size_t len = npos) const
```

- 插入: 支持下标索引插入, 在位置 pos 之前插入元素。插入元素之后, 该元素的位置为 position。(与 python 中 list 类的插入功能一致)

```
// string (1)
string& insert (size_t pos, const string& str);
// substring (2)
string& insert (size_t pos, const string& str, size_t subpos, size_t sublen);
// c-string (3)
string& insert (size_t pos, const char* s);
// buffer (4)
string& insert (size_t pos, const char* s, size_t n);
// fill (5)
string& insert (size_t pos, size_t n, char c);
iterator insert (const_iterator p, size_t n, char c);
// single character (6)
iterator insert (const_iterator p, char c);
// range (7)
template <class InputIterator>
iterator insert (iterator p, InputIterator first, InputIterator last);
// initializer list (8)
string& insert (const_iterator p, initializer_list<char> il);
```

### 1.19.2 vector

```
#include<vector>
```

底层实现：顺序表（数组）。

- 元素个数：size(), empty()
- 逐元素比较：==, !=, <, <=, >, >=
- 内存空间：capacity()
- 访问：[pos], at(pos)
- 头部元素：front(), 返回的是引用
- 尾部元素：back(), 返回的是引用
- 尾部插入：push\_back(x)
- 尾部弹出：pop\_back()
- 迭代器插入：在 position 之前插入元素。

```
iterator insert (iterator position, const value_type& val)
```

- 尾部删除：pop\_back()
- 申请空间：至少能容纳 n 个元素。

```
void reserve (size_type n)
```

- 赋值
  - 数组或其他向量区间 [first,last) 内的值赋给当前向量。

```
template <class InputIterator>
void assign (InputIterator first, InputIterator last)
```

- 赋予 n 个 val 元素给当前向量。

```
void assign (size_type n, const value_type& val)
```

- 删除：删除一个元素之后，此位置之后所有元素往前移动。虽然当前迭代器没有 +1，但是由于后续元素的前移，相当于迭代器自动指向了下一个元素。故删除了一个元素之后如果要访问下一个元素，不必执行 it++。

```
iterator erase (const_iterator position);
iterator erase (const_iterator first, const_iterator last); // 区间 [first,last)
```

- 清除：

- `vector< value_type >().swap(myVec)`
- `myVec.clear()` 让 `myVec.size()` 为 0, 但是 `myVec.capacity()` 不为 0, 调用 `myVec.clear()` 之后再调用 `myVec.shrink_to_fit()`。 `shrink_to_fit()` 的作用是减小 `capacity()` 以匹配 `size()`。

### 1.19.3 list

```
#include<list>
```

底层实现：双向链表。

- 元素个数： `size()`, `empty()`
- 表首元素： `front()`
- 表尾元素： `back()`
- 插入： `push_front()`, `push_back()`, `emplace_front()`, `emplace_back()`
- 删除： `pop_front()`, `pop_back()`
- 迭代器插入：在 `position` **之前**插入元素。

```
iterator insert (iterator position, const value_type& val)
```

### 1.19.4 deque

```
#include<deque>
```

底层实现：循环队列。

- 元素个数： `size()`, `empty()`
- 队首元素： `front()`
- 队尾元素： `back()`
- 插入： `push_front(x)`, `push_back(x)`
- 删除： `pop_front()`, `pop_back()`
- 迭代器插入：在 `position` **之前**插入元素。

```
iterator insert (iterator position, const value_type& val)
```

**Note:** 顺序容器构造函数

- `C c; // 默认构造函数, 空容器`

- `C c1(c2);` // 拷贝构造函数
  - `C c(it_begin, it_end);` // 迭代器指定的范围 `[it_begin, it_end)` 内的元素赋值给 `c` (`array` 不支持)
  - `C c{a, b, c, ...};` // 列表初始化
- 

### 1.19.5 pair

```
#include<utility>
```

- 构造

```
template <class T1, class T2>
pair<T1,T2> make_pair (T1 x, T2 y);
```

- 访问：成员 `first` 访问第一个元素，成员 `second` 访问第二个元素。

### 1.19.6 map

```
#include<map>
```

底层实现：红黑树。

`map<K, T>` 容器，保存的是 `pair<const K, T>` 类型的元素。

`map<K, T>::key_type` : 键类型

`map<K, T>::mapped_type` : 值类型

`map<K, T>::value_type` : `pair` 类型, `<map<K, T>::key_type, map<K, T>::mapped_type>`

- 访问：`[key]`, `at(key)`
  - `[key]`, `key` 不存在，会创建新的键值对。
  - `at(key)`, `key` 不存在，抛出 `out_of_range` 异常。
- 查找：找不到 `key` 则返回 `map::end` 。

```
iterator find (const key_type& k);
const_iterator find (const key_type& k) const;
```

- 插入：如果 `key` 已经存在，则插入无效。`map` 的元素自动按照 `key` 升序排序，不能人为对 `map` 进行排序。

```
pair<iterator,bool> insert (const value_type& val);
```

- 删除：返回删除元素后的下一个元素的迭代器，当前迭代器失效。

```

iterator erase (const_iterator position);
size_type erase (const key_type& k);
iterator erase (const_iterator first, const_iterator last);

```

it = myMap.erase(it) 等效为 myMap.erase(it++) 。

例子

```

1  #include <iostream>
2  #include <map>
3
4  int main ()
5  {
6      std::map<char,int> mymap;
7
8      // first insert function version (single parameter):
9      mymap.insert ( std::pair<char,int>('a',100) );
10     mymap.insert ( std::map<char,int>::value_type('z',200) );
11
12     std::pair<std::map<char,int>::iterator,bool> ret;
13     ret = mymap.insert ( std::pair<char,int>('z',500) );
14     if (ret.second==false)
15     {
16         std::cout << "element 'z' already existed";
17         std::cout << " with a value of " << ret.first->second << '\n';
18     }
19
20     return 0;
21 }

```

### 1.19.7 stack

```
#include<stack>
```

- 大小: size(), empty()
- 栈顶元素: top()
- 入栈: push(x)
- 出栈: pop()

```
void pop();
```

### 1.19.8 queue

```
#include<queue>
```

- 大小: size(), empty()
- 队首元素: front()
- 队尾元素: back()
- 入队: push(x)
- 出队: pop()

```
void pop();
```

### 1.19.9 to\_string 函数

```
#include <string>
```

把 val 转化为字符串。

```
string to_string (int val);  
string to_string (long val);  
string to_string (long long val);  
string to_string (unsigned val);  
string to_string (unsigned long val);  
string to_string (unsigned long long val);  
string to_string (float val);  
string to_string (double val);  
string to_string (long double val);
```

### 1.19.10 atoi, atof, atol

```
#include <cstdlib>
```

把 C 类型的字符串转换为数字 (C++ 的 string 需要使用 c\_str() 转换)。

```
int atoi (const char * str);  
double atof (const char* str);  
long int atol ( const char * str );
```



### 1.19.11 size\_t 和 size\_type

#### size\_t

size\_t 提供了一种可移植（不同平台下）的方法声明与系统可寻址的内存区域一致的长度。size\_t 是通过 typedef 定义的一些 **无符号整型** 的别名，通常是 unsigned int，unsigned long 甚至是 unsigned long long。

常用于循环计数器、数组索引，或指针的算术运算。

VS 32 位编译器：sizeof(size\_t) = 32；VS 64 位编译器：sizeof(size\_t) = 64。

#### 头文件

- <cstddef>
- <cstdio>
- <cstring>
- <ctime>
- <cstdlib>
- <wchar>

#### size\_type

size\_type 是 STL 定义的类型属性，足够保持对应容器最大可能的容器大小。也是 **无符号整型**。

size() 的返回类型就是 size\_type。把 size() 赋值给一个 int 变量，会有 warning。

#### VS 32 位编译器

- sizeof(string::size\_type) = 32
- sizeof(vector<int>::size\_type) = 32
- ...

#### VS 64 位编译器

- sizeof(string::size\_type) = 64
- sizeof(vector<int>::size\_type) = 64
- ...

#### Warning:

无符号整型尤其是要注意下标为 0 时的边界情况。

```
1  vector<int> vec; // vec = {}
2  for(size_t k = 0; k < vec.size() - 1; ++k) // 判断改为: k + 1 < vec.size()
3  {
4      cout << vec[k+1] - vec[k] << endl;
5  }
```

上例中，本意是只有当 vec 至少包含 2 个元素时，才输出。但是，当 `vec.size() = 0`，`vec.size() - 1 = 232 - 1` 或 `264 - 1`，而不是预想的 -1。

### 1.19.12 参考资料

#### 1. C++ reference

<http://www.cplusplus.com/reference/string/string>  
<http://www.cplusplus.com/reference/vector/vector>  
<http://www.cplusplus.com/reference/list/list>  
<http://www.cplusplus.com/reference/deque/deque>  
<http://www.cplusplus.com/reference/map/map>  
<http://www.cplusplus.com/reference/stack/stack>  
<http://www.cplusplus.com/reference/queue/queue>  
[http://www.cplusplus.com/reference/string/to\\_string](http://www.cplusplus.com/reference/string/to_string)

#### 2. C++ STL 快速入门

<https://www.cnblogs.com/skyfsm/p/6934246.html>

#### 3. STL 教程：C++ STL 快速入门（非常详细）

<http://c.biancheng.net/stl/>

#### 4. 标准 C++ 中的 string 类的用法总结（转）

<https://www.cnblogs.com/aminxu/p/4686320.html>

#### 5. `std::size_t`

[https://zh.cppreference.com/w/cpp/types/size\\_t](https://zh.cppreference.com/w/cpp/types/size_t)

## 1.20 #define

### 原则

- 对于单纯常量，最好以 `const` 对象或 `enum` 替换 `#define`。

- 对于形似函数的宏 (macros)，最好改用 inline 函数替换 #define。

### 1.20.1 const

“宁可以编译器替换 预编译器”。

```
#define ASPECT_RATIO 1.653

const double AspectRatio = 1.653;
```

也许名称 ASPECT\_RATIO 从未被编译器看见，也许在编译器开始处理源码之前它已经被预处理器移走了。于是，记号名称 ASPECT\_RATIO 可能没有进入记号表 (symbol table) 内。当运用此常量获得编译错误时，这个错误也许会提到 1.653 而不是 ASPECT\_RATIO，导致对其追踪变得困难。

作为一个语言常量，AspectRatio 一定会被编译器看到并记入记号表。

此外，对浮点常量而言，使用常量可能比使用 #define 导致较小的代码量，因为预处理器将宏名称 ASPECT\_RATIO 替换为 1.653，可能导致目标码 (object code) 出现多份 1.653，使用常量则不会。

还可以在类定义 static const 成员。

```
1 class Player
2 {
3 private:
4     static const int NumTurns = 5; // 常量声明 (不是定义)
5     int scores[NumTurns]; // 使用该常量
6 }
```

### 1.20.2 enum

如果编译器不允许 static 成员在声明式上获得初始值，一方面，可以在头文件定义类，在源文件中初始化它；另一方面，如果该类在编译期间必须使用一个常量值，例如上例中数组 scores 的大小必须在编译期间知道，此时可以使用 enum。一个属于枚举类型的数值可以权当 int 被使用。

```
1 class Player
2 {
3 private:
4     enum {NumTurns = 5}; // NumTurns 成为数值 5 的一个记号。
5     int scores[NumTurns]; // 使用该常量
6 }
```

**Note:** enum 的行为类似于 #define，取一个 enum 的地址或 #define 的地址通常不合法，而取一个 const 的地址是合法的。

### 1.20.3 inline

使用宏 (macros) 不会有函数调用带来的额外开销。宏中的所有实参必须添加括号，但是仍然可能出现问题。此时，可以定义内联函数 (inline)，它与普通函数一样遵守作用域 (scope) 和访问规则。内联函数的特点：

- 在调用处直接展开该函数的内容
- 运行速度快，但占用更多内存
- 适用于规模小、流程直接（无递归和众多判断）、频繁调用的函数

普通函数的调用：

1. 执行到函数调用指令时，程序将立即存储该指令的内存地址，并将函数参数复制到栈（为此保留的内存块）；
2. 跳到标记函数起点的内存单元，执行函数代码（也许还需将返回值放入寄存器中）；
3. 然后跳回到地址被保存的指令处。

```
1  #include <iostream>
2  using namespace std;
3
4  #define CALL_WITH_MAX(a, b) f((a)>(b)? (a):(b))
5
6  #define MAX_COMP_1(a, b) (a)>(b)? (a):(b)
7
8  #define MAX_COMP_2(a, b) ((a)>(b)? (a):(b)) // 有外层括号
9
10 template<class T>
11 void f(T elem)
12 {
13     cout << "max out: " << elem << endl;
14 }
15
16 template<class T>
17 inline void CallWithMax(const T& a, const T& b) // 形参使用常量引用，因为不知道 T 的具体类型，比较安全
18 {
19     f(a > b ? a : b);
20 }
21
22 int main(int argc, char ** argv)
23 {
24     int a = 5, b = 0;
25     CALL_WITH_MAX(++a, b); // a 自增 2 次，变为 7 (++a > b => ++a)
26     cout << a << endl;
27     CALL_WITH_MAX(++a, b+10); // a 自增 1 次，变为 8 (++a < b => b)
```

(continues on next page)

(continued from previous page)

```

28     cout << a << endl;
29
30     f(-10 + MAX_COMP_1(a, b)); // -10 + a > b ? a : b; 结果为 0
31     f(-10 + MAX_COMP_2(a, b)); // -10 + (a > b ? a : b); 结果为 -10 + 8 = -2
32
33     CallWithMax(a, b); // 8
34
35     return 0;
36 }
```

## 1.20.4 附：C/C++ 编译过程（简）

### 编译过程

#### 1. (分离式) 编译：每个文件独立编译

1. 预处理：处理伪指令（# 开头）和特殊符号。
  - 宏定义：#define, #undef
  - 条件编译：#ifdef, #ifndef, #endif
  - 头文件包含：#include
  - 特殊符号：\_\_LINE\_\_, \_\_FILE\_\_
2. 编译：词法分析、语法分析，确认所有指令符合语法规则，将其翻译成等价的中间代码表示或汇编代码。
3. 汇编：把汇编代码翻译成目标机器指令，得到目标文件（obj）。

**2. 链接：**将相关的目标文件进行连接（头文件包含关系、符号引用等），使这些目标文件能够成为一个被执行的同一整体。

## 1.20.5 参考资料

1. 《Effective C++》条款 02。
2. 《C++ Primer 第 5 版中文版》Page 213–214。
3. C++ 内联函数详解

<https://www.cnblogs.com/shijingjing07/p/5523224.html>

## 1.21 运算符优先级

Table 6: 运算符的优先级

priority	operator	meaning
1	[]	下标运算
	()	圆括号
	->	指向运算符
	.	成员运算符
	++	自增（后缀）
	--	自减（后缀）
2	!	逻辑非
	~	按位反
	++	自增（前缀）
	--	自减（前缀）
	-	负号
	+	正号
	(类型)	类型转换
	*	指针运算符
	&	取地址
	sizeof	长度运算
3	*	乘法
	/	除法
	%	求余
4	+	加法
	-	减法
5	<<	左移位
	>>	右移位
6	< <= > >=	关系运算符
7	==	等于
	!=	不等于
8	&	按位与
9	^	按位异或
10		按位或
11	&&	逻辑与
12		逻辑或
13	? :	条件运算符
14	= += -= *=	赋值运算符
	/= %= &=  =	
	^= <<= >>=	
15	,	逗号运算符

**Note:** 位运算的优先级是：~ > & > ^ > | 。

逻辑运算的优先级是：！ > && > || 。

不能重载的运算符：. , ? : , sizeof , .\* , ::。

逻辑与：exp1 && exp2，如果 exp1 值为 0，则不对 exp2 求值。

逻辑或：exp1 || exp2，如果 exp1 值为 1，则不对 exp2 求值。

### 1.21.1 参考资料

#### 1. Operator Overloading

<https://isocpp.org/wiki/faq/operator-overloading>

## 1.22 const

### 1.22.1 顶层 const 与底层 const

由于指针本身是一个对象，它有可以指向另一个对象，因此，指针本身是不是常量以及指针所指对象是不是常量就是两个相互独立的问题。

**顶层 const** (top-level const) 表示 **对象本身**是常量，**底层 const** (low-level const) 与指针或引用等复合类型的 **基本类型**部分有关。对指针而言，既可以是顶层 const，也可以是底层 const。

```

1  int i = 0;
2  int *const p1 = &i; // 顶层 const, 不能改变 p1 的值
3
4  const int ci = 42; // 顶层 const, 不能改变 ci 的值
5  const int *p2 = &ci; // 底层 const, 可以改变 p2 的值, 但不能通过 p2 改变 ci 的值。
6  const int *const p3 = p2; // 顶层 const + 底层 const
7
8  const int &r = ci; // 用于声明引用的 const 都是底层 const, r 不能改变 i

```

执行拷贝操作，被拷贝对象的顶层 const 属性不受影响。而对于底层 const，要求拷入和拷出的对象具有相同的底层 const 属性，或在两个对象的数据类型能够转换。

```

1  // 承接上例的定义
2
3  i = ci; // 正确
4  p2 = p3; // 正确
5
6  int *p = p3; // 错误, p 没有底层 const, 防止通过 p 间接改变 p3 所指对象。
7  p2 = &i; // 正确, int* 能转换为 const int*
8

```

(continues on next page)

(continued from previous page)

```

9  int &r1 = ci; // 错误, r1 没有底层 const
10 const int &r2 = i; // 正确, const int& 可以绑定到普通 int

```

总结：可以使用 **非常量对象** 初始化一个 **底层 const 对象**，但是不能使用 **底层 const 对象** 初始化一个 **非常量对象**。同时，一个普通的引用必须使用同类型的对象初始化。同样的初始化规则可以应用到函数的 **参数传递**上。

### 1.22.2 const 形参和实参

使用实参初始化形参时，会 **忽略掉顶层 const**，换句话说，形参的顶层 const 被忽略了。因此，当形参有顶层 const 时，传给它常量对象或非常量对象都是可以的。

```

void fcn(const int i) { /* */ }
void fcn(int i) { /* */ } // 重复定义了 fcn

```

上例中其实重复定义了 fcn，而不是重载。调用 fcn 时，既可以传入 const int 对象，也可以传入 int 对象。反之，如果参数类型是 int，不能传入 const int 对象。

### 1.22.3 const 成员函数

默认情况下，this 指针的类型是指向 **类类型非常量版本的常量指针**，即 `ClassName *const`。尽管 this 是隐式的，但它仍然需要遵守初始化规则。意味着在默认情况下，我们不能把 this 绑定到一个常量对象上。这一情况也使得我们不能在一个常量对象上调用普通的成员函数。通过把关键字 const 放在成员函数参数列表后面，定义该成员函数为 **常量成员函数**。

```

class Sale
{
    double avg_price() const;
};

```

此时，this 成为指向常量的指针，所以常量成员函数 **不能改变调用它的对象的内容**。

---

**Note:** 常量对象、常量对象的引用和指针都只能调用常量成员函数。

---

### 1.22.4 参考资料

1. 《C++ Primer 第 5 版中文版》Page 57 – 58, Page 190 – 191, Page 231 – 232。



## 2.1 in-place 运算

### 2.1.1 += 运算

+= 是一个 in-place 运算符，看如下例子：

```
1 a = []  
2 b = a  
3 a += [1,2]
```

结果如下：

```
>>> print a  
[1,2]  
>>> print b  
[1,2]
```

如果改变成如下形式：

```
1 a = []  
2 b = a  
3 a = a + [1,2]
```

则结果如下：

```
>>> print a  
[1,2]  
>>> print b  
[]
```

**Note:** `a = a + [1,2]` 不是 in-place 运算，尽管使用了同一个变量名。

## 2.1.2 add 和 iadd

`operator` 包中有两个操作：`add` 和 `iadd`。`add` 是正常加运算，`iadd` 是原位加运算。

`_add_` does simple addition, takes two arguments, returns the sum and stores it in other variable without modifying any of the argument. Normal operator's `add()` method, implements “`a+b`” and stores the result in the mentioned variable.

`_iadd_` also takes two arguments, but it makes in-place change in 1st argument passed by storing the sum in it. As object mutation is needed in this process, immutable targets such as numbers, strings and tuples, shouldn't have `_iadd_` method. Inplace operator's `iadd()` method, implements “`a+=b`” if it exists (i.e in case of immutable targets, it doesn't exist) and changes the value of passed argument. But if not, “`a+b`” is implemented.

分两种情况讨论。

### immutable targets

对于不可变目标 (immutable targets)，如数字、字符串、元组，`_add_` 和 `_iadd_` 结果是一样的，输入实参不会发生改变。

```
1 import operator
2
3 x = 5
4 y = 6
5 a = 5
6 b = 6
7
8 z = operator.add(a, b)
9 p = operator.iadd(x, y)
```

结果如下：

```
>>> print z
11
>>> print a
5
>>> print p
11
>>> print x
5
```

### mutable targets

对于可变目标 (mutable targets)，如列表、字典，输入实参会被重现赋值和更新。

```

1 import operator
2
3 a = [1,2,4,5]
4 b = [1,2,4,5]
5
6 z = operator.add(a, [1,2,3])
7 p = operator.iadd(b, [1,2,3])

```

结果如下：

```

>>> print z
[1, 2, 4, 5, 1, 2, 3]
>>> print p
[1, 2, 4, 5, 1, 2, 3]
>>> print a
[1, 2, 4, 5]
>>> print b
[1, 2, 4, 5, 1, 2, 3]

```

---

**Note:** 不可变目标（数字、字符串、元组）作为函数参数，相当于 **值传递**，函数对实参进行拷贝。

可变目标（列表、字典）作为函数参数，相当于 **引用传递**，函数对实参的修改有效。

---

### 2.1.3 参考资料

1. pytorch issue：

<https://github.com/pytorch/pytorch/issues/5687>

2. GeeksforGeeks：

<https://www.geeksforgeeks.org/inplace-vs-standard-operators-python/>

## 2.2 \_\_all\_\_ 的使用

### 2.2.1 从 \_\_init\_\_.py 谈起

`__init__.py` 的 作用一：package 的标识

在每一个 package 文件夹中都会有一个 `__init__.py` 文件。

`__init__.py` 的 作用二：定义该 package 的 `__all__`，用以模糊导入

python 中包 (package) 和模块 (module) 有两种导入形式：精确导入和模糊导入。

### 精确导入

```
from PACK import CLASS1, CLASS2
import PACK.CLASS1
```

### 模糊导入

```
from PACK import *
```

`__all__` 是一个字符串列表，用于定义模糊导入中的 `*` 中的模块，即暴露接口，也是对于模块公开接口的一种约定。

举个例子，建立如下目录结构的文件夹及文件：

```
.
|__ test.py
|__ myPack
    |__ __init__.py
    |__ func.py
```

创建了包 `myPack`，其中 `func.py` 中定义了该包的功能，包括变量、类、函数的定义。使用 `test.py` 来测试这个包的调用。`__init__.py` 中的内容如下：

```
1 from func import x, foo
2 # 假设 x 是一个变量，foo 是一个函数
3
4 __all__ = ['x', 'foo']
```

`test.py` 中的内容如下：

```
1 from myPack import *
2
3 print x
4
5 foo()
```

## 2.2.2 参考资料

1. Python 中的 `__all__`

<https://www.jianshu.com/p/ca469f693c31>

2. Python 包中 `__init__.py` 作用

<https://www.cnblogs.com/AlwinXu/p/5598543.html>

3. Python `__init__.py` 作用详解

<https://www.cnblogs.com/Lands-ljk/p/5880483.html>

#### 4. Python 中 `__all__` 的用法

<https://www.codetd.com/article/2136881>

## 2.3 is 和 ==

### 2.3.1 is 和 ==

`is` 的作用是用来检查对象的标示符 (object identity) 是否一致，也就是比较两个对象在内存中的地址是否一样，而 `==` 是用来检查两个对象是否相等。我们在检查 `a is b` 的时候，其实相当于检查 `id(a) == id(b)`。而检查 `a == b` 的时候，实际是调用了对象 `a` 的 `__eq__()` 方法，`a == b` 相当于 `a.__eq__(b)`。一般情况下，如果 `a is b` 返回 `True` 的话，即 `a` 和 `b` 指向同一块内存地址的话，`a == b` 也返回 `True`，即 `a` 和 `b` 的值也相等。

```

1 a = "hello"
2 b = "hello"
3 print a==b # True
4 print a is b # False
5 print id(a)
6 print id(b)
7
8 a = [1,2,3]
9 b = a
10 print a==b # True
11 print a is b # True

```

### 2.3.2 None

与 `None` 比较是 `is None` 而不是 `== None`。这是因为 `None` 在 Python 里是个单例对象 (singleton)：一个变量如果是 `None`，它一定和 `None` 指向同一个内存地址。而 `== None` 背后调用的是 `__eq__`，而 `__eq__` 可以被重载，下面是一个 `is not None` 但 `== None` 的例子。

```

1 class Foo(object):
2     def __eq__(self, other):
3         return True
4
5 foo = Foo()
6 print foo==None # True
7 print foo is None # False

```

### 2.3.3 参考资料

1. 经典 7 大 Python 面试题

[https://blog.csdn.net/qq\\_41597912/article/details/81459804](https://blog.csdn.net/qq_41597912/article/details/81459804)

## 2.4 装饰器

### 2.4.1 作用

装饰器本质上是一个 Python 函数，它可以让其他函数在 **不需要做任何代码变动的前提下 增加额外功能**，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。

概括的讲，装饰器的作用就是为已经存在的函数或对象添加额外的功能。

### 2.4.2 使用装饰器计时

```
1 from functools import wraps
2 import time
3
4 def timer(func):
5     @wraps(func)
6     def function_timer(*arg, **kwargs):
7         start_time = time.time()
8         result = func(*args, **kwargs)
9         end_time = time.time()
10        return result
11    return function_timer
12
13 @timer
14 def foo():
15     print "hello"
16
17 print foo.__name__
18 print foo.__doc__
```

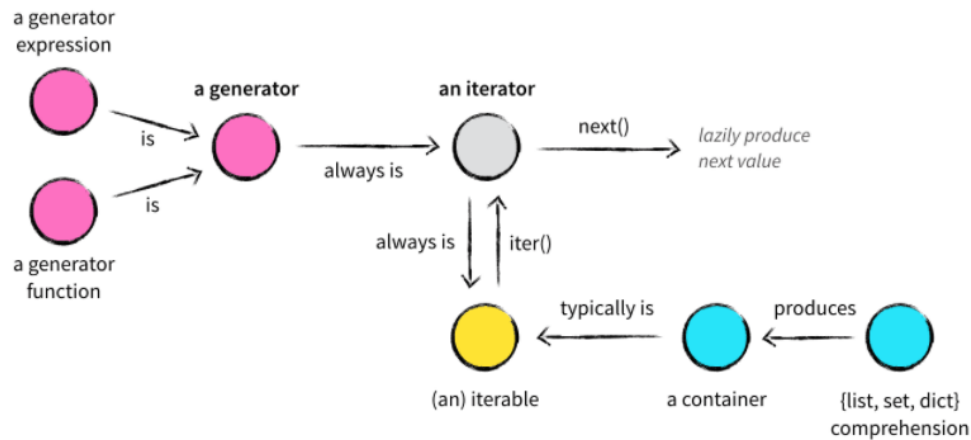
使用 `wraps` 可以保持函数 `foo()` 的属性 `__name__` 和 `__doc__`，而不变成函数 `function_timer` 的相关属性。

### 2.4.3 参考资料

#### 1. 详解 Python 的装饰器

<https://www.cnblogs.com/cicaday/p/python-decorator.html>

## 2.5 迭代器和生成器



### 2.5.1 迭代器 (iterator)

特点:

- 迭代器是访问集合元素的一种方式，不能随机访问集合中的某个值，只能从头到尾依次访问（`next()` 方法），访问到一半时不能往回退。
- 不需要事先准备好整个迭代过程中的所有元素。迭代器仅仅在迭代到某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。
- 便于循环比较大的数据集合，节省内存。
- 不能复制一个迭代器，如果要再次（或者同时）迭代同一个对象，只能去创建另一个迭代器对象。

```

1  ## enumerate 返回迭代器
2  a = enumerate(['a', 'b'])
3
4  for i in range(2): ## 迭代两次 enumerate 对象
5      for x, y in a:
6          print x, y
7  print "====="

```

结果是：

```
(0, 'a')
(1, 'b')
=====
=====
```

可以发现：第二次返回值为空。

### 可迭代对象 (iterable)

可以直接作用于 `for` 循环的对象统称为可迭代对象 (Iterable) 。只要定义了可以返回一个迭代器的 `__iter__()` 方法，或者定义了可以支持下标索引的 `__getitem__()` 方法，那么它就是一个可迭代对象。

```
1 class Iterator_test(object):
2     def __init__(self, data):
3         self.data = data
4         self.index = len(data)
5
6     def __iter__(self):
7         return self
8
9     def next(self):
10        if self.index <= 0:
11            raise StopIteration
12        self.index -= 1
13        return self.data[self.index]
14
15 iterator_winter = Iterator_test('abcde')
16 for item in iterator_winter:
17     print item
18 ## 打印 e d c b a
19
20 class Iterator_test2(object):
21     def __init__(self, data):
22         self.data = data
23     def __getitem__(self, it):
24         return self.data[it]
25 no_iter = Iterator_test2('abcde')
26 for item in no_iter:
27     print item
28 ## 打印 a b c d e
```

常见的可迭代对象：

- 集合数据类型，如 `list`、`tuple`、`dict`、`set`、`str` 等。



- generator, 包括生成器和带 yield 的 generator function。

可以被 `next()` 函数调用并不断返回下一个值的对象称为迭代器 (Iterator) 。生成器都是 Iterator 对象, 但 list、dict、str 虽然是 Iterable, 却不是 Iterator。

所有的 Iterable 均可以通过内置函数 `iter()` 来转变为 Iterator 。

判断一个对象是否是可迭代对象:

```
1 from collections import Iterable
2 a = [1,2,3]
3 isinstance(a, Iterable) # True
4
5 a = iter(a)
6 next(a) # 或 a.next(), 返回 1
7 next(a) # 返回 2
8 next(a) # 返回 3
9 next(a) # 抛出 StopIteration 异常
```

一个可迭代对象是不能独立进行迭代的, Python 中, 迭代是通过 `for ... in` 来完成的。for 循环会不断调用迭代器对象的 `__next__()` 方法 (python3 `__next__()` ; python2 `next()` ), 每次循环, 都返回迭代器对象的下一个值, 直到遇到 `StopIteration` 异常。

任何实现了 `__iter__()` 和 `__next__()` (python2 中实现 `next()` ) 方法的对象都是迭代器, `__iter__()` 返回迭代器自身, `__next__()` 返回容器中的下一个值。

## 2.5.2 生成器 (generator)

生成器其实是一种特殊的迭代器。它不需要再像上面的类一样写 `__iter__()` 和 `__next__()` 方法了, 只需要一个 `yield` 关键字。yield 就是 return 返回的一个值, 并且记住这个返回的位置。下一次迭代就从这个位置开始。生成器一定是迭代器 (反之不成立), 因此任何生成器也是以一种懒加载的模式生成值。

```
1 def generator_winter():
2     i = 1
3     while i <= 3:
4         yield i
5         i += 1
6
7 generator_iter = generator_winter() ## 函数体中的代码并不会执行, 只有显示或隐式地调用 next 的时候才会
   真正执行里面的代码。
8 print generator_iter.next() ## 1
9 print generator_iter.next() ## 2
10 print generator_iter.next() ## 3
11 print generator_iter.next() ## 抛出 StopIteration 异常
```

生成器表达式 (类似于列表推导式, 只是把 `[]` 换成 `()`)。

```
1 gen = (x for x in range(10)) ## <generator object <genexpr> at 0x0000000012BC4990>
2 for item in gen:
3     print item
4
5 ## fibonacci 数列
6 def fibonacci(n):
7     a, b = 0, 1
8     while b <= n:
9         yield b
10        a, b = b, a+b
11 f = fibonacci(10)
12 for item in f:
13     print item
```

### 2.5.3 参考资料

1. Python 迭代器，生成器-精华中的精华

<https://www.cnblogs.com/deeper/p/7565571.html>

2. python 生成器和迭代器有这篇就够了

<https://www.cnblogs.com/wj-1314/p/8490822.html>

## 2.6 lambda 用法

使用：

```
lambda [arg1[, arg2, ... argN]]: expression
```

特性：

- 匿名函数。函数没有名字。
- 输入是 arg list，输出是根据 expression 计算得到的值。
- 功能简单。

### 2.6.1 使用场景

1. 将 lambda 函数赋值给一个变量，通过这个变量间接调用该函数。

```
1 plus = lambda x, y: x+y
2 print plus(1,2)
```

2. 将 lambda 函数赋值给其他函数，从而屏蔽其他函数本来的功能。

3. 将 lambda 函数作为其他函数的返回值（内部函数）。

4. 将 lambda 函数作为参数传递给其他函数。

- **map**：将序列中的元素通过处理函数处理后返回一个新的列表。
- **filter**：将序列中的元素通过函数过滤后返回一个新的列表。
- **reduce**：将序列中的元素通过一个二元函数处理返回一个结果。
- **sorted**：结合 lambda 对列表进行排序。`sorted(iterable, cmp=None, key=None, reverse=False)`

```

1 a = [1, 2, 6, 5, 2, -8, -5, -1, -10]
2
3 ## 每个元素加 1
4 b = map(lambda x: x+1, a) # [2, 3, 7, 6, 3, -7, -4, 0, -9]
5
6 ## 提取序列中大于 0 的数
7 c = filter(lambda x: x>0, a) # [1, 2, 6, 5, 2]
8
9 ## 返回所有元素相乘的结果
10 d = reduce(lambda x, y: x*y, a) # 48000
11
12 ## 负数排在正数前面，同时绝对值大的排在后面
13 ## 两个 key，先按第一个 key 排序，若第一个 key 相同则按下一个 key 排序
14 e = sorted(a, key=lambda x: (x>0, abs(x))) # [-1, -5, -8, -10, 1, 2, 2, 5, 6]
```

## 2.6.2 参考资料

1. 关于 Python 中的 lambda，这可能是你见过的最完整的讲解

<https://blog.csdn.net/zjuxsl/article/details/79437563>

2. 在 Python 中使用 lambda 高效操作列表的教程

<https://www.cnblogs.com/mxp-neu/articles/5316557.html>

## 2.7 \*args 和 \*\*kwargs

### 2.7.1 \*args

`*args` 用来将 不定数量的参数打包成 `tuple` 给函数体使用。

例一：

```
1 def foo(x, *args):
2     print "x:", x
3     for k in range(len(args)):
4         print "args[{}]:".format(k), args[k]
```

```
1 >>> foo(1, 100, '200k', 300)
2 x: 1
3 args[0]: 100
4 args[1]: 200k
5 args[2]: 300
6
7 >>> args = [1,2,'abc']
8 >>> foo('A', *args)
9 x: A
10 args[0]: 1
11 args[1]: 2
12 args[2]: abc
13
14 >>> foo('A', args) ## 注: 此时把 args 当做一个参数, 参数类型为列表
15 x: A
16 args[0]: [1, 2, 'abc']
```

例二:

```
1 def foo(x, var1, var2, var3):
2     print "x:", x
3     print "var1:", var1
4     print "var2:", var2
5     print "var3:", var3
```

```
1 >>> args = [1,2,'A'] # list
2 >>> foo(1, args)
3 TypeError: foo() takes exactly 4 arguments (2 given)
4 >>> foo(1, *args)
5 x: 1
6 var1: 1
7 var2: 2
8 var3: A
9
10 >>> args = (1,2,'A') # tuple
11 >>> foo(1, args)
12 TypeError: foo() takes exactly 4 arguments (2 given)
13 >>> foo(1, *args)
14 x: 1
```

(continues on next page)

(continued from previous page)

```

15 var1: 1
16 var2: 2
17 var3: A

```

### 2.7.2 \*\*kwargs

**\*\*kwargs** 打包 不定数量的键值对参数成 dict 给函数体使用。

例一：

```

1 def foo(**kwargs):
2     for key, val in kwargs.items():
3         print "{} : {}".format(key, val)

```

```

1 >>> foo(var1=1, var2='a', var3=[1,2,3])
2 var1 : 1
3 var3 : [1, 2, 3]
4 var2 : a

```

例二：

```

1 def foo(x, var1=2, var2='a'):
2     print "x:", x
3     print "var1:", var1
4     print "var2:", var2

```

```

1 >>> dict_input = {"var1": 10, "var2": "A"}
2 >>> foo(1, dict_input)
3 x: 1
4 var1: {'var1': 10, 'var2': 'A'}
5 var2: a
6
7 >>> foo(1, **dict_input)
8 x: 1
9 var1: 10
10 var2: A

```

### 2.7.3 arg, \*args, \*\*kwargs

位置参数、\*args、\*\*kwargs 三者的顺序必须是 (arg, \*args, \*\*kwargs)。

```
1 def foo(arg, *args, **kwargs):
2     print "arg:", arg
3     print "args:", args
4     print "kwargs:", kwargs
```

```
1 >>> foo(1, 2, 3, 4, x=1, y='b')
2 arg: 1
3 args: (2, 3, 4)
4 kwargs: {'y': 'b', 'x': 1}
5
6 >>> foo(1, x=1, y='b', 2, 3, 4)
7 SyntaxError: non-keyword arg after keyword arg
```

位置参数、默认参数、\*\*kwargs 三者的顺序必须是（位置参数，默认参数，\*\*kwargs）。

```
1 def foo(x, y=1, **kwargs): ## 不能出现 (x=1,y,**kwargs)
2     print "x:", x
3     print "y:", y
4     print "kwargs:", kwargs
```

```
1 >>> foo(4, var1=1, var2='b')
2 x: 4
3 y: 1
4 kwargs: {'var1': 1, 'var2': 'b'}
```

## 2.7.4 参考资料

1. 大话 Python 中 \*args 和 \*\*kwargs 的使用  
<https://www.cnblogs.com/shitaotao/p/7609990.html>
2. python 函数——形参中的：\*args 和 \*\*kwargs  
<https://www.cnblogs.com/xuyuanyuan123/p/6674645.html>

## 2.8 基本数据类型

### 2.8.1 类型与方法

- str
  - 索引、切片：[ind], [first:last] 获取区间 [first, last) 内的元素。
  - 长度：len()

- 查找：若字符/序列不在字符串内，`index()` 报错 `ValueError`，`find()` 返回-1。
- 判断字符串内容：字母，`isalpha()`；数字，`isdigit()`；数字或字母，`isalnum()`。
- 大小写转换：`capitalize()`、`lower()`、`upper()`。
- 判断以什么开头结尾：`startswith()`、`endswith()`。
- 连接：`join()`，将字符串、元组、列表中的元素以指定的字符（分隔符）连接生成一个新的字符串。
- 分割：`split()`、`partition()`。**如果想把字符串分割成独立的字符，用 `list(string)`。**
- 替代：`replace()`
- 清除空白：`strip()`、`lstrip()`、`rstrip()`

```

1 >>> s = "abcde"
2 >>> "-".join(s)
3 a-b-c-d-e
4 >>> s = ['abc', 'def', 'ghi']
5 >>> "-".join(s)
6 abc_def_ghi
7 >>> s
8 ['abc', 'def', 'ghi']
9
10 >>> s = "a-b-c-d-e"
11 >>> s.partition('-') ## 只能分割为 3 部分
12 ('a', '-', 'b-c-d-e')
13 >>> s.split('-')
14 ['a', 'b', 'c', 'd', 'e']
15 >>> s
16 "a-b-c-d-e"

```

**Warning:** `str` 是不可变对象，其所有方法都 **不改变对象本身**，而是返回所创建的新对象。

#### • list

- 索引、切片：`[ind]`，`[first:last]` 获取区间 `[first, last)` 内的元素。
- 统计元素出现的次数：`count()`
- 追加：`append()`
- 拓展：`extend()`
- 插入：`insert()`
- 弹出元素：`pop()`，默认弹出列表末尾的元素
- 移除/删除元素：`remove()`，`del`（`del` 可删除切片）

- 排序: `sort()`

```
1 >>> a = [1,2,3]
2 >>> a.append(4)
3 >>> a
4 [1, 2, 3, 4]
5 >>> a.extend([10,20,30])
6 >>> a
7 [1, 2, 3, 4, 10, 20, 30]
8
9 >>> a.insert(1, 5) ## 在第一个元素之后插入
10 >>> a
11 [1, 5, 2, 3, 4, 10, 20, 30]
12
13 >>> a.remove(2)
14 >>> a
15 [1, 5, 3, 4, 10, 20, 30]
16 >>> del a[3]
17 >>> a
18 [1, 5, 3, 10, 20, 30]
19
20 >>> a.sort(reverse=True)
21 >>> a
22 [30, 20, 10, 5, 3, 1] ## 直接修改 a, 无返回值。使用 sorted 返回排序后的副本。
23
24 >>> a2 = a.pop(2)
25 >>> a
26 [1, 5, 10, 20, 30] ## a
```

- **dict**

- 获取: `keys()`, `values()`, `items()`。
- 清除: `clear()`
- 访问: `get(key)`, 不存在时返回 `None`。
- 更新: `update(d)`, 把另一个字典 `d` 中的项添加到当前字典。
- 浅复制: `copy()`

```
1 >>> info ={
2 ...     "name": "Tom",
3 ...     "age": 25,
4 ...     "sex": "man",
5 ... }
6 >>> info.keys()
```

(continues on next page)



(continued from previous page)

```

7  ['age', 'name', 'sex']
8  >>> info.values()
9  [25, 'Tom', 'man']
10 >>> info.items()
11 [('age', 25), ('name', 'Tom'), ('sex', 'man')]
12
13 >>> info.get(age)
14 25
15 >>> new = {"weight": 60}
16 >>> info.update(new)
17 >>> info
18 {'age': 25, 'name': 'Tom', 'weight': 60, 'sex': 'man'}
19 >>> info.clear()
20 >>> info
21 {}

```

- **collections.defaultdict** : defaultdict 类使用一种给定数据类型来初始化。当所访问的 key 不存在的时候，会实例化一个 value 作为默认值。因此，判断某个 key 是否存在，可使用 get(key)。

```

1  >>> from collections import defaultdict
2  >>> dd = defaultdict(list) ## 使用 list 作为 value type
3  defaultdict(<type 'list'>, {})
4  >>> dd['a']
5  []
6  >>> dd['b'].append("hello")
7  defaultdict(<type 'list'>, {'a': [], 'b': ['hello']})

```

**Warning:** 如果一个 defaultdict 必须包含给定的 key，则首先要 **显式**地对所有的 key 进行访问和初始化。毕竟 defaultdict 只会为访问过的 key 关联一个默认值。

- set

- 特征：无重复，无须，每个元素为不可变类型
- 增加元素：单个元素，add()；多个元素，update()
- 删除：删除元素不存在，remove() 报错，discard() 无反应。
- 集合操作：&, |, -, ^ (交集补集，去除交集后剩下元素的并集)，issubset()、issuperset()。

```

1  >>> s1 = {'a', 'b', 'c'} ## 或者 s1 = set(['a', 'b', 'c'])
2  >>> s1.update({'e', 'd'})
3  >>> s1
4  set(['a', 'c', 'b', 'e', 'd'])

```

**Note:** 对于切片 (slice) 操作, 下标越界 **不会**报错, 返回空。

对于索引 (index) 操作, 下标越界 **会**报错。

---

```
s[i:j]
```

The **slice** of **s** from **i** to **j** is defined as the sequence of items with index **k** such that  $i \leq k < j$ .

If **i** or **j** is greater than **len(s)**, use **len(s)**.

If **i** is omitted or **None**, use 0.

If **j** is omitted or **None**, use **len(s)**.

If **i** is greater than or equal to **j**, the **slice** is empty.

## 2.8.2 深复制和浅复制

- **直接赋值:** 并没有拷贝对象, 而是拷贝了对象的引用, 因此原始对象或被赋值对象的改变, 都会导致另一个对象被修改。

```
1 >>> alist = [1,2,3]
2 >>> b = alist ## 引用
3 >>> c = alist[:] ## 复制
4 >>> alist.append(5)
5 >>> alist
6 [1, 2, 3, 5]
7 >>> b
8 [1, 2, 3, 5]
9 >>> c
10 [1, 2, 3]
11 >>> b[0] = -1
12 >>> a
13 [-1, 2, 3, 5]
14 >>> b
15 [-1, 2, 3, 5]
16 >>> c
17 [1, 2, 3]
```

- **浅复制:** 只会复制父对象, 而不会复制对象的内部的子对象。

```
1 >>> from copy import copy
2 >>> alist = [1,2,3,['a','b']] ## ['a','b'] 是列表, 是一个子对象
3 >>> a_copy = copy(alist) ## dict 类有 copy() 方法, e.g., d.copy()
4 >>> alist.append(5) ## 非子对象的修改
```

(continues on next page)

(continued from previous page)

```
5 >>> alist
6 [1, 2, 3, ['a', 'b'], 5]
7 >>> a_copy
8 [1, 2, 3, ['a', 'b']]
9 >>> a_copy[0] = -1
10 >>> alist
11 [1, 2, 3, ['a', 'b'], 5]
12 >>> a_copy
13 [-1, 2, 3, ['a', 'b']]
14
15 >>> alist[3].append('c') ## 子对象的修改
16 >>> alist
17 [1, 2, 3, ['a', 'b', 'c'], 5]
18 >>> a_copy
19 [-1, 2, 3, ['a', 'b', 'c']]
20 >>> a_copy[3].append('d')
21 >>> alist
22 [1, 2, 3, ['a', 'b', 'c', 'd'], 5]
23 >>> a_copy
24 [-1, 2, 3, ['a', 'b', 'c', 'd']]
```

- **深复制**：复制对象及其子对象，原始对象的改变不会造成深复制里任何子元素的改变。

```
1 >>> from copy import deepcopy
2 >>> alist = [1,2,3,['a','b']] ## ['a','b'] 是列表，是一个子对象
3 >>> a_copy = deepcopy(alist)
4 >>> alist[3].append('c') ## 子对象的修改
5 >>> alist
6 [1, 2, 3, ['a', 'b', 'c']]
7 >>> a_copy
8 [1, 2, 3, ['a', 'b']]
9 >>> a_copy[3].append('d')
10 >>> alist
11 [1, 2, 3, ['a', 'b', 'c']]
12 >>> a_copy
13 [1, 2, 3, ['a', 'b', 'd']]
```

**Note:** 对于可变对象 `dict` 和 `list`，需要暂存临时对象或者作为函数参数传递时，如果不希望对象被更改，都需要使用深复制。

### 2.8.3 再谈可变对象与不可变对象

第一章曾提到过可变对象与不可变对象。

`dict` 和 `set` 的底层实现都是 **哈希表**。哈希要求 `key` 唯一，因此 `dict` 和 `set` 的 `key` 都要求是 **不可变对象**。

```
1 >>> x = 'abcd'
2 >>> id(x)
3 313010056L
4 >>> y = 'abcd'
5 >>> id(y)
6 313010056L
7 ## x 和 y 都是 str 对象，值相同，占用同一块内存。
8
9 >>> a = [5, 3, 4, 3]
10 >>> id(a)
11 314009096L
12 >>> b = [5, 3, 4, 3] ## b = a[:]
13 >>> id(b)
14 314011080L
15 ## a 和 b 的 id 不同，尽管值相同。因此 list 不能作为 key。
```

### 2.8.4 参考资料

1. Python 基本数据类型

<https://www.cnblogs.com/littlefivebolg/p/8982889.html>

2. 切片 python 字符串时为何不会引起下标越界？

<https://segmentfault.com/q/1010000011412371>

3. python 中 defaultdict 方法的使用

<https://www.cnblogs.com/dancesir/p/8142775.html>

4. python 的复制，深拷贝和浅拷贝的区别

<https://www.cnblogs.com/xueli/p/4952063.html>

5. Python 学习日记之字典深复制与浅复制

<https://www.cnblogs.com/mokero/p/6662202.html>

## 2.9 random

### 2.9.1 random

python 自带的 random 库。

```
import random
```

- **random.random()**
  - 生成 0~1 的随机浮点数。
- **random.uniform(a,b)**
  - 生成指定范围 [a, b] 内的随机浮点数。
- **random.randint(a,b)**
  - 生成指定范围 [a, b] 内的随机整数。
- **random.randrange(start,stop,step)**
  - 指定范围内，按 step 递增的集合中的随机数。
- **random.choice(lst)**
  - 给定的集合中选择一个元素。
- **random.shuffle(lst)**
  - 对一个序列或者元组随机打乱。

### 2.9.2 numpy.random

```
import numpy as np
```

- **numpy.random.random([  $d_0, d_1, \dots, d_n$  ])**
  - 生成 0~1 的随机浮点数，维度为  $d_0 \times d_1 \times \dots \times d_n$ （缺省为 1）。
- **numpy.random.rand(  $d_0, d_1, \dots, d_n$  )**
  - 生成 0~1 的随机浮点数，维度为  $d_0 \times d_1 \times \dots \times d_n$ （缺省为 1）。
- **numpy.random.randn(  $d_0, d_1, \dots, d_n$  )**
  - 标准正态分布。
- **numpy.random.randint(low, high=None, size=None, dtype=' i' )**
  - 返回随机的整数，位于半开区间 [low, high)。如果 high=None，区间为 [0, low)。
- **numpy.random.choice(arr, size=None, replace=True, p=None)**

- 从一个给定的一维数组，按概率  $p$  抽样一定数量的元素，`replace=True` 表示允许重复元素。
- `numpy.random.shuffle(arr)`
  - 随机打乱 `arr`。
- `numpy.random.permutation(arr)`
  - 返回一个随机排列。
- `numpy.random.seed(n)`
  - 改变随机数生成器的种子。设置相同的 `seed`，每次生成的随机数相同；如果不设置 `seed`，则每次会生成不同的随机数。

```

1  ## 注：生成的数组都是 numpy array 类型
2
3  >>> 2.5 * np.random.randn(2, 4) + 3
4  [[-0.52410303  1.68461615 -0.04895917  2.81907944]
5   [ 6.89754303  2.95949232  1.85296809  1.56361545]]
6
7  >>> np.random.randint(5, size=(2, 4))
8  [[0 4 2 3]
9   [0 0 4 4]]
10
11 ## 从 np.arange(4) 选取 3 个元素
12 >>> np.random.choice(4, 3)
13 [3 1 2]
14 >>> np.random.choice([1,3,9,0], 3)
15 [9 3 0]
16 >>> np.random.choice(4, 3, p=[0.1, 0.2, 0, 0.7])
17 [1 3 3]
18
19 >>> arr = np.arange(10)
20 >>> np.random.shuffle(arr)
21 >>> print arr
22 [6 9 0 8 1 7 4 5 3 2]
23
24 >>> np.random.permutation(10)
25 [5 8 9 7 3 1 0 2 6 4]
26 >>> np.random.permutation([1, 4, 9, 12, 15])
27 [ 9  1  4 12 15]
28 >>> arr = np.arange(9).reshape((3, 3))
29 >>> np.random.permutation(arr)
30 [[3 4 5]
31  [6 7 8]
32  [0 1 2]]
33

```

(continues on next page)

(continued from previous page)

```

34 >>> np.random.seed(1)
35 >>> np.random.random()
36 0.417022004702574
37 >>> np.random.seed(1)
38 >>> np.random.random()
39 0.417022004702574
40 >>> np.random.random()
41 0.7203244934421581

```

### 2.9.3 参考资料

1. random 与 numpy.random

<https://www.jianshu.com/p/36a4bbb5536e>

2. numpy 的 random 模块详细解析

<https://www.cnblogs.com/zuoshoushizi/p/8727773.html>

## 2.10 归一化

### 2.10.1 numpy.linalg.norm

```
numpy.linalg.norm(x, ord=None, axis=None, keepdims=False)
```

### 2.10.2 sklearn.preprocessing.normalize

```
sklearn.preprocessing.normalize(X, norm='l2', axis=1, copy=True, return_norm=False)
```

### 2.10.3 torch.nn.functional.normalize

```
torch.nn.functional.normalize(input, p=2, dim=1, eps=1e-12)
```

```

1 >>> import numpy as np
2 >>> import numpy.linalg as la
3 >>> arr = np.array([[2,1,2],[2,1,1]], dtype=np.float32)
4 >>> print arr
5 [[ 2.  1.  2.]
6  [ 2.  1.  1.]]

```

(continues on next page)

(continued from previous page)

```
7 >>> norm = la.norm(arr, axis=0, keepdims=True)
8 >>> print norm
9 [[ 2.82842708  1.41421354  2.23606801]]
10 >>> print arr / np.tile(norm,(2,1))
11 [[ 0.70710677  0.70710677  0.89442718]
12  [ 0.70710677  0.70710677  0.44721359]]
13
14 >>> from sklearn import preprocessing
15 >>> print preprocessing.normalize(arr, axis=0, norm='l2')
16 [[ 0.70710677  0.70710677  0.89442718]
17  [ 0.70710677  0.70710677  0.44721359]]
18
19 >>> import torch.nn.functional as F
20 >>> print F.normalize(torch.from_numpy(arr), p=2, dim=0)
21 0.7071  0.7071  0.8944
22 0.7071  0.7071  0.4472
23 [torch.FloatTensor of size 2x3]
```

## 2.10.4 参考资料

1. `numpy.linalg.norm`

<https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.norm.html>

2. `sklearn.preprocessing.normalize`

<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html>

3. `torch.nn.functional.normalize`

<https://pytorch.org/docs/0.3.0/nn.html?highlight=normalize#torch.nn.functional.normalize>

## 2.11 常用数据结构

### 2.11.1 栈

`list` 的 `append()` 和 `pop()` 方法使得 `list` 类型可以作为简单的栈使用。



## 2.11.2 队列

### Queue

```
import Queue
```

- FIFO

```
Queue.Queue(maxsize=0)
```

先进先出。

`maxsize` 指明了队列中能存放的数据个数的上限。

一旦达到上限，插入会导致阻塞，直到队列中的数据被消费掉。

如果 `maxsize` 小于或者等于 0，队列大小没有限制。

- LIFO

```
Queue.LifoQueue(maxsize=0)
```

后进先出，类似于栈。

- Priority

```
Queue.PriorityQueue(maxsize=0)
```

优先队列。

一般使用 `tuple`（优先级 + 数据）作为队列元素，优先级为 `tuple` 的第一项。

默认 `sorted(list(entries))[0]`，即 `tuple` 第一项越小，优先级越高，越先出队列。

### 插入元素

```
## que is an initialization of Queue
que.put(item)
```

### 弹出并返回元素

```
item = que.get()
```

### 判断是否为空

```
que.empty()
```

### 队列大小

```
que.qsize()
```

例子：

```
1 from Queue import PriorityQueue
2 que = PriorityQueue()
3 que.put((1, 'apple'))
4 que.put((10, 'app'))
5 que.put((5, 'banana'))
6
7 while not que.empty():
8     print que.get(), que.qsize()
9
10 ## print result
11 ## (1, 'apple') 2
12 ## (5, 'banana') 1
13 ## (10, 'app') 0
```

## deque

double-ended queue, 双端队列。

```
from collections import deque
```

### 方法:

- `append()`, `appendleft()`
- `pop()`, `popleft()`
- `extend()`, `extendleft()`
- `reverse()`
- `rotate()`
- `count()`
- `clear()`

### 例子:

```
1 >>> dq = deque(range(5))
2 >>> dq
3 deque([0, 1, 2, 3, 4])
4 >>> dq.rotate() ## right-shift
5 >>> dq
6 deque([4, 0, 1, 2, 3])
7 >>> dq.rotate(3)
8 >>> dq
9 deque([1, 2, 3, 4, 0])
```

(continues on next page)

(continued from previous page)

```

10 >>> dq.rotate(-3) ## left-shift
11 deque([4, 0, 1, 2, 3])
12 >>> dq.reverse()
13 >>> dq
14 deque([3, 2, 1, 0, 4])

```

### 2.11.3 堆

```
import heapq
```

heapq 创建的是 **小顶堆**，堆顶元素是堆的最小元素。

#### 创建堆

- **heappush()**

基于空列表 `[]`，使用 `heappush()` 把元素逐个插入堆中。`heappop(h)` 弹出并返回堆顶元素。`h[0]` 是最小值。

```

1 >>> def heapsort(iterable):
2 ...     h = []
3 ...     for value in iterable:
4 ...         heapq.heappush(h, value)
5 ...     return [heapq.heappop(h) for _ in range(len(h))] ## 不能直接返回 h
6 ...
7 >>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
8 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

- **heapify(list\_x)**

把列表转换为堆，in-place，线性时间。

```

1 >>> h = [2, 3, 5, 1, 54, 23, 132]
2 >>> heapq.heapify(h)
3 >>> print h
4 [1, 2, 5, 3, 54, 23, 132] ## h 是堆，但是 h 不一定是有序的，只能保证 h[0] 是最小值。
5 >>> print [heapq.heappop(h) for _ in range(len(h))]
6 [1, 2, 3, 5, 23, 54, 132]

```

- **merge**

合并多个排序后的序列，返回排序后的序列的迭代器。

```
1 >>> h1 = [32, 3, 5, 34, 54, 23, 132]
2 >>> h2 = [23, 2, 12, 656, 324, 23, 54]
3 >>> h1 = sorted(h1)
4 >>> h2 = sorted(h2)
5 >>> h = heapq.merge(h1, h2)
6 >>> print type(h), list(h)
7 <type 'generator'> [2, 3, 5, 12, 23, 23, 23, 32, 34, 54, 54, 132, 324, 656]
```

- **heapreplace**

删除堆中最小元素，并插入新的元素。

```
1 >>> h = [32, 3, 5, 34, 54, 23, 132]
2 >>> heapq.heapify(h)
3 >>> heapq.heapreplace(h, 9)
4 >>> print [heapq.heappop(h) for _ in range(len(h))]
5 [5, 9, 23, 32, 34, 54, 132]
```

## 获取最值

```
heapq.nlargest(n, iterable[, key])
heapq.nsmallest(n, iterable[, key])
```

返回一个长度为 n 的列表，包含数据中的前 n 个最大/最小的元素。使用 key 定义排序关键字。

```
1 >>> nums = [1, 3, 4, 5, 2]
2 >>> print heapq.nlargest(3, nums)
3 [5, 4, 3]
4 >>> print heapq.nsmallest(3, nums)
5 [1, 2, 3]
6
7 >>> info = [
8     {'name': 'IBM', 'price': 91.1},
9     {'name': 'AAPL', 'price': 543.22},
10    {'name': 'FB', 'price': 21.09},
11    {'name': 'HPQ', 'price': 31.75},
12    {'name': 'YHOO', 'price': 16.35},
13    {'name': 'ACME', 'price': 115.65}
14 ]
15 >>> cheap = heapq.nsmallest(2, info, key=lambda x:x['price'])
16 >>> expensive = heapq.nlargest(2, info, key=lambda x:x['price'])
17 >>> print cheap
18 [{'price': 16.35, 'name': 'YHOO'}, {'price': 21.09, 'name': 'FB'}]
```

(continues on next page)

(continued from previous page)

```

19 >>> print expensive
20 [{'price': 543.22, 'name': 'AAPL'}, {'price': 115.65, 'name': 'ACME'}]

```

## 大顶堆

heapq 默认创建小顶堆，为了创建大顶堆，有以下 trick:

```

heapq.heappush(-x) ## 插入 x
x = - heapq.heappop(h) ## 弹出堆顶元素

```

## 数列前 K 大的数

Hint: 建立大小为 K 的小顶堆，对后续所有数进行遍历：如果大于堆顶元素，则有可能是前 K 大的数，堆顶元素弹出，插入该数。时间复杂度  $\mathcal{O}(N\log K)$ 。

```

1 import heapq as hq
2
3 class TopKHeap(object):
4     def __init__(self, k=3):
5         self.k = k
6         self.data = []
7
8     def push(self, x):
9         if len(self.data) < self.k:
10             hq.heappush(self.data, x)
11         else:
12             min_number = self.data[0]
13             if x > min_number:
14                 hq.heapreplace(self.data, x)
15
16     def topk(self):
17         return list(reversed([hq.heappop(self.data) for _ in range(len(self.data))]))
18
19 def main():
20     nums = range(1, 10)
21     tkh = TopKHeap(3)
22     for n in nums:
23         tkh.push(n)
24     print tkh.topk() ## [9, 8, 7]
25
26 if __name__ == '__main__':
27     main()

```

## 2.11.4 计数器

```
from collections import Counter
```

Counter 用于统计频率。属性与字典类似，有 keys(), values(), items() 等。

---

**Note:** Counter 统计之后并不一定是按照频率从高到低排列的。

---

```
1 >>> cnt = Counter() ## 空计数器
2 >>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
3 ...     cnt[word] += 1
4 >>> cnt
5 Counter({'blue': 3, 'red': 2, 'green': 1})
6 >>> cnt = ['red', 'blue', 'red', 'green', 'blue', 'blue']
7 >>> cnt
8 Counter({'blue': 3, 'red': 2, 'green': 1})
9
10 >>> cnt.most_common(2) ## 返回出现频率最高的两个元素
11 [('blue', 3), ('red', 2)]
12
13 >>> c = Counter('gallahad')
14 >>> c
15 Counter({'a': 3, 'l': 2, 'h': 1, 'g': 1, 'd': 1})
16
17 >>> c = Counter({'red': 4, 'blue': 12})
18 >>> c
19 Counter({'blue': 12, 'red': 4})
20 >>> c['green'] ## 访问不存在关键字，可使用 c.get('green')
21 0
```

## 2.11.5 参考资料

1. python 中的 Queue(队列) 详解

<https://www.cnblogs.com/wdliu/p/6905396.html>

2. Python collections 使用

<https://www.jianshu.com/p/f2a429aa5963>

3. Python 标准库模块之 heapq

<https://www.jianshu.com/p/801318c77ab5>

<https://docs.python.org/2/library/heapq.html>

4. python 使用 heapq 实现小顶堆 (TopK 大) /大顶堆 (BtmK 小)

<https://blog.csdn.net/tanghaiyu777/article/details/55271004>

5. Counter

<https://docs.python.org/2/library/collections.html?highlight=counter>





## 3.1 基本命令

### 3.1.1 文件和目录

```
cd ..  
pwd  
ls -a -F -R -l  
  
cp [-i] src dst  
cp -R  
  
mv src des  
rm -i -r -f folder  
  
touch new ## 创建新文件或修改文件时间属性  
  
mkdir new  
rmdir new  
  
file my_file ## 查看文件类型  
  
cat -n log.txt  
tail log.txt  
head -5 log.txt  
  
wc file -c -w -l
```

### 3.1.2 磁盘空间

```
df -h
du [-s] -h
```

### 3.1.3 处理数据文件

```
sort [-n] log.txt ## -n : 行号

grep [-n] [-c] t file ## find *t* in file

gzip my*
gunzip myfile.gz

tar -cvf test.tar test/
tar -xvf test.tar
tar -xzvf test.tgz
```

### 3.1.4 参考资料

1. 《Linux 命令行与 shell 脚本编程大全》
2. 每天一个 linux 命令目录  
<http://www.cnblogs.com/peida/archive/2012/12/05/2803591.html>

## 3.2 更多指令

### 3.2.1 Debian 系 PMS

PMS : Package Management System, 包管理系统。主要介绍:

- dpkg
- apt-get
- apt-cache
- aptitude

#### dpkg

- dpkg -L package\_name: 列出软件包所安装的所有文件

- `dpkg -S absolute_file_name`: 查找特定的某个文件属于哪个软件包（使用绝对路径）

### apt-get

- `apt-get install package_name`: 安装一个新软件包
- `apt-get remove package_name`: 卸载一个已安装的软件包（保留配置文件）
- `apt-get -purge remove package_name`: 卸载一个已安装的软件包（删除配置文件）
- `apt-get clean`: 删除安装的软件的备份，不过不影响软件的使用。
- `apt-get update`: 更新 **软件包列表**
- `apt-get upgrade`: 升级所有已安装的 **软件包**（upgrade 之前先 update，确保升级的是最新版本）

### apt-cache

- `apt-cache showpkg package_name`: 显示软件包信息
- `apt-cache policy package_name`: 显示软件包是否已经安装、版本号等

### aptitude

- `aptitude install package_name`: 安装软件包
- `aptitude remove package_name`: 删除软件包
- `aptitude purge package_name`: 删除软件包及其配置文件
- `aptitude search package_name`: 搜索软件包
- `aptitude show package_name`: 显示软件包的详细信息
- `aptitude clean`: 删除下载的软件包文件
- `aptitude autoclean`: 仅删除过期的软件包文件
- `aptitude update`: 更新可用的软件包列表
- `aptitude upgrade`: 升级可用的软件包

## 3.2.2 tee

读取标准输入的数据，并将其内容输出成文件。

```
tee [-ai] [--help] [--version] [file ...]
```

### 参数

- `-a` 或 `-append`: 追加到既有文件

- -i 或 -ignore-interrupts: 忽略中断信号
- -help: 帮助
- -version: 版本信息

### Example

```
1  ## 将用户输入的数据同时保存到 out1 out2
2  $ tee out1 out2
3  a b c d e f g # 输入
4  a b c d e f g # 反馈
5  ^C           # 结束输入
6  $ cat out1
7  a b c d e f g
8  $ cat out2
9  a b c d e f g
10
11 ## 管道: 将屏幕的输出保存到文件
12 $ echo "hello world" | tee out
13 hello world
14 $ cat out
15 hello world
16
17 ## 把 python 程序打印到屏幕的内容保存到文件
18 $ python test.py | tee out
```

## 3.2.3 查找

### which

```
which [可执行文件...]
```

在 PATH 变量指定的路径中，搜索某个系统命令的位置，并且返回第一个搜索结果。

### whereis

```
whereis [-bmsu] [-BMS -f 目录...] [文件...]
```

只能用于程序名的搜索，而且只搜索二进制文件 (-b)、帮助说明文件 (-m) 和源代码文件 (-s)。如果省略参数，则返回所有信息。参数：

- -b: 定位可执行文件。

- -m: 定位帮助文件。
- -s: 定位源代码文件。
- -u: 搜索默认路径下除可执行文件、源代码文件、帮助文件以外的其它文件。
- -B: 指定搜索可执行文件的路径。
- -M: 指定搜索帮助文件的路径。
- -S: 指定搜索源代码文件的路径。

## locate

```
locate [-d] [--help] [--version] [范本样式...]
```

配合数据库查找文件位置。参数:

- -d: 配置 locate 指令使用的数据库。locate 指令预设的数据库位于 /var/lib/slocate 目录里, 文档名为 slocate.db。

## find

```
find pathname -options [-print -exec -ok ...]
```

find 是在硬盘文件树查找。参数:

- pathname: 查找的目录。例如用 . 来表示当前目录, 用 / 来表示系统根目录。
- -name: 按照文件名查找文件。
- -print: 将匹配的文件输出到标准输出。也可以使用 > 或 >> (追加) 写到文件。
- -exec: 对匹配的文件执行该参数所给出的 shell 命令。相应命令的形式为 'command' {} \; , 注意 {} 和 \; 之间的空格。
- -ok: 和 -exec 的作用相同, 只不过以一种更为安全的模式来执行该参数所给出的 shell 命令, 在执行每一个命令之前, 都会给出提示, 让用户来确定是否执行。

### Example

```

1 $ which python
2 /usr/bin/python
3
4 $ whereis -s -S /usr/lib -f python
5 python: /usr/lib/python3.5 /usr/lib/python2.7
6
7 $ locate /usr/bin/pytho ## 以 pytho 开头的文件
8 /usr/bin/python
9 /usr/bin/python-config
10 /usr/bin/python2
11 ...
```

(continues on next page)

(continued from previous page)

```
12  
13 ## 查找 /var/log 中扩展名为 .tmp 的文件，并在删除之前询问用户 (y/n)  
14 $ find /var/log -name "*.tmp" -ok rm {} \;  
15 < rm ... ./t.tmp > ? y
```

### 3.2.4 参考资料

1. runoob.com

<https://www.runoob.com/linux/linux-comm-tee.html>

<https://www.runoob.com/linux/linux-comm-find.html>

2. 每天一个 linux 命令目录

<http://www.cnblogs.com/peida/archive/2012/12/05/2803591.html>

3. aptitude 和 apt-get 的区别和联系【转，有添加和修改】

<https://blog.csdn.net/u010670794/article/details/42520209>

4. apt-get update 与 upgrade 的区别

<https://www.jianshu.com/p/42a1850bdcf6>

## 3.3 权限管理

### 3.3.1 文件描述符

ls -l 命令输出的第一个字段就是描述文件和目录权限的编码。

```
drwxrwxr-x 13 fong fong 4096 5月  7 10:33 source/
```

该字段的第一个字符代表对象的类型：

- -: 文件
- d: 目录
- l: 链接
- c: 字符型设备
- b: 块设备

- n: 网络设备

之后是 3 组三字符的编码，每一组定义了 3 种访问权限（若没有某种权限，使用 - 代替）：

- r: 可读
- w: 可写
- x: 可执行

`ls -F` 能够在可执行文件的文件名后加一个 `*`，目录后方加 `/`。

3 组权限对应对象的 3 个安全等级：

- 对象的属主（登录名 fong）
- 对象的属组（组名 fong）
- 系统其它用户

### 3.3.2 默认文件权限

`umask` 查看默认文件权限，

```

1 $ umask
2 0002
3
4 ## 临时修改默认权限 （若要长期生效，把 umask 0022 写进 ~/.bashrc 并 source）
5 $ umask 0022
6 $ umask
7 0022

```

第 1 位代表粘着位（sticky bit），后 3 位表示文件或目录对应的 `umask` 八进制值（对应 3 组权限）。

Table 1: Linux 文件权限码

权限	二进制值	八进制值
- - -	000	0
- - x	001	1
- w -	010	2
- w x	011	3
r - -	100	4
r - x	101	5
r w -	110	6
r w x	111	7

`umask` 值只是 **掩码**，屏蔽不想授予的权限，即：真正的权限是用 **全权限**值减去 `umask` 值。文件的全权限值是 **666**，目录的全权限值是 **777**。当 `umask=0022`，文件默认权限是 644，目录默认权限是 755。

### 3.3.3 改变权限

```
chmod [-cfvR] [--help] [--version] [mode] [file...]
```

参数:

- c: 若该文件权限确实已经更改, 才打印更改动作
- f: 若该文件权限无法被更改, 也不打印错误讯息
- v: 打印权限变更的详细动作
- R: 对目前目录下的所有文件与子目录进行相同的权限变更 (即以递归的方式逐个变更)

mode:

```
[ugoa...] [+==] [rwxX...]
```

- u 表示对象的属主, g 表示对象的属组成员, o 表示系统其它用户, a 表示所有用户。
- + 表示增加权限、- 表示取消权限、= 表示唯一设定权限。
- r 表示可读, w 表示可写, x 表示可执行。

```
1 $ mkdir test
2 $ ls -l
3 drwxr-xr-x  2 fong fong 4096 5月   7 13:34 test/
4 $ chmod -v 777 test
5 mode of 'test' changed from 0755 (rwxr-xr-x) to 0777 (rwxrwxrwx)
6 $ chmod -v a-w test
7 mode of 'test' changed from 0777 (rwxrwxrwx) to 0555 (r-xr-xr-x)
```

### 3.3.4 参考资料

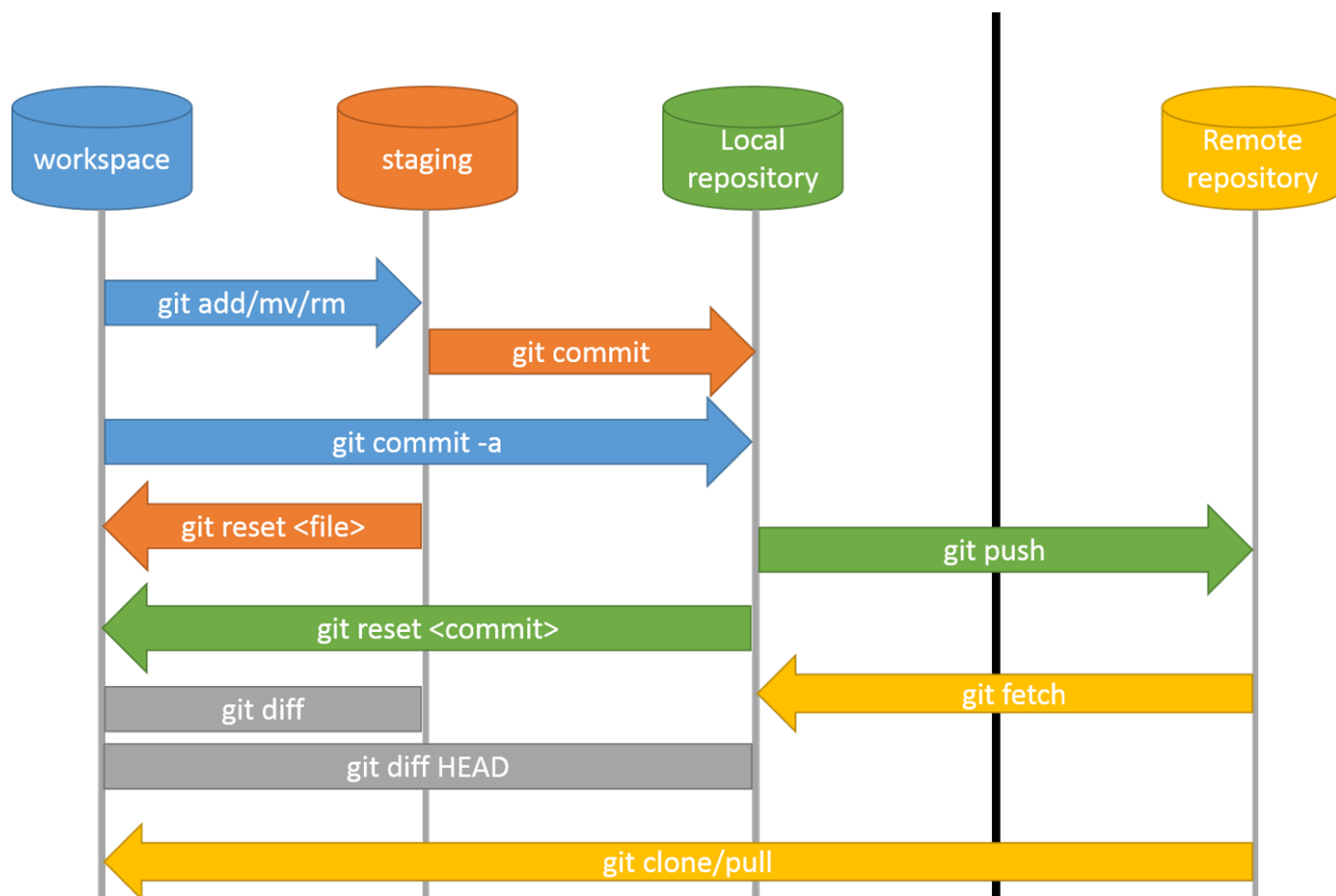
1. 《Linux 命令行与 shell 脚本编程大全》
2. Linux chmod 命令

<https://www.runoob.com/linux/linux-comm-chmod.html>



## 4.1 本地版本库

Git 是目前世界上最先进的分布式版本控制系统。



(图片来源: <http://blog.podrezo.com/git-introduction-for-cvssvntfs-users/>)

- **workspace** : 工作区
- **stage** : 暂存区

- **local repository** : 本地版本库
- **remote repository** : 远程仓库

### 4.1.1 本地版本库

#### 创建与修改

- `git init` 把当前目录变为 Git 可管理的仓库 (目录下多了子目录.git/, 自动创建的第一个分支 master, 以及指向 master 的一个指针叫 HEAD )。
- `git add my_file` 把文件加入暂存区。

---

**Note:** `git add .` : 把工作时的 **所有变化**提交到暂存区, 包括文件内容 **修改 (modified)** 以及 **新文件 (new)** , 但不包括被删除的文件。

`git add -u` : `git add - -update`, 仅监控已经被 add 的文件 (即 **tracked file** ), 他会将被修改的文件提交到暂存区。不会提交新文件 (untracked file)。

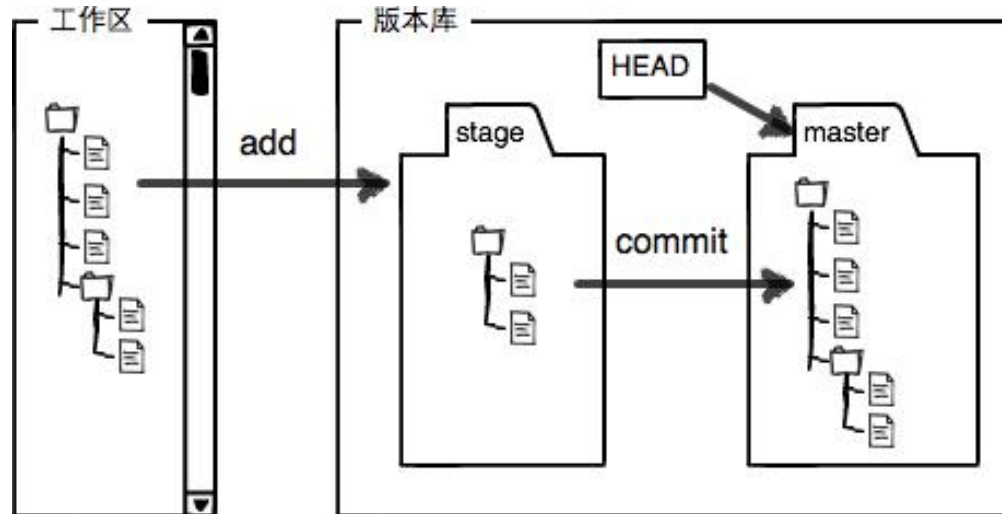
`git add -A` : `git add - -all`, 是上面两个功能的合集。

---

- `git commit -m "add my_file"` 提交到本地版本库, 并写 log。
- `git status` 查看当前仓库的状态 (文件是不是被 tracked? 修改是不是已经 commit?...等)。
- `git diff` 查看当前状态和最新的 commit 之间的不同 (修改还没有 add), 命令可以加具体文件名以查看某个文件的修改。
- `git diff < 版本号`, 如 `7ed6b16`> 查看当前状态和之前某次 commit 之间的不同。
- `git log` 查看 commit 记录。
- `git reflog` 查看之前每次 commit 之后的分支状态。

```
1 $ git reflog
2 41c873a (HEAD -> master) HEAD@{0}: commit: update b
3 3e2b7f2 HEAD@{1}: reset: moving to HEAD
4 3e2b7f2 HEAD@{2}: commit: update out
5 7ed6b16 HEAD@{3}: reset: moving to HEAD
6 7ed6b16 HEAD@{4}: commit: add a
7 8337301 HEAD@{5}: commit (initial): add readme
```

## 版本管理



(图片来源: <https://www.liaoxuefeng.com/wiki/896043488029600/897271968352576>)

**HEAD** 指针指向当前版本的 **master** 分支。

- `git checkout -- my_file` 如果修改或删除了已经 `commit` 的内容, 这条指令可以丢弃该操作, 一键还原。
- `git reset --hard` 撤销修改, 回到上一次 `commit` 之后的状态。
- `git reset --hard < 版本号, 如 7ed6b16>` 回到某一次 `commit` 之后的状态, 同时会删除该次 `commit` 之后的 `commit log`。

### 4.1.2 参考资料

1. Git 和 Github 简单教程

<https://www.cnblogs.com/schaepfer/p/5561193.html#reset>

2. Git 教程

<https://www.liaoxuefeng.com/wiki/896043488029600>

3. Git 使用教程

<http://www.cnblogs.com/tughua0707/p/4050072.html>

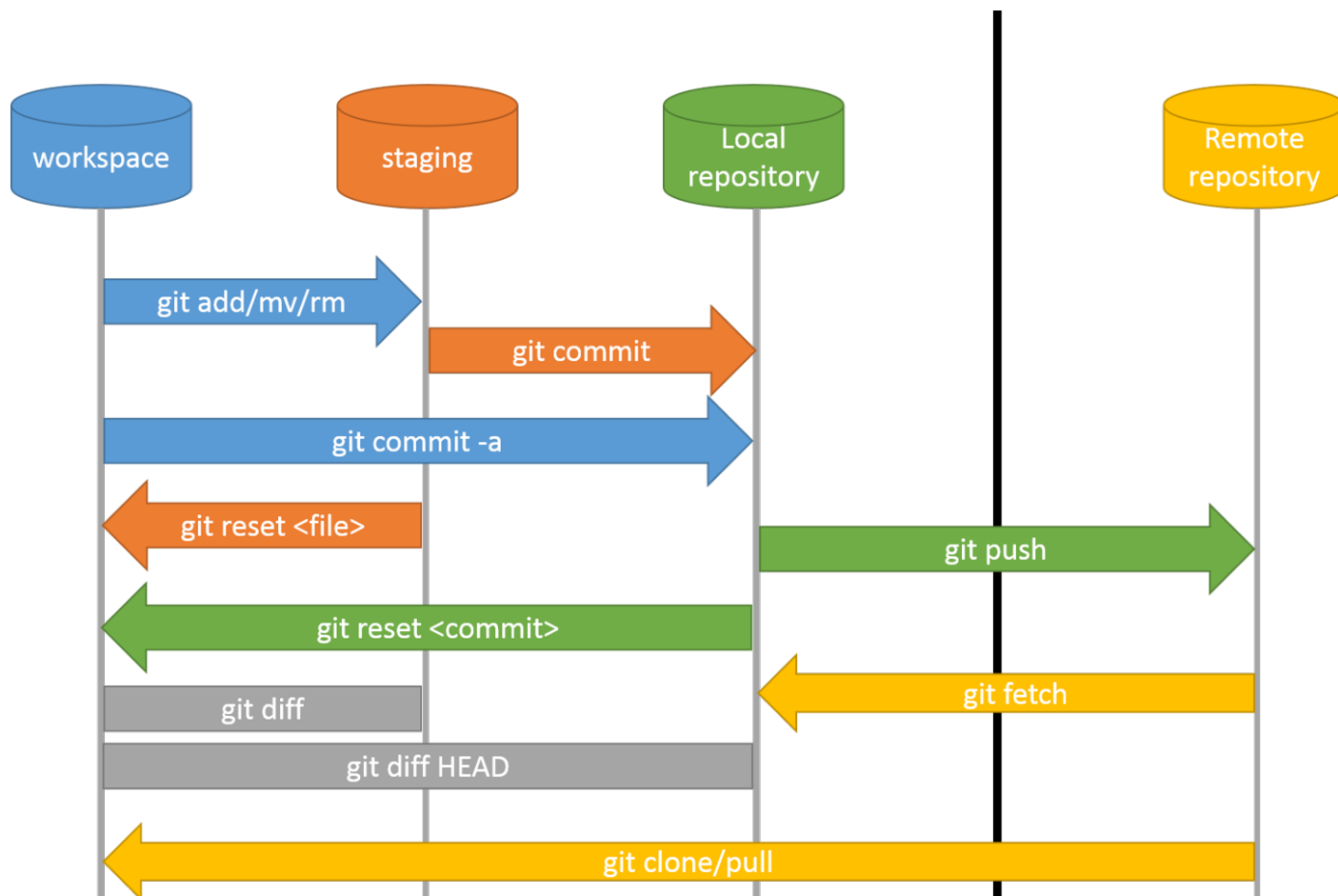
4. Git 操作详解

<https://www.cnblogs.com/bestzhang/p/6903338.html>

5. `git add -A` 和 `git add .` 的区别

<https://www.cnblogs.com/skura23/p/5859243.html>

## 4.2 远程仓库



(图片来源: <http://blog.podrezo.com/git-introduction-for-cvssvntfs-users/>)

- **workspace** : 工作区
- **stage** : 暂存区
- **local repository** : 本地版本库
- **remote repository** : 远程仓库

### 4.2.1 远程仓库

- `git clone < 版本库的网址 >` 从远程主机克隆一个版本库。
- `git remote` 管理主机名, 使用参数 `-v`, 可以参看远程主机的网址。

```
1 $ git remote -v
2 origin  git@github.com:*****/*****.git (fetch)
```

(continues on next page)

(continued from previous page)

```

3 origin git@github.com:*****/*****.git (push)
4 ## 结果表明：当前只有一台远程主机，叫做 origin。

```

- `git fetch < 远程主机名 >` 将某个远程主机的更新，全部取回本地。默认情况下，`git fetch` 取回所有分支（branch）的更新。
- `git fetch < 远程主机名 > < 分支名 >` 如果只想取回特定分支的更新，可以指定分支名，比如 `master`。
- `git branch -r`：查看远程分支，`-a`：查看所有分支（包括本地分支）。
- `git merge origin/master` 在本地分支上合并远程分支（`origin/master`）。
- `git pull < 远程主机名 > < 远程分支名 >:< 本地分支名 >` 取回远程主机某个分支的更新，再与本地的指定分支合并。比如，取回 `origin` 主机的 `next` 分支，与本地的 `master` 分支合并，需要写成 `git pull origin next:master`。如果远程分支是与当前分支合并，可直接写为 `git pull origin next`。等效于 `fetch+merge`：`git fetch origin`，`git merge origin/next`。

**Note:** Git Pull Failed: Your local changes would be overwritten by merge. Commit, stash or revert them.

- 保留未 `push` 的本地代码，并把 `git` 服务器上的代码 `pull` 到本地（本地刚才修改的代码将会被暂时封存起来）。
  - `git stash`
  - `git pull origin master`（其中 `origin master` 表示 `git` 的主分支）
  - …（一些别的操作，直到结束了对 `pull` 到本地的代码的操作。例如，`push` 之后。）
  - `git stash pop`
- 完全地覆盖本地的代码，只保留服务器端代码，则直接回退到上一个版本，再进行 `pull`。
  - `git reset -hard`
  - `git pull origin master`

- `git push < 远程主机名 > < 本地分支名 >:< 远程分支名 >` 将本地分支的更新，推送到远程主机。

## 4.2.2 参考资料

1. Git 和 Github 简单教程

<https://www.cnblogs.com/schaepfer/p/5561193.html#reset>

2. Git 教程

<https://www.liaoxuefeng.com/wiki/896043488029600>

3. Git 使用教程

<http://www.cnblogs.com/tugenhua0707/p/4050072.html>

#### 4. Git 操作详解

<https://www.cnblogs.com/bestzhang/p/6903338.html>

#### 5. Git Pull Failed

<https://blog.csdn.net/gymaisyl/article/details/84899191>

## 4.3 分支管理

### 4.3.1 分支管理

- `git checkout -b dev` 创建分支 `dev`，并切换到 `dev` 分支（此时可以正常 `add`，`commit` 等）。相当于下面两条指令：

- `git branch dev` 创建分支
- `git checkout dev` 切换分支

---

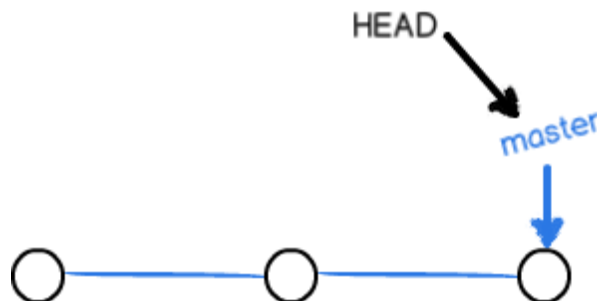
**Note:** `dev` 分支的仓库已经包含了 `master` 分支的内容，但是在 `master` 分支下，无法看到 `dev` 分支新增或修改的内容。

---

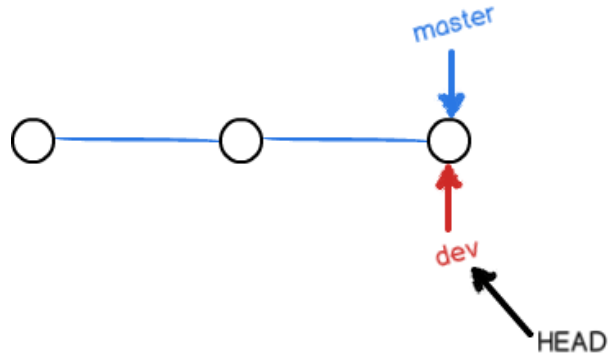
- `git branch` 查看当前分支。
- `git merge dev` 把 `dev` 分支的内容合并到当前分支（如，`master` 分支）。
- `git branch -d dev`（合并之后）删除 `dev` 分支。

流程如下（图片来源于 [Git 教程 - 廖雪峰的官方网站](#)）：

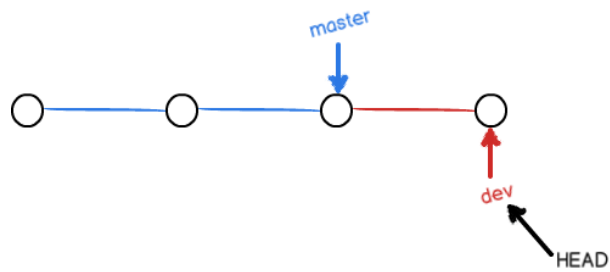
1. 初始状态：**master** 分支。



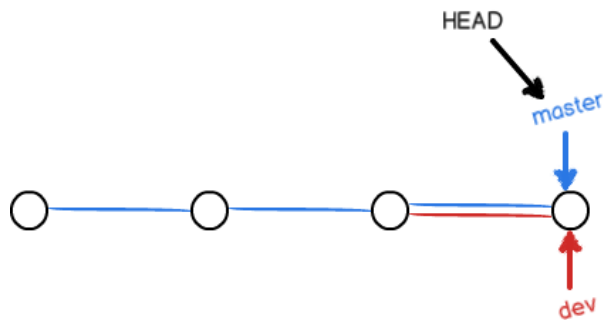
2. 创建并切换到 **dev** 分支。



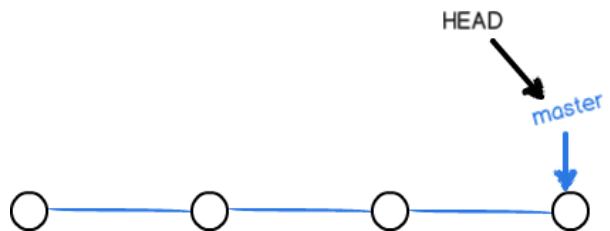
3. 更新 dev 分支。



4. 合并 dev 分支到 master 分支。



5. 删除 dev 分支。



### 4.3.2 参考资料

1. Git 和 Github 简单教程

<https://www.cnblogs.com/schaepfer/p/5561193.html#reset>

2. Git 教程

<https://www.liaoxuefeng.com/wiki/896043488029600>

3. Git 使用教程

<http://www.cnblogs.com/tughenhu0707/p/4050072.html>

4. Git 操作详解

<https://www.cnblogs.com/bestzhang/p/6903338.html>



## 5.1 Logistic Regression

模型：

$$\begin{aligned}h_{\theta}(\mathbf{x}) &= \\g(\theta^{\top} \mathbf{x}), \\g(z) &= \\ \frac{1}{1 + e^{-z}}, \\g'(z) &= \\(1 - g(z))g(z) &\in (0, 0.25].\end{aligned}$$

对数损失函数（极大似然）：

$$\hat{J}_{\theta} = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log h_{\theta}(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\theta}(\mathbf{x}^{(i)}))$$

虽然使用了 sigmoid 函数，但该模型仍然是线性分类器，因为即使不经过 sigmoid 函数也可以得出分类结果（与 0 比较），sigmoid 将其转化为概率。

### 5.1.1 基本假设

1. 数据服从伯努利分布， $y \sim \text{Bernoulli}(\phi)$
2. 样例为正例的概率为  $\phi = h_{\theta}(\mathbf{x})$

### 5.1.2 求解方法

梯度下降

- 批梯度下降：全局最优；每次参数更新需要遍历所有样本，计算量大，效率低。
- 随机梯度下降（SGD）：以高方差频繁更新，能跳到新的、更好的局部最优解；收敛到局部最优的过程更加复杂。

- 小批量梯度下降：减少了参数更新次数，达到更稳定的收敛结果。

### 5.1.3 优缺点

#### 优点

- 模型简单，可解释性好，效果不错
- 训练速度快，资源占用少
- 直接输出样本的分类概率，便于做阈值划分

#### 缺点

- 准确性不高
- 很难处理数据不平衡问题
- 只能处理线性问题
- 逻辑回归本身无法筛选特征

### 5.1.4 解析

1. 为什么使用极大似然函数作为损失函数？

- 极大似然：希望最大化每个样本的分类正确概率，样本服从伯努利分布。
- 将极大似然取对数后就等同于对数损失函数，在 LR 模型中，这个损失函数使参数更新速度较快：

$$\theta_j := \theta_j + \alpha \frac{1}{m} \sum_{i=1}^m (y^{(i)} - h_{\theta}(\mathbf{x}^{(i)})) \mathbf{x}_j^{(i)}$$

只与  $y^{(i)}, \mathbf{x}^{(i)}$  有关，与  $h_{\theta}$  的梯度无关。

- 为什么不用平方损失函数（多用于线性回归）？在线性回归中，前提假设是  $y$  服从正态分布，即  $y \sim \mathcal{N}(\mu, \sigma^2)$ 。另外，如果使用平方损失函数， $\theta$  更新与  $h_{\theta}$  的梯度有关，而 sigmoid 函数的梯度在定义域内小于 0.25，导致参数更新慢。

2. 训练中如何有很多特征高度相关或将某个特征重复 100 遍，影响如何？

如果损失函数收敛，不影响分类结果（每个特征对应的权重  $\theta_j$  变为原来的百分之一）。将相关特征去除，使模型具有更好的解释性，也能加快训练速度。

### 5.1.5 参考资料

1. 逻辑回归的常见面试题总结

<http://www.cnblogs.com/ModifyRong/p/7739955.html>

2. LR 逻辑斯回归分析（优缺点）

[https://blog.csdn.net/touch\\_dream/article/details/79371462](https://blog.csdn.net/touch_dream/article/details/79371462)

3. logistic 回归（内附推导）

<https://www.jianshu.com/p/894bda167422>

4. 周志华《机器学习》Page 57 – 60。

## 5.2 支持向量机

样本空间中任意点到超平面的距离为：

$$r = \frac{|w^T x + b|}{\|w\|}$$

原始问题：

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} \|w\|^2 \\ \text{s.t.} \quad & y_i(w^T x + b) \geq 1, i = 1, 2, \dots, m \end{aligned}$$

拉格朗日函数：

$$\begin{aligned} L(w, b, \alpha) = \frac{1}{2} w^T w + \sum_{i=1}^m \alpha_i (1 - y_i(w^T x + b)), \\ \min_{w,b} (\max_{\alpha_i > 0} L(w, b, \alpha)) \end{aligned}$$

对偶问题：

$$\max_{\alpha_i > 0} (\min_{w,b} L(w, b, \alpha))$$

即：

$$\max_{\alpha_i > 0} \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j y_i y_j x_i^T x_j, \quad w = \sum_{i=1}^m \alpha_i y_i x_i$$

KKT 条件：

$$\begin{aligned} y_i(w^T x + b) &\geq 1, \\ \sum_{i=1}^m \alpha_i y_i &= 0, \\ \alpha_i(1 - y_i(w^T x + b)) &= 0. \end{aligned}$$

### 5.2.1 核函数

核函数  $\mathcal{K}$

- 对称半正定。(  $\mathcal{K} \geq 0 : \forall z, z^T \mathcal{K} z \geq 0.$  )

- 主要使用线性核，高斯核（RBF）。
- 当特征维度高且样本少，不宜使用高斯核，容易过拟合。
- 当特征维度低，且样本够多，考虑使用高斯核。首先需要特征缩放（归一化）。若  $\sigma$  过大，导致特征间差异变小，欠拟合。

### 5.2.2 多分类

1. 一对一 ( $O(N^2)$ )
2. 一对多 ( $O(N)$ )
3. 使用多分类 loss

### 5.2.3 SVM 库

sklearn, libsvm

### 5.2.4 优缺点

#### 优点

- 基于结构风险最小化，泛化能力强（自带正则化， $\|w\|^2$ ）。
- 它是凸优化问题，可得到全局最优。
- SVM 在小样本训练集上可得到比其他方法好的结果。
- 利用核函数，可借助线性可分问题的求解方法，直接求解对应高维空间的问题。

#### 缺点

- SVM 对缺失特征敏感。
- 如何确定核函数？
- 求解问题的二次规划，耗时耗存储。

### 5.2.5 解析

1. 为什么要间隔最大化？  
最优超平面，解唯一，更加鲁棒。
2. 为什么转化为对偶问题？
  - 便于求解（交换  $\alpha$  和  $(w, b)$  位置之后，可直接对  $(w, b)$  求导）。
  - 解的过程可以引入核函数。

### 5.2.6 SVM 与 LR 的异同

相同点：

- 都是分类算法。
- 不考虑核函数，分类面都是线性。
- 都是监督学习算法。
- 都是判别模型。（判别模型：KNN，SVM，LR；生成模型：HMM，朴素贝叶斯）

不同点：

- 本质不同：loss function 不同
- SVM 只有支持向量影响模型，LR 中每个样本都有作用。
- SVM 针对线性不可分问题有核函数。
- SVM 依赖样本间的距离测度，样本特征需要归一化，也就是说 SVM 基于距离，LR 基于概率。
- SVM 是结构风险最小化算法（在训练误差和模型复杂度之间的折中，防止过拟合，从而达到真实误差最小化）。因为 SVM 自带正则（ $\|w\|^2$ ）。

### 5.2.7 参考资料

1. LR 与 SVM 的异同

<https://www.cnblogs.com/zhizhan/p/5038747.html>

2. 核函数

<https://www.cnblogs.com/loujiayu/archive/2013/12/19/3481320.html>

3. SVM 面试题

<https://www.jianshu.com/p/fa02098bc220>

4. SVM 的优缺点

<https://blog.csdn.net/fengzhizhizhizhizi/article/details/23911699>

5. 机器学习技法-SVM 的对偶问题

<https://www.jianshu.com/p/de882f0fc434>

6. 周志华《机器学习》Page 121 – 124。

## 5.3 主成分分析

PCA : Principal Component Analysis.

**最大可分性：**样本点到超平面的投影能尽可能分开（投影后样本点方差最大化）。

### 5.3.1 优化目标

$$\begin{aligned} \max_W \quad & \text{tr}(W^T X X^T W) \\ \text{s.t.} \quad & W^T W = I. \\ & X \in \mathbb{R}^{d \times m}, \\ & W \in \mathbb{R}^{d \times d'}, \quad d' < d. \end{aligned}$$

### 5.3.2 求解

1. 计算样本的协方差矩阵  $C = X X^T$  ；
2. 对协方差矩阵做特征值分解（EVD）；
3. 取最大的  $d'$  个特征值  $(\lambda_1, \lambda_2, \dots, \lambda_{d'})$  对应的特征向量：

$$W = (w_1, w_2, \dots, w_{d'})$$

### 5.3.3 PCA 白化

白化的目的是降低输入的冗余性：

- 特征之间相关性降低
- 所有特征具有相同的方差

$$\begin{aligned} x_{rot} &= W^T x, \\ x_{pca \text{ white}, i} &= \frac{x_{rot, i}}{\sqrt{\lambda_i}} \end{aligned}$$

### 5.3.4 SVD 分解

$$\begin{aligned} A &= U \Sigma V^T, \\ A &\in \mathbb{R}^{m \times n}, \\ \text{rank}(A) &= r, \\ U &\in \mathbb{R}^{m \times r}, \\ \Sigma &\in \mathbb{R}^{r \times r}, \\ V &\in \mathbb{R}^{n \times r} \end{aligned}$$

其中  $U$  是  $AA^T$  的特征向量矩阵,  $V$  是  $A^T A$  的特征向量矩阵。

当  $d$  很大时,  $C = XX^T$  是很高维的矩阵, 计算该矩阵并求特征向量开销大。此时对  $X$  做 SVD 分解, 得到  $U$  便是协方差矩阵  $C$  的特征向量。

### 5.3.5 参考资料

1. 周志华《机器学习》Page 229 – 232。
2. ufdl

<http://ufdl.stanford.edu/wiki/index.php/PCA>

## 5.4 常用矩阵求导公式

导数有两种不同的表示形式:

### Pictorial Representation

	$\frac{\cdot}{\cdot}$	$\frac{\boxed{\phantom{0}}}{\cdot}$	$\frac{\boxed{\phantom{0}}}{\cdot}$	$\frac{\cdot}{\boxed{\phantom{0}}}$	$\frac{\boxed{\phantom{0}}}{\boxed{\phantom{0}}}$	$\frac{\cdot}{\boxed{\phantom{0}}}$
numerator layout	$\cdot$	$\boxed{\phantom{0}}$	$\boxed{\phantom{0}}$	$\boxed{\phantom{0}}$	$\boxed{\phantom{0}}$	$\boxed{\phantom{0}}$
denominator layout	$\cdot$	$\boxed{\phantom{0}}$		$\boxed{\phantom{0}}$	$\boxed{\phantom{0}}$	$\boxed{\phantom{0}}$

这里使用 **denominator layout**。

$x, a, b$  均为 **列向量**。

### 5.4.1 一阶

$$\begin{aligned}\frac{\partial(\mathbf{x}^\top \mathbf{a})}{\partial \mathbf{x}} &= \\ \frac{\partial(\mathbf{a}^\top \mathbf{x})}{\partial \mathbf{x}} &= \mathbf{a} \\ \frac{\partial(\mathbf{a}^\top \mathbf{X} \mathbf{b})}{\partial \mathbf{X}} &= \\ \mathbf{a} \mathbf{b}^\top & \\ \frac{\partial(\mathbf{a}^\top \mathbf{X}^\top \mathbf{b})}{\partial \mathbf{X}} &= \\ \mathbf{b} \mathbf{a}^\top & \\ \frac{\partial(\mathbf{a}^\top \mathbf{X} \mathbf{a})}{\partial \mathbf{X}} &= \\ \mathbf{a} \mathbf{a}^\top &\end{aligned}$$

### 5.4.2 二阶

$$\begin{aligned}\frac{\partial(\mathbf{x}^\top \mathbf{x})}{\partial \mathbf{x}} &= \\ 2\mathbf{x} & \\ \frac{\partial(\mathbf{x}^\top \mathbf{B} \mathbf{x})}{\partial \mathbf{x}} &= \\ (\mathbf{B} + \mathbf{B}^\top) \mathbf{x} & \\ \frac{\partial(\mathbf{a}^\top \mathbf{X}^\top \mathbf{X} \mathbf{b})}{\partial \mathbf{X}} &= \\ \mathbf{X}(\mathbf{a} \mathbf{b}^\top + \mathbf{b} \mathbf{a}^\top) &\end{aligned}$$



### 5.4.3 迹

$$\text{Tr}(\mathbf{A}^\top \mathbf{B}) = \text{Tr}(\mathbf{B}^\top \mathbf{A})$$

$$\begin{aligned} \frac{\partial(\text{Tr}(\mathbf{X}))}{\partial \mathbf{X}} &= \mathbf{I} \\ \frac{\partial(\text{Tr}(\mathbf{X}\mathbf{A}))}{\partial \mathbf{X}} &= \mathbf{A}^\top \\ \frac{\partial(\text{Tr}(\mathbf{X}^\top \mathbf{A}))}{\partial \mathbf{X}} &= \mathbf{A} \\ \frac{\partial(\text{Tr}(\mathbf{A}\mathbf{X}^\top))}{\partial \mathbf{X}} &= \mathbf{A} \\ \frac{\partial(\text{Tr}(\mathbf{A}\mathbf{X}\mathbf{B}))}{\partial \mathbf{X}} &= \mathbf{A}^\top \mathbf{B}^\top \\ \frac{\partial(\text{Tr}(\mathbf{A}\mathbf{X}^\top \mathbf{B}))}{\partial \mathbf{X}} &= \mathbf{B}\mathbf{A} \\ \frac{\partial(\text{Tr}(\mathbf{X}\mathbf{X}^\top))}{\partial \mathbf{X}} &= 2\mathbf{X} \\ \frac{\partial(\text{Tr}(\mathbf{X}^2))}{\partial \mathbf{X}} &= 2\mathbf{X}^\top \\ \frac{\partial(\text{Tr}(\mathbf{X}\mathbf{B}\mathbf{X}^\top))}{\partial \mathbf{X}} &= \mathbf{X}\mathbf{B}^\top + \mathbf{X}\mathbf{B} \\ \frac{\partial(\text{Tr}(\mathbf{X}^\top \mathbf{B}\mathbf{X}))}{\partial \mathbf{X}} &= \mathbf{B}\mathbf{X} + \mathbf{B}^\top \mathbf{X} \end{aligned}$$

### 5.4.4 参考资料

1. The Matrix Cookbook

<https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf>

2. Matrix Derivatives

<https://www.comp.nus.edu.sg/~cs5240/lecture/matrix-differentiation.pdf>



## 6.1 pytorch: 多 GPU 模式

pytorch 中可以通过 `torch.nn.DataParallel` 切换到多 GPU(multi-GPU) 模式, 有两种使用方式: 网络外指定、网络内指定。

### 6.1.1 网络外指定

使用方法:

```
1 # 在 GPU 上运行
2 model.cuda()
3 # 使用第 0、1、2 个 GPU, 注意设定 batch_size 大一些, 否则数据不足以跑多 GPU
4 model = torch.nn.parallel.DataParallel(model, device_ids=[0, 1, 2])
```

`DataParallel` 只对 `forward()` 和 `backward()` 有效, 直接调用 `model` 中自定义的 `attribute` 如 `forward_1()` 无效。

另外, 在 `DataParallel` 模式下, 引用 `model` 的 `attribute` 必须采用如下格式:

```
# 相比于 'model.attribute' 多了 'module'。
model.module.attribute
```

比如, `model.module.classifier.parameters()` 。

### 6.1.2 网络内指定

使用方法:

```
1 # 定义网络结构
2 self.layer1 = nn.Linear(227, 128)
3 self.layer1 = nn.DataParallel(self.layer1, device_ids=[0, 1, 2])
```

在 CPU 模式下不需要更改代码。

### 6.1.3 参考资料

1. pytorch documentation

<https://pytorch.org/docs/stable/nn.html#torch.nn.DataParallel>

2. 网络内指定

[https://ptorch.com/docs/3/parallelism\\_tutorial](https://ptorch.com/docs/3/parallelism_tutorial)

3. 引用 attribute

<https://discuss.pytorch.org/t/how-to-reach-model-attributes-wrapped-by-nn-dataparallel/1373>

## 6.2 激活函数

神经网络引入激活函数（Activation Function）主要是为了增强网络的非线性，提升网络的拟合能力和学习能力。激活函数有以下几个性质：

- 非线性
- 可微性
- 单调性：保证单层网络是凸函数

下面介绍 **sigmoid**、**tanh** 以及 **ReLU**。

### 6.2.1 sigmoid

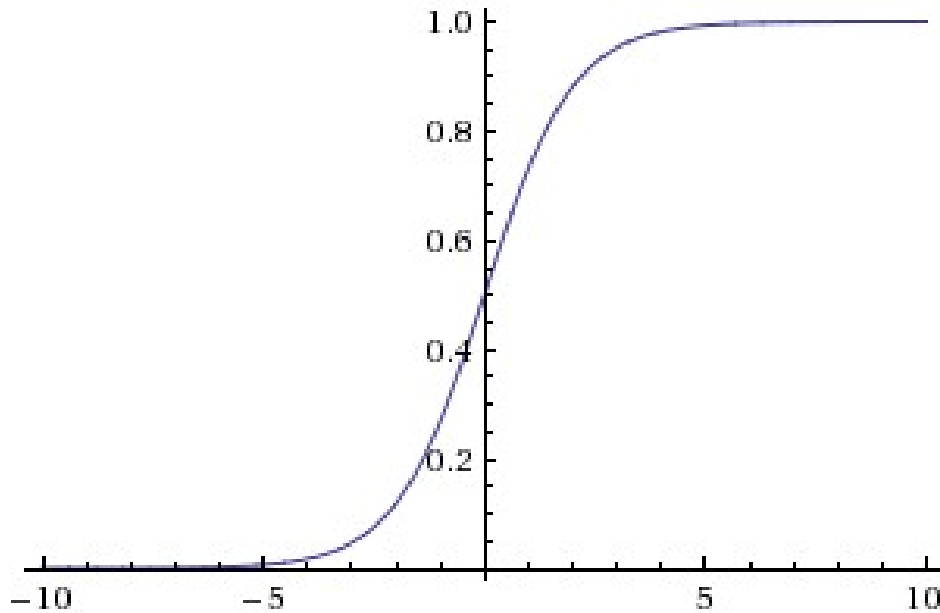
**sigmoid** 函数的数学表达式如下：

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

其导数具有如下性质：

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

**sigmoid** 函数能够把输入的连续值压缩到 (0, 1) 范围内，其函数曲线如下：



优点:

- 单调连续, 输出范围有限, 优化稳定
- 求导容易

缺点:

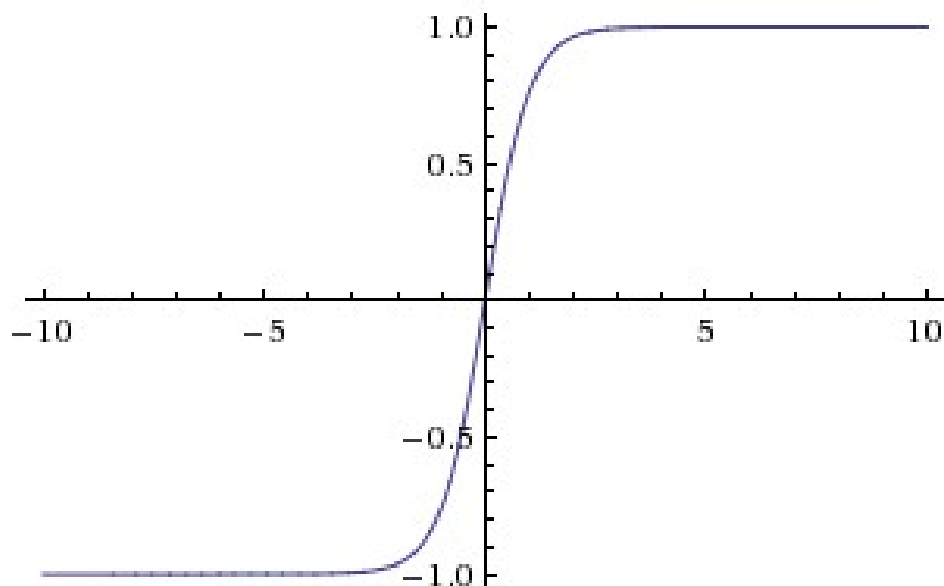
- 容易饱和。当输入很大、很小时 (saturation, 饱和), 神经元的梯度接近 0, 出现“梯度消失” (gradient vanishing), 导致无法完成深层网络的训练。
- 输出不是零均值的 (not zero-centered)。假设某个神经元的输入一直是正的, 即  $x > 0$ 。对于  $f(x) = w^T x + b$ , 则  $w$  获得的梯度将是恒正或者恒负 (取决于  $f$  得到的梯度的符号), 导致  $w$  的更新非常“曲折” (zig-zagging)。当然, 如果是按 batch 训练, 最终梯度是各个样本下梯度的和, 而每个样本下的梯度可能是符号各异的, 因此在一定程度上可以缓解这个问题。

### 6.2.2 tanh

**tanh** 函数的数学表达式如下:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1.$$

其函数曲线如下:



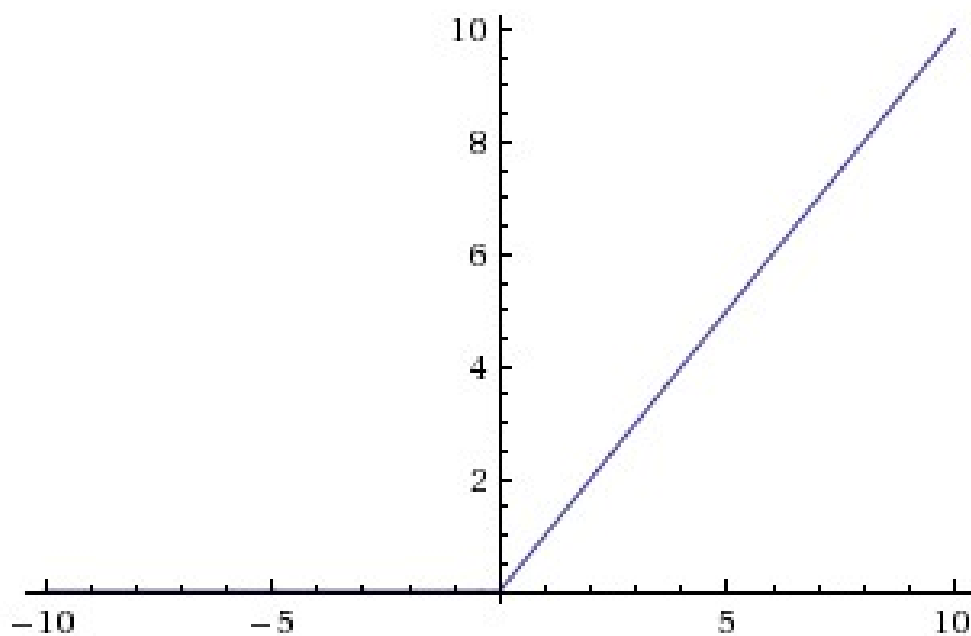
与 `sigmoid` 一样，`tanh` 也会产生饱和现象，但是 `tanh` 的输出是零均值的 (zero-centered)。

### 6.2.3 ReLU

`ReLU` 函数的数学表达式如下：

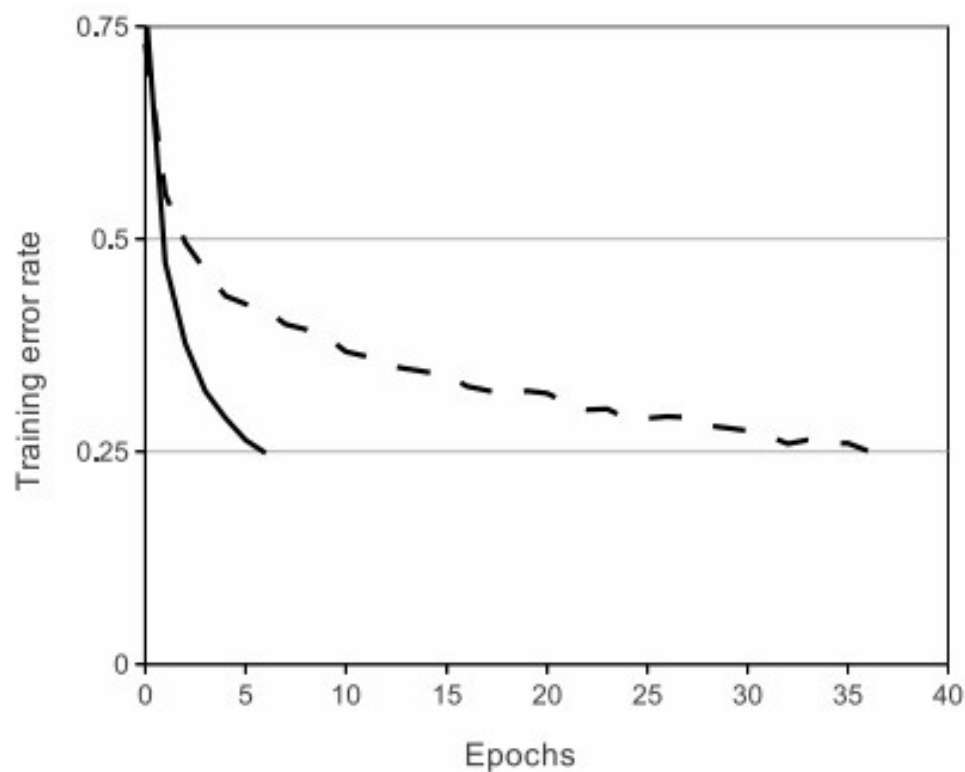
$$\text{relu}(z) = \max(0, z).$$

其函数曲线如下：



优点:

- 计算简单。**sigmoid** 和 **tanh** 都需要计算指数。
- 收敛速度快。Krizhevsky et al. 论文 指出 **ReLU** 收敛速度比 **tanh** 快 6 倍。



缺点:

- 容易产生死亡节点 (dead ReLU)。一个非常大的梯度流过一个 **ReLU** 神经元，更新过参数之后，这个神经元对很多输入数据都输出 0，则梯度一直为 0。当然 **ReLU** 的输出依靠  $w$  和  $x$  的共同作用，死亡节点可能会被重新激活。

**LeakyReLU** 可以有效应对上述缺点。

## 6.2.4 参考资料

1. CS231n

<http://cs231n.github.io/neural-networks-1/#actfun>

2. 神经网络之激活函数 (Activation Function)

<https://blog.csdn.net/memray/article/details/51442059>

3. What is the “dying ReLU” problem in neural networks?

<https://www.quora.com/What-is-the-dying-ReLU-problem-in-neural-networks>

## 6.3 Batch Normalization

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}] + \epsilon}}$$
$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

### 6.3.1 BN 允许更大的学习率

由于网络参数不断更新，导致各层输入的分布不断变化，导致往往需要使用较小的学习率，并精心设计参数初始化。

使用 BN 进行归一化之后，各层输入的分布相同，因此可以使用更大的学习率更快地收敛，并降低网络对初始化的依赖。

### 6.3.2 BN 提供了正则化的作用

进行 BN 之后，各样本的 feature map 已经融合了一个 batch 之中其他样本的特性（均值，方差），因此单一样本的影响变小，网络更好学习整体的规律，有效地减小了过拟合的可能性。因此，可以不使用 dropout。

### 6.3.3 BN 回滚

如果网络发现这种 normalization 是多余的，可以通过学习使得：

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$
$$\beta^{(k)} = E[x^{(k)}]$$

从而消除 BN 的作用。

### 6.3.4 缺点

BN 统计均值、方差与 batch size 有关，batch size 太小会导致性能变差。而某些任务受内存限制，batch size 难以设置很大，因此 BN 作用难以显现。这时候出现了 Group Normalization。

### 6.3.5 参考资料

1. Batch Normalization

<https://arxiv.org/pdf/1502.03167.pdf>

2. Group Normalization

[http://openaccess.thecvf.com/content\\_ECCV\\_2018/papers/Yuxin\\_Wu\\_Group\\_Normalization\\_ECCV\\_2018\\_paper.pdf](http://openaccess.thecvf.com/content_ECCV_2018/papers/Yuxin_Wu_Group_Normalization_ECCV_2018_paper.pdf)



## 6.4 过拟合

复杂的模型将训练数据的抽样误差考虑在内，对抽样误差也进行了拟合。过拟合的模型可以看成是完全记忆型模型。

### 6.4.1 表现

训练误差小，测试误差大，泛化能力差。

### 6.4.2 原因

训练集大小与模型复杂度不匹配；

样本的噪声太大甚至掩盖了真实样本的分布规律；

训练迭代次数太多（over-training）。

### 6.4.3 解决方案

1. 调小模型复杂度。
2. data augmentation.
3. dropout.
4. early stopping. 记录观察 validation accuracy，及时停止训练。
5. 集成学习。Bagging：并行化模型生成，减小模型 variance。Boosting：串行化模型生成，减小模型 bias。
6. 正则化。

<1>. L0 正则化（非 0 元素个数），难以优化求解（NP-hard）。

<2>. L1 正则化（元素绝对值之和，Lasso regression），是 L0 范数的最优凸近似，使权值稀疏。  
权值稀疏的好处：特征选择 && 可解释性。

<3>. L2 正则化（元素平方和，Ridge regression / weight decay），使权值分布均匀且值较小。

## 6.5 pytorch：模型保存与读取

### 6.5.1 简单

```
import torch
## save
torch.save(model, 'model.pkl')
```

(continues on next page)

(continued from previous page)

```
## load
model = torch.load('model.pkl')
```

这种方法存储的模型包括了模型框架及模型参数，一般存储的 pkl 文件较大。

## 6.5.2 详细

模型除了本身的框架、参数信息，还应包括训练的信息，比如训练迭代次数、优化器参数等。

```
1  import torch
2  import shutil
3
4  ## save
5  def save_checkpoint(state, is_best, save_path, filename):
6      filename = os.path.join(save_path, filename)
7      torch.save(state, filename)
8      if is_best:
9          bestname = os.path.join(save_path, 'model_best.pth.tar')
10         shutil.copyfile(filename, bestname)
11
12     save_checkpoint({
13         'epoch': cur_epoch,
14         'state_dict': model.state_dict(),
15         'best_prec': best_prec,
16         'loss_train': loss_train,
17         'optimizer': optimizer.state_dict(),
18     }, is_best, save_path, 'epoch-{}_checkpoint.pth.tar'.format(cur_epoch))
19
20     ## load
21     def load_checkpoint(checkpoint, model, optimizer):
22         """ loads state into model and optimizer and returns:
23             epoch, best_precision, loss_train[]
24             e.g., model = alexnet(pretrained=False)
25         """
26         if os.path.isfile(load_path):
27             print("> loading checkpoint '{}'".format(load_path))
28             checkpoint = torch.load(load_path)
29             epoch = checkpoint['epoch']
30             best_prec = checkpoint['best_prec']
31             loss_train = checkpoint['loss_train']
32             model.load_state_dict(checkpoint['state_dict'])
33             optimizer.load_state_dict(checkpoint['optimizer'])
34             print("> loaded checkpoint '{}' (epoch {})"
```

(continues on next page)

(continued from previous page)

```

35         .format(epoch, checkpoint['epoch']))
36     return epoch, best_prec, loss_train
37 else:
38     print("=> no checkpoint found at '{}'.format(load_path))
39     # epoch, best_precision, loss_train
40     return 1, 0, []

```

### 6.5.3 load 部分参数

当我们只需要从 `state_dict()` load 部分模型参数是，可以采用如下方法：

```

1  # args has the model name, num classes and other irrelevant stuff
2  pretrained_state = model_zoo.load_url(model_names[args.arch])
3  model_state = my_model.state_dict()
4  pretrained_state = { k:v for k,v in pretrained_state.iteritems() if k in model_state and v.size()
↳ == model_state[k].size() }
5  model_state.update(pretrained_state)
6  my_model.load_state_dict(model_state)

```

### 6.5.4 参考资料

1. Saving and loading a model in Pytorch?

<https://discuss.pytorch.org/t/saving-and-loading-a-model-in-pytorch/2610>

2. How to load part of pre trained model?

<https://discuss.pytorch.org/t/how-to-load-part-of-pre-trained-model/1113/8>

## 6.6 pytorch: cuda()

### 6.6.1 使用指定 GPU

- 直接终端中设定

```
CUDA_VISIBLE_DEVICES=1 python my_script.py
```

- 代码中设定

```
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "2"
```

- 使用函数 `set_device`

```
import torch
torch.cuda.set_device(1)
```

## 6.6.2 `cuda()`

对于一个 `tensor` 对象, `cuda()` 返回该对象在 CUDA 内存中的拷贝

```
obj = obj.cuda()
```

对于一个 `nn.Module` 实例, `cuda()` 直接将该模型的参数和 buffers 转移到 GPU。

```
model.cuda()
```

## 6.6.3 参考资料

1. PyTorch 中使用指定的 GPU

<https://www.cnblogs.com/darkknightzh/p/6836568.html>

2. pytorch documentation

<https://pytorch.org/docs/0.3.1/tensors.html?highlight=cuda#torch.Tensor.cuda>

<https://pytorch.org/docs/0.3.1/nn.html?highlight=cuda#torch.nn.Module.cuda>

## 6.7 反向传播

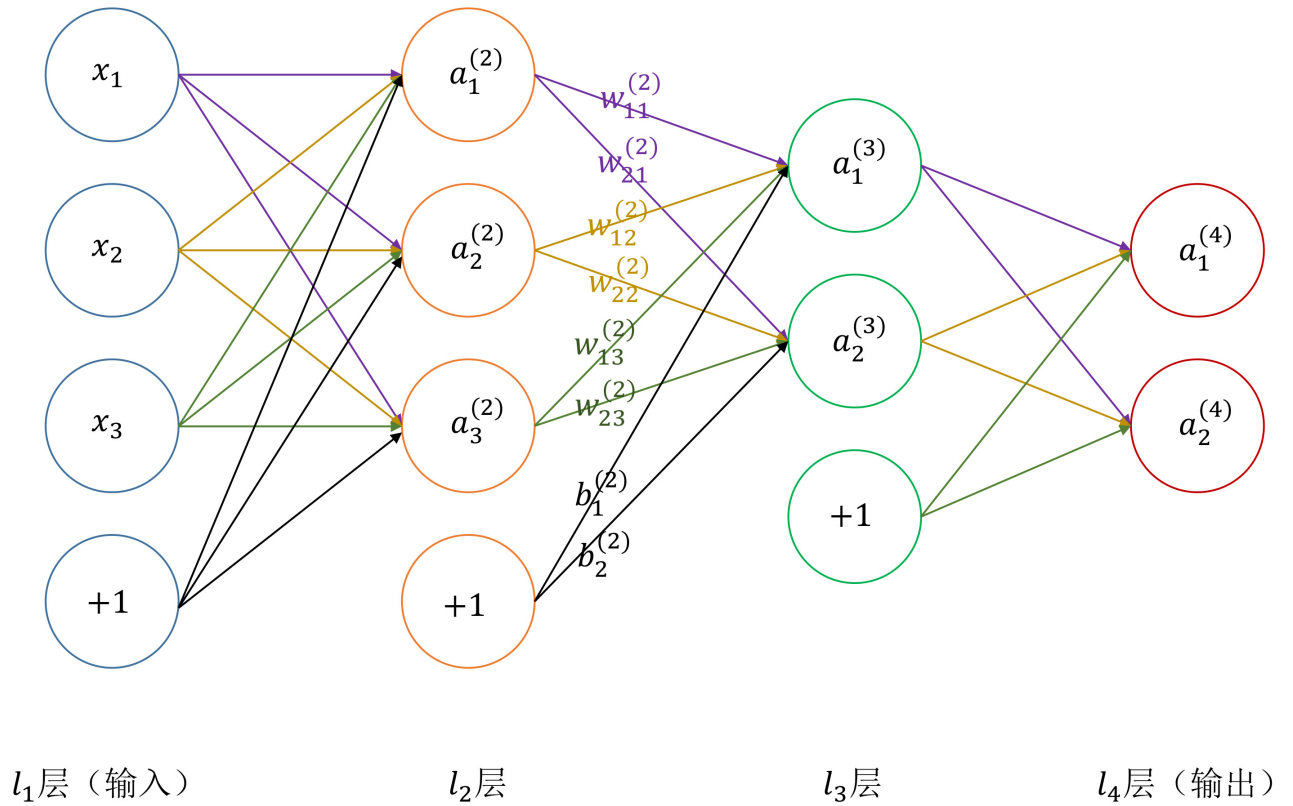


Table 1: 符号说明

符号	含义
$n$	网络层数
$C_l$	第 $l$ 层神经元个数（不包括偏置）
$g(x)$	激活函数
$w_{ji}^{(l)}$	第 $l$ 层第 $i$ 个神经元与第 $l+1$ 层第 $j$ 个神经元的连接权重
$b_i^{(l)}$	第 $l+1$ 层第 $i$ 个神经元的偏置
$z_i^{(l)}$	第 $l$ 层第 $i$ 个神经元的输入
$a_i^{(l)}$	第 $l$ 层第 $i$ 个神经元的激活值
$\delta_i^{(l)}$	第 $l$ 层第 $i$ 个神经元的误差（error）
$y_j$	标签第 $j$ 维（第 $j$ 类）
$\mathcal{L}_{w,b}(x)$	损失函数，简称 $\mathcal{L}$
$x$	训练样本
$m$	小批量训练样本个数

### 6.7.1 前向传播

$$\begin{aligned} z_i^{(l+1)} &= \\ b_i^{(l)} + \sum_{j=1}^{C_l} w_{ij}^{(l)} a_j^{(l)}, \\ g(t) &= \\ \frac{1}{1 + e^{-t}}, \\ g'(t) &= \\ (1 - g(t))g(t), \\ a_i^{(l)} &= \\ g(z_i^{(l)}). \end{aligned}$$

误差定义为：

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}}{\partial z_i^{(l)}}$$

### 6.7.2 误差反向传播

#### MSE (Mean Squared Error)

对每一个样本，损失为：

$$\mathcal{L} = \frac{1}{2} \sum_{j=1}^{C_n} (y_j - a_j^{(n)})^2.$$

最后一层的误差：

$$\begin{aligned} \delta_i^{(n)} &= \\ \frac{\partial \mathcal{L}}{\partial z_i^{(n)}} &= \\ \frac{1}{2} \frac{\partial \left[ \sum_{j=1}^{C_n} (y_j - a_j^{(n)})^2 \right]}{\partial z_i^{(n)}} &= \\ \frac{1}{2} \frac{\partial \left[ (y_i - g(z_i^{(n)}))^2 \right]}{\partial z_i^{(n)}} &= \\ -(y_i - g(z_i^{(n)}))g'(z_i^{(n)}) \end{aligned}$$

递推前层误差：

$$\begin{aligned}
 \delta_i^{(l)} &= \frac{\partial \mathcal{L}}{\partial z_i^{(l)}} \\
 &= \sum_{j=1}^{C_{l+1}} \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \\
 &= \sum_{j=1}^{C_{l+1}} \frac{\partial \mathcal{L}}{\partial z_j^{(l+1)}} \frac{\partial \left( b_i^{(l)} + \sum_{k=1}^{C_l} w_{jk}^{(l)} a_k^{(l)} \right)}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \\
 &= \sum_{j=1}^{C_{l+1}} \delta_j^{(l+1)} w_{ji}^{(l)} g'(z_i^{(l)}) \\
 &= g'(z_i^{(l)}) \sum_{j=1}^{C_{l+1}} \delta_j^{(l+1)} w_{ji}^{(l)}
 \end{aligned}$$

权重和偏置的梯度：

$$\begin{aligned}
 \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} &= \frac{\partial \mathcal{L}}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial w_{ij}^{(l)}} \\
 &= \delta_i^{(l+1)} \frac{\partial z_i^{(l+1)}}{\partial w_{ij}^{(l)}} \\
 &= \delta_i^{(l+1)} \frac{\partial \left( b_i^{(l)} + \sum_{k=1}^{C_l} w_{ik}^{(l)} a_k^{(l)} \right)}{\partial w_{ij}^{(l)}} \\
 &= \delta_i^{(l+1)} a_j^{(l)} \\
 &= \delta_i^{(l+1)}
 \end{aligned}$$

梯度下降

- 权重更新

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \alpha \times \frac{1}{m} \sum_x \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = w_{ij}^{(l)} - \frac{\alpha}{m} \sum_x \delta_i^{(l+1)} a_j^{(l)}$$

- 偏置更新

$$b_i^{(l)} \leftarrow b_i^{(l)} - \alpha \times \frac{1}{m} \sum_x \frac{\partial \mathcal{L}}{\partial b_i^{(l)}} = b_i^{(l)} - \frac{\alpha}{m} \sum_x \delta_i^{(l+1)}$$

## Cross Entropy (交叉熵)

损失函数为:

$$\begin{aligned}\mathcal{L} &= - \sum_{j=1}^{C_n} y_j \ln \hat{y}_j, \\ y_j &\in \{0, 1\}, \\ \hat{y}_j &= \text{softmax}(\mathcal{D}^{(n)}, j) = \frac{e^{a_j^{(n)}}}{\sum_{k=1}^{C_n} e^{a_k^{(n)}}}.\end{aligned}$$

softmax 偏导为:

$$\frac{\partial \hat{y}_j}{\partial a_i^{(n)}} = \begin{cases} -\hat{y}_j \hat{y}_i & i \neq j \\ \hat{y}_i (1 - \hat{y}_i) & i = j \end{cases}$$

另外, 由链式法则 (chain rule):

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial z_i^{(n)}} &= \\ \frac{\partial \mathcal{L}}{\partial a_i^{(n)}} \frac{\partial a_i^{(n)}}{\partial z_i^{(n)}} &= \\ \frac{\partial \mathcal{L}}{\partial a_i^{(n)}} &= \\ \sum_{j=1}^{C_n} \frac{\partial \mathcal{L}}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial a_i^{(n)}} &= \\ \frac{\partial \mathcal{L}}{\partial \hat{y}_j} &= \\ -\frac{y_j}{\hat{y}_j} &\end{aligned}$$



可推得：

$$\begin{aligned}
 \frac{\partial \mathcal{L}}{\partial a_i^{(n)}} &= \\
 \sum_{j=1}^{C_n} \frac{\partial \mathcal{L}}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial a_i^{(n)}} &= \\
 \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial a_i^{(n)}} + \sum_{j \neq i}^{C_n} \frac{\partial \mathcal{L}}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial a_i^{(n)}} &= \\
 -\frac{y_i}{\hat{y}_i} \times \hat{y}_i (1 - \hat{y}_i) + \sum_{j \neq i}^{C_n} -\frac{y_j}{\hat{y}_j} \times (-\hat{y}_j \hat{y}_i) &= \\
 -y_i \times (1 - \hat{y}_i) + \sum_{j \neq i}^{C_n} y_j \times \hat{y}_i &= \\
 -y_i + \sum_{j=1}^{C_n} y_j \times \hat{y}_i &= \\
 -y_i + \hat{y}_i
 \end{aligned}$$

最后一层的误差：

$$\begin{aligned}
 \delta_i^{(n)} &= \\
 \frac{\partial \mathcal{L}}{\partial z_i^{(n)}} &= \\
 \frac{\partial \mathcal{L}}{\partial a_i^{(n)}} \frac{\partial a_i^{(n)}}{\partial z_i^{(n)}} &= \\
 (-y_i + \hat{y}_i) g'(z_i^{(n)})
 \end{aligned}$$

### 6.7.3 参考资料

1. 反向传播公式推导

[https://www.cnblogs.com/nowgood/p/backprop2.html#\\_nav\\_0](https://www.cnblogs.com/nowgood/p/backprop2.html#_nav_0)

2. 神经网络-反向传播详细推导过程

[https://blog.csdn.net/qq\\_29762941/article/details/80343185](https://blog.csdn.net/qq_29762941/article/details/80343185)



## 资源链接

### 7.1 Github Page

<https://fongyq.github.io/>

### 7.2 AnyKnew

<https://www.anyknew.com/#/>

### 7.3 arXiv

<https://arxiv.org/>

### 7.4 Read the Docs

<https://readthedocs.org/>

### 7.5 C++ Reference

<http://www.cplusplus.com/reference/>

### 7.6 Numpy

<http://cs231n.github.io/python-numpy-tutorial/>

## 7.7 Pytorch

Tutorials: <https://pytorch.org/tutorials/>

Docs: <https://pytorch.org/docs/master/index.html>

## 7.8 Jupyter Notebook

<https://jupyter.org/>

## 7.9 Stanford University Lectures

CS229: <http://cs229.stanford.edu/syllabus.html>

CS231: <http://cs231n.github.io/>

## 7.10 ShareLatex

<https://www.sharelatex.com/login>

## 7.11 PlanetB

<http://www.planetb.ca/syntax-highlight-word>

## 7.12 Vision Open Source Library

检索: <http://yael.gforge.inria.fr/index.html>

特征: <http://www.vlfeat.org/index.html>

## 7.13 牛客网

<https://www.nowcoder.com/>

## 8.1 Listary

---

**Note:** Windows 下快速查找文件及应用程序

<http://www.listary.com/>

---

## 8.2 FreeCommander

---

**Note:** Windows 下的资源管理器

<https://freecommander.com/en/summary/>

---

## 8.3 MobaXterm

---

**Note:** Windows 下连接服务器的终端

<https://mobaxterm.mobatek.net/>

---

## 8.4 TeamViewer

---

**Note:** 远程连接

---

<https://www.teamviewer.com/zhCN/>

---

## 8.5 Notepad++

---

**Note:** 强大的文本阅读/编辑器

<https://notepad-plus-plus.org/>

---

## 8.6 cmdr

---

**Note:** Windows 下终端神器

<https://cmder.net/>

---

## 9.1 技巧

- 怎样安装 Windows 7 与 Linux 的双系统?

<https://www.zhihu.com/question/19867618?sort=created>

- Windows10 终端优化方案: Ubuntu 子系统 +cmdr+oh-my-zsh

<https://zhuanlan.zhihu.com/p/34152045>

<https://www.jianshu.com/p/dc32a75e2de4>

- 将 DOS 格式文本文件转换成 UNIX 格式

<https://codingstandards.iteye.com/blog/810900>

- 加速 git clone

<https://blog.51cto.com/11887934/2051323>

<https://blog.lizebang.top/2018/02/git-clone-slow/>

## 9.2 问题

- Unable to locate package python-pip

<https://askubuntu.com/questions/268539/unable-to-locate-package-python-pip-on-live-system>





## 10.1 复习

### 10.1.1 汇总

1. github
  - [https://github.com/imhuay/Algorithm\\_Interview\\_Notes-Chinese](https://github.com/imhuay/Algorithm_Interview_Notes-Chinese)
  - <https://github.com/jwasham/coding-interview-university/blob/master/translations/README-cn.md>
2. 2018 校招算法岗面试题汇总  
<https://zhuanlan.zhihu.com/p/36801851>

### 10.1.2 编程算法

1. 动态规划
  - 有面值 1,5,10,20,50,100 的人民币，求问 10000 有多少种组成方法?  
<https://www.zhihu.com/question/315108379>
  - 如何用最少的次数测出鸡蛋会在哪一层摔碎?  
<https://www.zhihu.com/question/19690210>
  - [LeetCode] Maximum Product Subarray 求连续子数组的最大乘积  
[https://blog.csdn.net/xblog\\_/article/details/72872263](https://blog.csdn.net/xblog_/article/details/72872263)
2. 排序算法之桶排序  
<https://blog.csdn.net/developer1024/article/details/79770240>
3. 找出数组中 N 个出现 1（或奇数次）次的数字  
<https://www.jianshu.com/p/e1331664c8cf>

## 4. 均匀分布生成其他分布的方法

<https://blog.csdn.net/haolexiao/article/details/60511164>

## 5. 海量数据处理

- 面试题集锦

[https://blog.csdn.net/v\\_july\\_v/article/details/6685962](https://blog.csdn.net/v_july_v/article/details/6685962)

- 大文件中返回频数最高的 100 个词

[https://blog.csdn.net/tiankong\\_/article/details/77240283](https://blog.csdn.net/tiankong_/article/details/77240283)

## 6. 链表

- 求有环单链表中的环长、环起点、链表长

<https://www.cnblogs.com/xudong-bupt/p/3667729.html>

- 判断两个链表是否相交并找出交点

<https://blog.csdn.net/jiary5201314/article/details/50990349>

- 单链表  $O(1)$  时间删除给定节点

[https://blog.csdn.net/qq\\_35546040/article/details/80341136](https://blog.csdn.net/qq_35546040/article/details/80341136)

7. 全排列的非递归和递归实现（含重复元素）。Hint：在交换第  $i$  个元素与第  $j$  个元素之前，要求数组的  $[i, j)$  区间中的元素没有与第  $j$  个元素重复。

[https://blog.csdn.net/so\\_geili/article/details/71078945](https://blog.csdn.net/so_geili/article/details/71078945)

8. 排列组合： $k$  个球放入  $m$  个盒子

[https://blog.csdn.net/qwb492859377/article/details/50654627?tdsourcetag=s\\_pctim\\_aiomsg](https://blog.csdn.net/qwb492859377/article/details/50654627?tdsourcetag=s_pctim_aiomsg)

## 9. Next Permutation 下一个排列

<https://www.cnblogs.com/grandyang/p/4428207.html>

*Show/Hide Code*

```
1 // 从后往前先找到第一个开始下降的数字  $x$  (下标  $i$ ), 再从后往前找到第一个比  $x$  大的数  $y$  (下标  $j$ );  
   交换  $x, y$ ; 翻转区间  $[i+1, end)$ 。  
2 class Solution  
3 {  
4 public:  
5     void nextPermutation(vector<int> &num)  
6     {  
7         int i, j, n = num.size();  
8         for (i = n - 2; i >= 0; --i)  
9         {  
10            if (num[i + 1] > num[i])
```

(continues on next page)

(continued from previous page)

```

11         {
12             for (j = n - 1; j > i; --j)
13             {
14                 if (num[j] > num[i]) break;
15             }
16             swap(num[i], num[j]);
17             reverse(num.begin() + i + 1, num.end());
18             return;
19         }
20     }
21     reverse(num.begin(), num.end()); // 当前排列是最大的排列，则翻转为最小的排列
22 }
23 };

```

## 10. LeetCode 75. Sort Colors (三颜色排序 → K 颜色排序)

<https://blog.csdn.net/princexiaofeng/article/details/79645511>

11. 找到数组第  $k$  大的数

<https://leetcode.com/problems/kth-largest-element-in-an-array/>

*Show/Hide Code*

```

1  class Solution
2  {
3  public:
4      int partition(vector<int>& nums, int i, int j)
5      {
6          int pivot = nums[i];
7          int l = i+1;
8          int r = j;
9          while(true)
10         {
11             while(l<=j && nums[l]<pivot) l++;
12             while(r>i && nums[r]>pivot) r--;
13             if(l>=r) break;
14             swap(nums[l], nums[r]);
15             l++;
16             r--;
17         }
18         swap(nums[i], nums[r]);
19         return r;
20     }
21     // partition 可用如下更简洁的形式

```

(continues on next page)

(continued from previous page)

```

22     int partition(vector<int>& nums, int i, int j)
23     {
24         int pivot = nums[i];
25         int l = i;
26         int r = j+1;
27         while(true)
28         {
29             while(nums[++l]<pivot && l<j);
30             while(nums[--r]>pivot);
31             if(l>=r) break;
32             swap(nums[l], nums[r]);
33         }
34         swap(nums[i], nums[r]);
35         return r;
36     }
37
38
39     int quicksort(vector<int>& nums, int a, int b, int k)
40     {
41         int p = partition(nums, a, b);
42         if(b - p + 1 == k) return p;
43         if(b - p + 1 < k) return quicksort(nums, a, p-1, k - (b - p + 1));
44         else return quicksort(nums, p+1, b, k);
45     }
46     int findKthLargest(vector<int>& nums, int k)
47     {
48         int k_id = quicksort(nums, 0, nums.size()-1, k);
49         return nums[k_id];
50     }
51 };

```

## 12. [LeetCode] Best Time to Buy and Sell Stock 买卖股票的最佳时间

- 最多一次交易

<http://www.cnblogs.com/grandyang/p/4280131.html>

- 无限次交易

<http://www.cnblogs.com/grandyang/p/4280803.html>

- 最多两次交易

<http://www.cnblogs.com/grandyang/p/4281975.html>

- 最多 k 次交易

<http://www.cnblogs.com/grandyang/p/4295761.html>

<https://blog.csdn.net/linhuanmars/article/details/23236995>

- 交易冷却

<https://www.cnblogs.com/grandyang/p/4997417.html>

13. [LeetCode] Partition Equal Subset Sum 数组分成两个子集，和相等

<https://leetcode.com/problems/partition-equal-subset-sum/>

*Show/Hide Code*

```

1 class Solution(object):
2     def backtrack(self, nums, sum_nums, sum_current, i): ## self
3         if sum_current == sum_nums/2:
4             return True
5         if i == len(nums):
6             return False
7         if self.backtrack(nums, sum_nums, sum_current+nums[i],i+1): ## self
8             return True
9         if self.backtrack(nums, sum_nums, sum_current, i+1): ## self
10            return True
11        return False
12
13    def canPartition(self, nums):
14        """
15        :type nums: List[int]
16        :rtype: bool
17        """
18        if len(nums) <= 1:
19            return False
20        sum_nums = sum(nums)
21        if sum_nums % 2:
22            return False
23        return self.backtrack(nums, sum_nums, 0, 0) ## self

```

14. [LeetCode] Find All Anagrams in a String 统计变位词出现的位置。Hint：采用滑动窗口和 计数器进行比较。

<https://leetcode.com/problems/find-all-anagrams-in-a-string/>

*Show/Hide Code*

```

1  /* https://leetcode.com/problems/find-all-anagrams-in-a-string/discuss/92027/C%2B%2B-
   ↪ O(n)-sliding-window-concise-solution-with-explanation */
2
3  class Solution
4  {

```

(continues on next page)

(continued from previous page)

```

5 public:
6     vector<int> findAnagrams(string s, string p)
7     {
8         vector<int> vec;
9         if(s.size()<p.size() || (s.empty() && p.empty())) return vec;
10        vector<int> p_counter(26, 0), s_counter(26, 0);
11        for(int i = 0; i < p.size(); ++i)
12        {
13            ++ p_counter[p[i]-'a'];
14            ++ s_counter[s[i]-'a'];
15        }
16        if(p_counter == s_counter) vec.push_back(0);
17        for(int i = p.size(); i < s.size(); ++i)
18        {
19            -- s_counter[s[i-p.size()]-'a'];
20            ++ s_counter[s[i]-'a'];
21            if(s_counter == p_counter) vec.push_back(i-p.size()+1);
22        }
23        return vec;
24    }
25 };

```

15. [LeetCode] Find the Duplicate Number 寻找重复数。数值范围为  $\{1, 2, 3, \dots, n\}$ 。Hint: 把数组元素的值当做下标, 由于元素存在重复, 因此必然会 **重复多次访问同一个位置**。从另一个角度讲, 访问序列中存在“环”。哈希不满足空间复杂度为  $\mathcal{O}(1)$  的要求。

- 找到一个重复数字。

<http://www.cnblogs.com/grandyang/p/4843654.html>

*Show/Hide Code*

```

1 // 解法一: 快慢指针, 寻找某个“环”的入口
2 class Solution
3 {
4 public:
5     int findDuplicate(vector<int>& nums)
6     {
7         int slow = 0, fast = 0, t = 0;
8         while (true)
9         {
10            slow = nums[slow];
11            fast = nums[nums[fast]];
12            if (slow == fast) break;
13        }

```

(continues on next page)

(continued from previous page)

```

14     while (true)
15     {
16         slow = nums[slow];
17         t = nums[t];
18         if (slow == t) break;
19     }
20     return slow;
21 }
22 };
23
24 // 解法二：不断交换位置，找到第一个重复访问的元素
25 class Solution
26 {
27 public:
28     int findDuplicate(vector<int>& nums)
29     {
30         int duplicate;
31         for(int k = 0; k < nums.size(); ++k)
32         {
33             while(nums[k]-1 != k)
34             {
35                 if(nums[k] == nums[nums[k]-1])
36                 {
37                     duplicate = nums[k];
38                     break;
39                 }
40                 swap(nums[k], nums[nums[k]-1]);
41                 // 一次交换之后，下标为 nums[k]-1 的元素就等于 nums[k] 了。
42             }
43         }
44         return duplicate;
45     }
46 };

```

- 找到所有重复数字。

<http://www.cnblogs.com/grandyang/p/6209746.html>

Show/Hide Code

```

1 // 解法一：将访问过的元素置为相反数（负数），如果下次访问到一个负数，说明这个元素被重复访问
2 class Solution
3 {
4 public:
5     vector<int> findDuplicates(vector<int>& nums)

```

(continues on next page)

(continued from previous page)

```

6      {
7          vector<int> res;
8          for (int i = 0; i < nums.size(); ++i)
9          {
10             int idx = abs(nums[i]) - 1;
11             if (nums[idx] < 0) res.push_back(idx + 1);
12             else nums[idx] = -nums[idx];
13         }
14         return res;
15         // 这种方法得到的 res 可能多次包含同一个元素, 可以使用 set
16     }
17 };
18
19 // 解法二: 不断交换位置使得 i == nums[i]-1
20 class Solution
21 {
22 public:
23     vector<int> findDisappearedNumbers(vector<int>& nums)
24     {
25         vector<int> disappear;
26         if(nums.size()<=1) return disappear;
27         for(int k = 0; k < nums.size(); ++k)
28         {
29             while(nums[k] != nums[nums[k]-1]) swap(nums[k], nums[nums[k]-1]);
30         }
31         for(int k = 0; k < nums.size(); ++k)
32         {
33             if(nums[k]-1 != k) disappear.push_back(nums[k]);
34         }
35         return disappear;
36     }
37 };

```

## 16. [LeetCode] Spiral Matrix 环形打印矩阵

<https://leetcode.com/problems/spiral-matrix/>

*Show/Hide Code*

```

1  class Solution
2  {
3  public:
4      void tranverseMatrixAccorindTo4Directions(vector<vector<int>> &matrix, const_↵
↵ unsigned int row, const unsigned int col, int start, vector<int>& vec)
5      {

```

(continues on next page)



(continued from previous page)

```

6      // 特别注意
7      // 如果把 start, endX, endY, k 声明为 unsigned int 类型, 在减到 0 的时候可能会死循
    环, 因为 unsigned int 类型不会小于 0。
8
9      int endX = row-1 - start;
10     int endY = col-1 - start;
11
12     // 1 向右
13     for(int k = start; k <= endY; ++k) vec.push_back(matrix[start][k]);
14
15     // 2 向下
16     for(int k = start+1; k <= endX; ++k) vec.push_back(matrix[k][endY]);
17
18     // 3 向左: 要求至少存在两行 (不加判断会重复扫描同一行)
19     if(endX > start) for(int k = endY-1; k >= start; --k) vec.push_
    ↪back(matrix[endX][k]);
20
21     // 4 向上: 要求至少存在两列 (不加判断会重复扫描同一列)
22     if(endY > start) for(int k = endX-1; k > start; --k) vec.push_
    ↪back(matrix[k][start]);
23
24 }
25 vector<int> spiralOrder(vector<vector<int>>& matrix)
26 {
27     vector<int> vec;
28     unsigned int row = matrix.size();
29     if(row == 0) return vec;
30     unsigned int col = matrix[0].size();
31     if(col == 0) return vec;
32     int start = 0;
33     // 循环中止条件: 圈数判断 ( (start, start) 是每一圈的入口坐标)
34     while(start*2 < row && start*2 < col)
35     {
36         tranverseMatrixAccorindTo4Directions(matrix, row, col, start, vec);
37         ++ start;
38     }
39     return vec;
40 }
41 };

```

17. [LeetCode] Longest Consecutive Sequence 最长连续序列。Hint: 方法一, 排序; 方法二, 对于每个元素  $n$ , 搜索  $n+1$  是否在数组中, 使用 hash (set) 可以获得  $\mathcal{O}(1)$  的查找复杂度。

<https://leetcode.com/problems/longest-consecutive-sequence/>

*Show/Hide Code*

```

1 class Solution(object):
2     def longestConsecutive(self, nums):
3         """
4         :type nums: List[int]
5         :rtype: int
6         """
7
8         longest = 0
9         num_set = set(nums)
10
11        for num in nums:
12            if num-1 not in num_set:
13                current_long = 1
14                while num + 1 in num_set:
15                    current_long += 1
16                    num += 1
17                longest = max(longest, current_long)
18
19        num_set.clear()
20
21        return longest

```

18. 最小公约数与最大公倍数。Hint: 辗转相除法; 最大公倍数等于两数乘积除以最大公约数。

<https://www.cnblogs.com/Arvin-JIN/p/7247619.html>

19. 跳跃的蚂蚱: 从 0 点出发, 往正或负向跳跃, 第一次跳跃一个单位, 之后每次跳跃距离比上一次多一个单位, 跳跃多少次可到达坐标  $x$  处? Hint: 走  $n$  步之后能到达的坐标是一个差为 2 的等差数列 (如  $n = 3$ , 可到达  $\{-3, -1, 1, 3\}$  )。只需找到第最小的  $n$  使得

$$(1 + 2 + \dots + n) - x = \frac{n(n+1)}{2} - x$$

是非负偶数。跳到  $x$  和跳到  $-x$  的次数相同, 因此只考虑  $x$  为正的情况。

<https://www.zhihu.com/question/50790221>

*Show/Hide Code*

```

1 // 作者: Rukia
2 // 链接: https://www.zhihu.com/question/50790221/answer/125213696
3
4 int minStep(int x)
5 {
6     if (x==0) return 0;
7     if (x<0) x=-x;

```

(continues on next page)

(continued from previous page)

```

8  int n=sqrt(2*x); // 快速找到一个接近答案的 n
9  while ((n+1)*n/2-x & 1 || (n+1)*n/2 < x) // & 的优先级低
10         ++n;
11  return n;
12 }

```

20. 求  $n$  的阶乘末尾有多少个 0。Hint: 1 个 5 和 1 个 2 搭配可以得到 1 个 0；2 的个数比 5 多，因此只关心 5 的个数；25 包含 2 个 5，125 包含 3 个 5 …。

*Show/Hide Code*

```

1  class Solution
2  {
3  public:
4      int trailingZeroes(int n)
5      {
6          if(n <= 0) return 0;
7          int res = 0;
8          while(n)
9          {
10             res += n / 5;
11             n /= 5;
12          }
13          return res;
14      }
15 };

```

21. 求一个整数的二进制表示中 1 的个数。Hint: 移位操作，负数可能造成死循环。注：指定移位次数大于或等于对象类型的比特数（如 int 型的 32 位），结果是未定义的。例如： $n \gg 32$  是未定义的，但是允许  $n \gg= 1$  执行无限次，这是安全的。

*Show/Hide Code*

```

1  // 方法一：不断右移 n。如果 n 是负数，需要保持最高位为 1，不断移位后这个数字会变成 1
   // 而陷入死循环。
2  int NumberOf1(int n)
3  {
4      int cnt = 0;
5      while(n)
6      {
7          if(n & 1) cnt ++;
8          n >>= 1;
9      }
10     return cnt;

```

(continues on next page)

(continued from previous page)

```
11 }
12
13 // 方法二：n 不动，左移一个比较子。
14 int Numberof1(int n)
15 {
16     int cnt = 0;
17     unsigned int flag = 1;
18     while(flag) // 连续左移 32 次之后为 0
19     {
20         if(n & flag) cnt ++;
21         flag <<= 1;
22     }
23     return cnt;
24 }
25
26 // 方法三：把一个整数减 1，再和原整数做逻辑与运算，会把该整数最右边的一个 1 变成 0。
27 int Numberof1(int n)
28 {
29     int cnt = 0;
30     while(n)
31     {
32         cnt ++;
33         n = (n - 1) & n;
34     }
35     return cnt;
36 }
```

### 10.1.3 C++

#### 1. 虚函数

[https://blog.csdn.net/fighting\\_coder/article/details/77187151](https://blog.csdn.net/fighting_coder/article/details/77187151)

#### 2. C++ 构造函数和析构函数能否声明为虚函数? (转载)

<https://www.cnblogs.com/hxb316/p/3853544.html>

#### 3. 重载、重写（覆盖）和隐藏的区别

<https://blog.csdn.net/zx3517288/article/details/48976097>

#### 4. C++ STL 中 vector 内存用尽后，为啥每次是两倍的增长，而不是 3 倍或其他数值?

<https://www.zhihu.com/question/36538542>

### 10.1.4 Python

1. 基本数据类型

<https://www.cnblogs.com/littlefivebolg/p/8982889.html>

2. Python 中的 None

<https://www.cnblogs.com/changbaishan/p/8084863.html>

3. 使用 lambda 高效操作列表的教程

<https://www.cnblogs.com/mxp-neu/articles/5316557.html>

4. 经典 7 大 Python 面试题

[https://blog.csdn.net/qq\\_41597912/article/details/81459804](https://blog.csdn.net/qq_41597912/article/details/81459804)

5. 迭代器和生成器

<https://www.cnblogs.com/chongdongxiaoyu/p/9054847.html>

### 10.1.5 机器学习（深度学习）

1. 激活函数

[https://fongyq.github.io/blog/deepLearning/02\\_activationFunction.html](https://fongyq.github.io/blog/deepLearning/02_activationFunction.html)

2. Batch Normalization

[https://fongyq.github.io/blog/deepLearning/03\\_batchnorm.html](https://fongyq.github.io/blog/deepLearning/03_batchnorm.html)

3. 过拟合

[https://fongyq.github.io/blog/deepLearning/03\\_batchnorm.html](https://fongyq.github.io/blog/deepLearning/03_batchnorm.html)

4. 正则化项 L1 和 L2 的区别

<https://www.cnblogs.com/lyr2015/p/8718104.html>

5. KMeans 秘籍之如何确定 K 值

<https://blog.csdn.net/aliceImx/article/details/80991870>

6. 决策树

- ID3、C4.5

<https://www.cnblogs.com/coder2012/p/4508602.html>

- 预剪枝与后剪枝

<https://blog.csdn.net/zfan520/article/details/82454814>

- CART 分类与回归树

<https://www.jianshu.com/p/b90a9ce05b28>

7. Logistic Regression

[https://fongyq.github.io/blog/machineLearning/01\\_lr.html](https://fongyq.github.io/blog/machineLearning/01_lr.html)

8. Support Vector Machine

[https://fongyq.github.io/blog/machineLearning/02\\_svm.html](https://fongyq.github.io/blog/machineLearning/02_svm.html)

9. PCA

[https://fongyq.github.io/blog/machineLearning/03\\_pca.html](https://fongyq.github.io/blog/machineLearning/03_pca.html)

## 10.1.6 论文相关

1. AlexNet/VGG/GoogleNet

<https://blog.csdn.net/gdymind/article/details/83042729>

2. CNN 卷积神经网络 \_ GoogLeNet 之 Inception(V1-V4)

<https://www.cnblogs.com/haiyang21/p/7243200.html>

3. R-CNN 系列

- 【RCNN 系列】【超详细解析】

[https://blog.csdn.net/amor\\_tila/article/details/78809791](https://blog.csdn.net/amor_tila/article/details/78809791)

- 实例分割模型 Mask R-CNN 详解：从 R-CNN，Fast R-CNN，Faster R-CNN 再到 Mask R-CNN

<https://blog.csdn.net/jiongnima/article/details/79094159>

- RCNN (三)：Fast R-CNN

<https://blog.csdn.net/u011587569/article/details/52151871>

4. CapsuleNet 解读

<https://blog.csdn.net/u013010889/article/details/78722140/>

## 10.1.7 其他

1. 理解数据库的事务，ACID，CAP 和一致性

<https://www.jianshu.com/p/2c30d1fe5c4e>

## 10.2 rst 语法

makefile 规则：

```
target ... : prerequisites ...
    command
    ...
    ...
```

下面是几个定义：

**target** 可以是一个 object file（目标文件），也可以是一个执行文件，还可以是一个标签（label）。对于标签这种特性，在后续的“伪目标”章节中会有叙述。

**prerequisites** 生成该 target 所依赖的文件和/或 target

**command** 该 target 要执行的命令（任意的 shell 命令）

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。说白了就是说：

prerequisites 中如果有一个以上的文件比 target 文件要新的话，command 所定义的命令就会被执行。

这就是 makefile 的规则，也就是 makefile 中最核心的内容。

```
echo "Hello World!";
```

行内公式使用 math 这个 role:  $a^2 + b^2 = c^2$ .

$$\begin{aligned}(a + b)^2 &= (a + b)(a + b) \\ &= a^2 + 2ab + b^2\end{aligned}$$

latex math 测试:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \quad k = 0, \dots, N-1.$$

将高亮语言设置为 C

测试 C

```
1 int a = 0;
2 char c = 'c';
3 printf("%c\n", c);
```

这里是 C++：

```
1 int main()
2 {
3     int i;
4     int j;
5     cin >> i >> j;
6     cout << i << j << endl;
```

(continues on next page)

(continued from previous page)

```

7     return 1;
8 }
9 // 主函数注释

```

斜体 *text*

将高亮语言设置为 python

测试 python

```

1 import torch
2 import numpy as np
3 print "hello world"

```

这里是 python (code):

```

1 def foo():
2     print "Love Python, Love FreeDome"
3     print "E 文标点,.0123456789, 中文标点,. "

```

如果数据库有问题, 执行下面的 SQL:

```

-- Dumping data for table `item_table`
INSERT INTO item_table VALUES (
0000000001, 0, 'Manual', '', '0.18.0',
'This is the manual for Mantis version 0.18.0.\r\n\r\nThe Mantis manual is modeled after the
↳[url=http://www.php.net/manual/en/]PHP Manual[/url]. It is authored via the "\\"manual\\" module
↳in Mantis CVS. You can always view/download the latest version of this manual from [url=http://
↳mantisbt.sourceforge.net/manual/]here[/url].',
'', 1, 1, 20030811192655);

```

下面是 python:

```

1 # 测试注释
2 def foo():
3     print "Love Python, Love FreeDome"
4     print "E 文标点,.0123456789, 中文标点,. "

```

下面是 javascript 的 rst 源码:

```

1 .. code-block:: javascript
2     :linenos:
3
4     function whatever()
5     {

```

(continues on next page)



(continued from previous page)

```

6     return "such color"
7 }

```

下面是 bash :

```

1 cd home
2 echo $PATH
3 source ~/.bashrc
4 ls -l
5 mkdir filefolder
6 cd ..

```

下面是 python (code-block):

```

1 class Solution(object):
2     def jump_from_i(self, nums, i):
3         if i == len(nums) - 1:
4             return True
5         max_step = min(len(nums), i + nums[i] + 1)
6         for t in range(i+1, max_step):
7             if self.jump_from_i(nums, t):
8                 return True
9         return False
10    def canJump(self, nums):
11        """
12        https://leetcode.com/problems/jump-game/
13        Each element in the array represents your maximum jump length at that position.
14
15        Input: [2,3,1,1,4]
16        Output: true
17        Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
18
19        :type nums: List[int]
20        :rtype: bool
21        """
22        if nums == []:
23            return False
24        if len(nums) == 1:
25            return True
26        return self.jump_from_i(nums, 0)

```

代码显示与隐藏:

*Show/Hide Code*

*Show/Hide Code*

```
1 from plone import api
2 ...
```

插入空行使用 `|` 。

上面是两个空行。

### 10.2.1 参考资料

1. reStructuredText(rst) 快速入门语法说明

<https://www.jianshu.com/p/1885d5570b37>

2. RST 语法

<https://3vshej.cn/rstSyntax/rstSyntax.html>

3. 代码隐藏（自定义，`_templates` 放在 `conf.py` 同目录下）

<http://cn.voidcc.com/question/p-pnfmhomd-v.html>

<https://stackoverflow.com/questions/2454577/sphinx-restructuredtext-show-hide-code-snippets>

4. 代码隐藏（安装扩展，全屏显示，体验不好）

<https://sphinxcontrib-contentui.readthedocs.io/en/latest/installation.html>

<https://sphinxcontrib-contentui.readthedocs.io/en/latest/toggle.html>

5. Sphinx + Github Page + Read the Docs

<https://kyzhang.me/2018/05/08/Sphinx-Readthedocs-GitHub2build-wiki/>

<https://www.jianshu.com/p/78e9e1b8553a>

[https://blog.csdn.net/baidu\\_25464429/article/details/80805237](https://blog.csdn.net/baidu_25464429/article/details/80805237)

<https://github.com/mathLab/PyGeM/issues/94>

<https://jamwheeler.com/college-productivity/how-to-write-beautiful-code-documentation/>

<https://daler.github.io/sphinxdoc-test/includeme.html>

6. latex 颜色

<http://latexcolor.com/>