Fong Documentation

Release alpha

fong

Jun 10, 2019

目录

Note: **文中可能存在错误，欢迎 PR。**

https://github.com/fongyk

https://github.com/fongyq

CHAPTER

ONE

C/C++

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：

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

```
#include <ctime>
using namespace std;

struct TreeNode
{
  int val;
  TreeNode* left;
  TreeNode* right;
  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)

```
};

int main(int argc, char ** argv)
{
  //int* p = new int(1); /* 这两行与下面三行等效 */
  //cout << *p << endl;
  int* p = new int;
  *p = 1;                /* p 已经申请了内存空间，可以直接赋值 */
  cout << *p << endl;

  TreeNode* q = new TreeNode(10);
  cout << q->val << endl;

  TreeNode node(100);
  TreeNode* r = &node;   /* r 不能 delete */
  cout << r->val << endl;

  delete p;
  delete q;

  return 0;
}
```

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。

```
#include<iostream>
using namespace std;

double square(double x)
{
  return x * x;
}
```

(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：

```
int** da = new int*[r];
for(int i = 0; i < r; ++i)
{
    da[i] = new int[c];
}
```

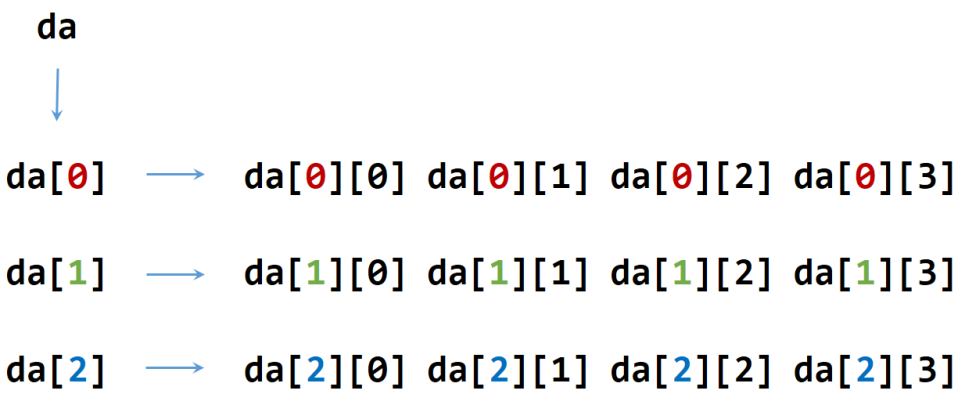
🄵存释放:

```
for(int i = 0; i < r; ++i)
{
    delete[] da[i]; // 释放指针指向的内存空间
    da[i] = nullptr; // 置为空指针，防止出现‘野指针’
}
delete[] da;
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🄵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:



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:

```
// int** f; // f 的声明
template<typename T>
void Init2DArray(T** &f, const int row, const int col)
{
    f = new T*[row];
    f[0] = new T[row * col];
    for(int i = 1; i < col; ++i)
    {
        f[i] = f[0] + col * i;
    }
}
```

内存释放方式如下:

```
template<typename T>
void Delete2DArray(T** &f)
{
    if(f != nullptr)
    {
        if(f[0] != nullptr)
        {
            delete[] f[0];
            f[0] = nullptr;
        }
        delete[] f;
        f = nullptr;
    }
}
```

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:

```
void func1(int *arr);// 函数可能会改变数组
void func2(const int *arr);// 不能改变数组

void func3(int arr[100])
{
    cout << sizeof(arr) << endl;// 4
}
// int arr[100];
// func3(arr); // 调用 func3
```

引用形式:

```
void func4(int (&arr)[100])
{
    cout << sizeof(arr) << endl;// 100
}
// int arr[100];
// 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 int (*)[L][M][N] , 指向 a 这个数组。

3. a 是 3 级指针, 类型是 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 int *ptr = (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:

```
char *strA = "abcdef";
char strB[] = "abcdef";
char strC[5] = {'a'};
char strD[3] = {'a', 'b', 'c'};
char strE[] = {'a', 'b', 'c'};
char strF[] = {'a', 'b', 'c', '\0'};
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。

```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
  int i = 200000003 / 100000002; // 1.9999999900000003

  float f_i = 200000003 / (float)100000002; // 浮点型常数默认为 const double，或用 200000003.0f 指定为 float。
  float f_f = (float)200000003.0 / (float)100000002.0; // 若不进行强制类型转换，会有 warning: truncation from 'double' to 'float'
  float f_d = (float)200000003.0 / (double)100000002;// warning: truncation from 'double' to 'float'
  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

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。

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

1.4.2 普通数据成员

遵循内存对齐原则。

```
class A
{
    int a;
    char c;
};
// sizeof(A) = 4 + 4 = 8

class B
{
    double a;
    char c;
};
// 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。

```
class A
{
    int a;
};
```

(continues on next page)

(continued from previous page)

```

class B: public A
{
    char ch;
    double b;
}
// sizeof(B) = 8 + 8 = 16 (对齐： 4+1 -> 8)

class C: public A
{
    double c;
    char ch;
}
// 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。

```
class Empty
{
};

class A: public Empty
{
    int b;
}
// 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。

```
class Empty
{
};

class A
{
    int b;
    Empty e;
```

(continues on next page)

(continued from previous page)

```
}
// 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

```
class Base
{
    virtual void f1();
    virtual void f2();
};
// sizeof(Base) = 4

class Derived: public Base
{
    virtual void f1();// 覆盖了基类中的 f1(), 多态
    virtual void f3();
};
// sizeof(Derived) = 4
```

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** 是继承的有虚函数的基类的个数。

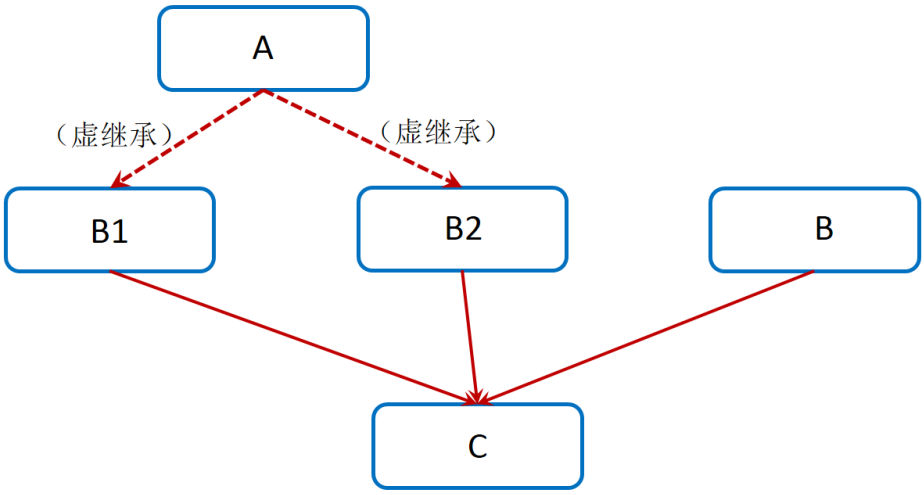
```
class A
{
};
// sizeof(A) = 1

class B
{
    char ch;
    virtual void f0();
};
// sizeof(B) = 4 + 4 = 8

class C
{
    char ch1;
    char ch2;
    virtual void f1();
    virtual void f2();
};
// sizeof(C) = 4 + 4 = 8

class D: public A, public C
{
    int d;
    virtual void f1();
    virtual void f2();
};
// sizeof(D) = 4 + 4 + 1*4 = 12

class E: public B, public C
{
    int e;
    virtual void f0();
    virtual void f1();
};
// 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，虚基类指针指向了一个 **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。

```
class A {
      int a;
};
// sizeof(A) = 4

class B :virtual public A
{
      int b;
      virtual void myfunB();
};
// 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”的整数倍偏移地址开始存储。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)

```
};
// sizeof(Data) = 1 + (3) + 4 + 1 + (3) = 12，括号内表示补齐的字节数。
// a 相对于起始位置的偏移量必须是 4 的整数倍，因此 c 后面需要补齐 3 个字节。

class Data
{
    char c;
    char d;
    int a;
};
// sizeof(Data) = 1 + 1 + (2) + 4 = 8

class BigData
{
      char array[33];
};
// sizeof(BigData) = 33

class Data
{
      BigData bd;
      //int integer; // 不管有没有注释这一行，sizeof(Data) 结果一样
      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
};
// 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

```
struct BitField
{
  unsigned int a:4;  //占用 4 个二进制位
  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
  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
  unsigned int c:4;  //占用 4 个二进制位
  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
  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
  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
```

(continues on next page)

(continued from previous page)

```
};
// 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

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

5. 位域的对齐

```
struct BFA
{
    unsigned char a:2;
    unsigned char b:3;
    unsigned char c:3;
};
// sizeof(BFA) = 1

struct BFB
{
    unsigned char a:2;
    unsigned char b:3;
    unsigned char c:3;
    unsigned int  d:4;
};
// sizeof(BFA) = 1 + (3) + 4 = 8
```

6. 一个例子

```
struct num
{
    int a:3;
    int b:2;
    int c:1;
};

int main()
```

(continues on next page)

(continued from previous page)

```
{
    struct num n = {8, -6, 5};
    cout << n.a << endl;
    cout << n.b << endl;
    cout << n.c << endl;
    return 0;
}
```

- $8 = (00001000)_2$ ，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

1. 库函数，包含在头文件 <utility> 中。

```
#include <utility>
using std::swap;
swap(a, b);
```

```
// swap algorithm example (C++11)
#include <iostream>     // std::cout
#include <utility>      // std::swap

int main ()
{
  int x=10, y=20;                   // x:10 y:20
  std::swap(x,y);                   // x:20 y:10

  int foo[4];                       // foo: ?  ?  ?  ?
  int bar[] = {10,20,30,40};        // foo: ?  ?  ?  ?    bar: 10 20 30 40
  std::swap(foo,bar);               // foo: 10 20 30 40   bar: ?  ?  ?  ?

  std::cout << "foo contains:";
  for (int i: foo) std::cout << ' ' << i;
  std::cout << '\n';

  return 0;
}
```

2. 指针。

```
templtate<class T>
void Swap(T *x, T *y)
{
    T tmp = *x;
    *x = *y;
    *y = tmp;
}
```

3. 引用。

```
templtate<class T>
void Swap(T &x, T &y)
{
    T tmp = x;
```

(continues on next page)

(continued from previous page)

```
    x = y;
    y = tmp;
}
```

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

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

template<class T>
void Swap(T& a, T& b)
{
    a = a ^ b;
    b = a ^ b;
    a = 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。

```
templtate<class T>
void Swap(T &x, T &y)
{
    x = x + y;
    y = x - y;
    x = x - y;
}
```

Note: 如果存在类型特定的 swap 版本（即为某个类定制的 swap），其匹配程度会优于 std 中定义的版本。

```
using std::swap; // 声明

void swap(Foo& a, Foo&b); // 声明

Foo a, b;

swap(a, b); // 此处匹配的是定制版本的 swap
```

1.5.1 参考资料

1. C++ reference

   http://www.cplusplus.com/reference/utility/swap

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。

```
void func1(int arr[100])
{
    cout << sizeof(arr) << endl; // 指针的大小为 4（32 位编译器）
    /*
        function body
    */
}

void func2(int *arr)
{
    cout << sizeof(arr) << endl; // 指针的大小为 4（32 位编译器）
    /*
        function body
    */
}

int a[10] = {1,2,3};
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。

```
void func1(int (&arr)[10])
{
    cout << sizeof(arr) << endl; // 大小为 4*10=40（32 位编译器）
    /*
        function body
```

(continues on next page)

(continued from previous page)

```
    */
}

int a[10] = {1,2,3};
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。

```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
  int k = 1;
  int* pk = &k;
  int& rk = k;

  cout << "&k:" << &k << endl;   // 0029FC44
  cout << "k:" << k << endl;     // 1

  cout << "&pk:" << &pk << endl; // 0029FC68
  cout << "pk:" << pk << endl;   // 0029FC44 (pk = &k)
  cout << "*pk" << *pk << endl;  // 1

  cout << "&rk:" << &rk << endl; // 0029FC44 (&rk = &k)
  cout << "rk:" << rk << endl;   // 1

  return 0;
}
```

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。

```
void rValue(const int &x)
{
    cout << x << endl;
}
```

(continues on next page)

(continued from previous page)

```
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。这么做的好处是使得程序员的意图更加清晰（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
```

(continues on next page)

(continued from previous page)

```
{
  void f(int) const; // 错误: f 禁止覆盖
  void f1(int) const override; // 正确
  void f2(int) override; // 错误: 基类中没有形如 f2(int) 的函数
  void f3() override; // 错误: f3 不是虚函数
  void f4() override; // 错误: 基类中没有名为 f4 的函数
};

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**）。

```
#include <iostream>
using namespace std;

class Base
{
public:
  virtual void f(){ cout << "base" << endl; }
};

class Derived : public Base // 注意: 这里必须为 public 继承
{
public:
  void f(){ cout << "derived" << endl; }
};

int main(int argc, char ** argv)
{
  Derived d = Derived(); // 派生类对象
  Base* pb = &d; // 基类指针
  pb->f(); // derived

  Base b = Base(); // 基类对象
  Base& rb = b; // 基类引用
  rb.f(); // base
```

(continues on next page)

(continued from previous page)

```
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 例子

虚析构函数

删除一个指向派生类对象的基类指针时，需要虚析构函数。

```
class A
{
public:
  ~A();
// virtual ~A();
};
A::~A()
{
  printf("delete A ");
}

class B : public A
{
public:
  ~B();
};
B::~B()
{
  printf("delete B ");
}
```

基类析构函数未加 virtual：

```
A *pa = new B();
delete pa;
// 输出：  delete B

B *pb = new B();
delete pb;
// 输出：  delete B delete A
```

基类析构函数加 virtual：

```
A *pa = new B();
delete pa;
// 输出：  delete B delete A

B *pb = new B();
delete pb;
// 输出：  delete B delete A
```

析构顺序

```
#include <iostream>
using namespace std;

class A
{
public:
  A() {  cout << "create A" << endl;  }

  A(A &obj) {  cout << "copy-construct A" << endl;  }

  ~A() {  cout << "~A" << endl;  }
};

class B: public A
{
public:
  B(A &a): _a(a) {  cout << "create B" << endl;  }

  ~B() {  cout << "~B" << endl;  }
private:
  A _a;
};

int main(void)
{
  A a;

  B b(a);

  cout << "-----------" << endl;

  return 0;
}
```

运行结果:

```
create A
create A
copy-construct A
create B
-----------
~B
~A
~A
~A
```

创建派生类对象时，调用构造函数的顺序如下:

- 先是父类的构造函数；(create A)
- 然后如果类成员变量中有某类（可能是父类，也可能不是）的对象，调用其相应的构造函数；(copy-construct A)
- 最后调用派生类自身的构造函数。(create B)

析构函数的调用顺序正好相反。

```
#include <iostream>
using namespace std;
class A
{
public:
  A()  {  cout<<"create A"<<endl;   }

  A(const A& other) { cout<<"copy A"<<endl;} // 拷贝构造函数

  ~A() {  cout<<"~A"<<endl;   }
};
class C
{
public:
  C()  {  cout<<"create C"<<endl;   }

  C(const A& other) { cout<<"copy C"<<endl;} // 拷贝构造函数

  ~C() {  cout<<"~C"<<endl;   }
};
class B:public A
{
public:
  B() {  cout<<"create B"<<endl;  }
```

(continues on next page)

(continued from previous page)

```
25
26    ~B() {  cout<<"~B"<<endl;  }
27  private:
28    C _c;
29  };
30
31  int main(void)
32  {
33    B b;
34    cout<<"-----------"<<endl;
35    return 0;
36  }
```

运行结果:

```
create A
create C
create B
-----------
~B
~C
~A
```

对象数组的析构

数组的多态会导致未定义的行为，不管析构函数是否声明为虚函数。所以在对数组元素执行析构时，还是要用 **派生类的指针**来 delete 。

参考：https://www.nowcoder.com/profile/3704231/myFollowings/detail/8528425。

```
1   #include <iostream>
2   using namespace std;
3
4   class A
5   {
6   public:
7     A() { cout << "A" << ends; }
8     ~A() { cout << "~A" << ends; }
9   };
10  class B:public A
11  {
12  public:
13    B() { cout << "B" << ends; }
```

(continues on next page)

(continued from previous page)

```
  ~B() { cout << "~B" << ends; }
};

int main(void)
{
  A arrA = new B[2];
  delete [] arrA;
  // 输出:  A B A B ~A ~A

  B arrB = new B[2];
  delete [] arrB;
  // 输出:  A B A B ~B ~A ~B ~A

  return 0;
}
```

1.8.5 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:

```
char* strcpy(char* dst, const char* src)
{
    char* cp = dst;
    while( *cp++ = *src++ );  /* Copy src over dst */
    return( dst );
}

char src[10] = "abcd";
char dst[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 。

```cpp
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main ()
{
  try
  {
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast.\n";

  } catch (exception& e) {cout << "Exception: " << e.what();}
  return 0;
}

// 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、volitale 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。

```cpp
// const_cast
#include <iostream>
using namespace std;

```

(continues on next page)

(continued from previous page)

```
void print (char * str)
{
  cout << str << '\n';
}

int main ()
{
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}

// 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

```
#include <iostream>
#include <sstream>
using namespace std;
```

(continues on next page)

(continued from previous page)

```
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;
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的对象，所以通过指针 **可以修改它所指的对象的值**。

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

templtate<class T>
void swap(T* x, T* y)
{
    T* tmp = x;
    x = y;
    y = tmp;
}
//  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  {
3  public:
4    void print()
5    {
6      cout << "print" << endl;
7    }
8
9    void getA()
10   {
11     cout << a << endl;
12   }
13
14   void setA(int x)
15   {
16     a = x;
```

(continues on next page)

(continued from previous page)

```
  }

  virtual test()
  {
    cout << "test" << endl;
  }

private:
  int a = 100;
};

TestNullPtr* ptr = nullptr;
ptr->print(); // 运行成功
ptr->getA(); // 编译成功，运行失败
ptr->setA(); // 编译成功，运行失败
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。

```
void func()
{
  static int a = 1; // 初次调用 func() 时才会执行初始化
  cout << a << endl;
  a ++;
}

int main()
{
  func(); // 1
  func(); // 2
  return 0;
}
```

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。

```
class Myclass
{
private:
    int a , b , c;
    static int sum;  //声明静态数据成员
public:
    Myclass(int a , int b , int c);
    void GetSum();
};

int Myclass::sum = 0;   //定义并初始化静态数据成员

Myclass::Myclass(int a , int b , int c)
{
    this->a = a;
    this->b = b;
    this->c = c;
    sum += a+b+c;
}
void Myclass::GetSum()
{
    cout<<"sum="<<sum<<endl;
}
```

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

```
#ifndef HEADER_INCLUDED // 条件编译，避免重复包含头文件
#define HEADER_INCLUDED

#ifdef __cplusplus // extern "C" 只用在 c++ 文件中
extern "C" {
#endif /* __cplusplus */

#include "c.h"

char* strcpy(char*,const char*);

/*..............................
 * do something else
 *..............................
 */

#ifdef __cplusplus
}
#endif /* __cplusplus */

#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 成员

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 变量时不能初始化。

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 成员

不能在定义对象时对静态变量进行 **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。

1.16.3 static const 成员

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

```cpp
/* header.h */
class Solution
{
public:
  static void print()
  {
    cout << var << endl; // 100
    cout << (mapping.begin()->second)[0] << endl; // 'a'
  }

private:
  static const map<char, vector<char> > mapping; // 常量声明式
  static const int var = 100; // 常量声明式（直接初始化）
};

/* source.cpp */
const map<char, vector<char> > Solution:: mapping = {{'2', {'a', 'b', 'c'}},
                                                     {'3', {'d', 'e', 'f'}},
                                                     {'4', {'g', 'h', 'i'}}}; // mapping 的定义

// 注：const map 只能通过迭代器 const_iterator 访问元素（it->first, it->second），不能通过下标 [] 的方式访问。

// 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，而无需提供以下的定义式。
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。

Warning: **static** 和 **const** 不能 **同时**修饰成员函数，只能修饰成员变量。因为常量成员函数（const）拥有一个 **this** 指针，是一个指向常量类类型的常量指针，而静态成员函数（static）没有 **this** 指针。

Note:

头文件中应该写什么？

- 变量和函数的声明，而不是定义。如：`extern int a; void f();` 是允许的，而 `int a; void f() {};` 是不允许的。

- const 全局变量可以定义在头文件中，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。
- 内联函数。
- 类的定义。成员函数定义在类的定义体内，编译器会视其为内联函数；如果定义在类的头文件内，而没有写进类定义体内，是不合法的，需要定义在别的源文件（.cpp）文件内。
- 函数模板和类模板成员函数。

把声明在头文件（header.h）中的函数或类成员函数定义在一个源文件（source1.cpp）中，需要包含该头文件（#include "header.h"），当另一个源文件（source2.cpp）需要调用上述函数时，只需要包含头文件（#include "header.h"），而不是包含函数定义的源文件（source1.cpp）。

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. 头文件重复包含和变量重复定义

   https://blog.csdn.net/u014557232/article/details/50354127

4. C++ 初始化列表

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

5. 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

6. 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}。可以定义 **无类型名**的枚举类型。

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。

```
enum Week {Sun=1, Mon, Tue, Wed, Thu, Fri=100, Sat} day_1;
// 全局变量 day_1 默认初始化为 0。

void ff(int num)
{
  cout << -10 * num << endl;
}
void ff(Week day)
{
  cout << static_cast<float>(10 * day) << endl;
}

int main(int argc, char** argv)
{
  Week day_2, day_3;

  day_1 = Sun; // 或者 day_1 = Week::Sun （不限定作用域的枚举类型）
  day_2 = day_1;
  int i = day_1; // i = 1
  int j = Mon; // j = 2

  Week day_4(Week(100)); // day_4 = Fri
```

(continues on next page)

(continued from previous page)

```
  bool equal = (day_4 == Fri); // true

  Week day_5(Week(-1)); // 越界，但是不报错，day_5 = -1 （VS 2013）

  ff(1); // 匹配 ff(int)，输出 -10
  ff(day_1); // 匹配 ff(Week)，输出 10
  ff(Fri); // 匹配 ff(Week)，输出 1000

  return 0;
}
```

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

以下函数基于 C++11 标准。

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

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char ** argv)
{
  int a[11] = {1,2,3,4,5,5,5,5,6,7,8};
  cout << lower_bound(a, a + 11, 5) - a << endl; // 4
  cout << upper_bound(a, a + 11, 5) - a << endl; // 8
  vector<int> v(a, a + 11); // 用 a 对 v 初始化
  cout << lower_bound(v.begin(), v.end(), 5) - v.begin() << endl; // 4
  cout << upper_bound(v.begin(), v.end(), 5) - v.begin() << endl; // 8

  *lower_bound(a, a + 11, 5) = 500;
  for (auto ai : a) cout << ai << ends; // 1 2 3 4 500 5 5 5 6 7 8
  cout << endl;

  *lower_bound(v.begin(), v.end(), 5) = 500;
  for (auto vi : v) cout << vi << ends; // 1 2 3 4 500 5 5 5 6 7 8
  cout << endl;

  return 0;
}
```

1.18.2 fill，fill_n，for_each

```cpp
#include <algorithm>
```

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

```cpp
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。

```cpp
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

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template<class T>
void print(T elem){ cout << elem << " "; }

int main(int argc, char ** argv)
{

  float a[4] = { 0.0 }; // {0.0, 0.0, 0.0, 0.0}
  vector<int> v(4, 0); // {0, 0, 0, 0}

  fill(a, a+4, 3.3); // {3.3, 3.3, 3.3, 3.3}
  fill_n(a, 2, 6.6); // {6.6, 6.6, 3.3, 3.3}
  fill_n(v.begin(), 4, 9); // {9, 9, 9, 9}

  for_each(a, a + 4, print<float>); //  6.6 6.6 3.3 3.3
  cout << endl;
  for_each(v.begin(), v.end(), print<int>); //  9 9 9 9
  cout << endl;

  int b[10][20];
  fill(b[0], b[0] + 200, 2); // b 所有元素为 2

  return 0;
}
```

最长上升子序列:

```
/* https://leetcode.com/problems/longest-increasing-subsequence/ */
/* O(nlogn) in time.*/

class Solution
```

(continues on next page)

(continued from previous page)

```
{
public:
  int lengthOfLIS(vector<int>& nums)
  {
    if(nums.size()<=1) return nums.size();
    int inf = INT_MAX;
    int len = nums.size();
    int* dp = new int[len];
    fill(dp, dp+len, inf);
    for(int k = 0; k < len; ++k) *lower_bound(dp, dp+len, nums[k]) = nums[k];
    int length = lower_bound(dp, dp+len, inf) - dp;
    delete[] dp;
    return length;
  }
};
```

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

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>
using namespace std;

bool comparator(int i, int j)
{
  return (i < j);
}

struct myclass
```

(continues on next page)

(continued from previous page)

```
{
  bool operator() (int i, int j)
  {
    return (i < j);
  }
} myobject;

int main(int argc, char ** argv)
{

  int a[] = { 32, 71, 12, 45, 26, 80, 53, 33 };
  vector<int> v(a, a + 8);               // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  sort(v.begin(), v.begin() + 4);           //(12 32 45 71)26 80 53 33

  // using comparator as comp
  sort(v.begin() + 4, v.end(), comparator); // 12 32 45 71(26 33 53 80)

  // using object as comp
  sort(v.begin(), v.end(), myobject);     //(12 26 32 33 45 53 71 80)

  // using build-in comp: greater
  sort(v.begin(), v.end(), greater<int>()); // (80 71 53 45 33 32 26 12)

  // using build-in comp: less
  sort(v.begin(), v.end(), less<int>());  //(12 26 32 33 45 53 71 80)

  // using reverse_iterator
  sort(v.rbegin(), v.rend());  // (80 71 53 45 33 32 26 12)

  // sort array
  sort(a, a + 8, greater<int>());  // (80 71 53 45 33 32 26 12)
  sort(a, a + 8);                 // (12 26 32 33 45 53 71 80)，可使用 comparator、myobject、less
↪<int>()

  return 0;
}
```

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 reverse

```
#include <algorithm>

template <class BidirectionalIterator>
void reverse (BidirectionalIterator first, BidirectionalIterator last);
```

翻转区间 [first,last) 内的元素。适用于 vector、string 以及静态数组、动态数组等。

Example

```
#include <iostream>     // std::cout
#include <algorithm>    // std::reverse
#include <numeric>      // std::iota
#include <vector>       // std::vector

int main ()
{
  std::vector<int> myvector;

  // set some values:
  myvector.resize(9);
  std::iota(myvector.begin(), myvector.end(), 1);   // 1 2 3 4 5 6 7 8 9

  std::reverse(myvector.begin(),myvector.end());    // 9 8 7 6 5 4 3 2 1

  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

1.18.5 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

```
#include <iostream>
#include <algorithm>
using namespace std;

int main(int argc, char ** argv)
{

  int a[10] = { 1, 2, 3, 4, 5, 6 };
  cout << a[9] << endl; // 0

  cout << *min_element(a, a + 10) << endl; // 0

  cout << *max_element(a, a + 10) << endl; // 6

  auto p = minmax_element(a, a + 10);

  cout << *p.first << ends << *p.second << endl; // 0 6

```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

1.18.6 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

```
#include <iostream>     // std::cout
#include <functional>   // std::minus
#include <numeric>      // std::accumulate

int myfunction (int x, int y) {return x+2*y;}

struct myclass
{
  int operator()(int x, int y) {return x+3*y;}
} myobject;

int main ()
{
  int init = 100;
  int numbers[] = {10,20,30};

  std::cout << "using default accumulate: ";
  std::cout << std::accumulate(numbers,numbers+3,init); // 100 + (10 + 20 + 30)
  std::cout << '\n';

```

(continues on next page)

(continued from previous page)

```
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  }
```

1.18.7 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);

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

Example

```cpp
#include <iostream>     // std::cout
#include <functional>   // std::multiplies
#include <numeric>      // std::partial_sum
#include <vector>

int myop (int x, int y) {return x+y+1;}

int main ()
{
  int val[] = {1,2,3,4,5};
  int result[5];

  std::partial_sum (val, val+5, result);
  std::cout << "using default partial_sum: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' '; // 1 3 6 10 15
  std::cout << '\n';

  std::vector<int> result_vec(6, 0); // 0 0 0 0 0 0
  std::partial_sum (val, val+5, result_vec.begin()); // 1 3 6 10 15 0

  std::partial_sum (val, val+5, result, std::multiplies<int>()); // 1 2 6 24 120
  std::cout << "using functional operation multiplies: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' ';
  std::cout << '\n';

  std::partial_sum (val, val+5, result, myop); // 1 4 8 13 19
  std::cout << "using custom function: ";
  for (int i=0; i<5; i++) std::cout << result[i] << ' ';
  std::cout << '\n';
  return 0;
}
```

1.18.8 iota

```cpp
#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

```
#include <iostream>     // std::cout
#include <numeric>      // std::iota

int main () {
  float numbers[10];

  std::iota (numbers,numbers+10,3.5);

  std::cout << "numbers:";
  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
  std::cout << '\n';

  return 0;
}
```

1.18.9 inner_product

```
#include <numeric>

// sum/multiply
emplate <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,
                 BinaryOperation1 binary_op1,
                 BinaryOperation2 binary_op2);
```

内积运算，再与 **init** 做运算:

```
while (first1!=last1)
{
```

(continues on next page)

(continued from previous page)

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

Example

```
#include <iostream>     // std::cout
#include <functional>   // std::minus, std::divides
#include <numeric>      // std::inner_product

int myaccumulator (int x, int y) {return x-y;}
int myproduct (int x, int y) {return x+y;}

int main ()
{
  int init = 100;
  int series1[] = {10,20,30};
  int series2[] = {1,2,3};

  std::cout << "using default inner_product: ";
  std::cout << std::inner_product(series1,series1+3,series2,init); // 100 + (10*1 + 20*2 + 30*3)
  std::cout << '\n';

  std::cout << "using functional operations: ";
  std::cout << std::inner_product(series1,series1+3,series2,init,
                                  std::minus<int>(),std::divides<int>()); // 100 - (10/1 + 20/2 +␣
↪30/3)
  std::cout << '\n';

  std::cout << "using custom functions: ";
  std::cout << std::inner_product(series1,series1+3,series2,init,
                                  myaccumulator,myproduct); // 100 - (10+1 + 20+2 + 30+3)
  std::cout << '\n';

  return 0;
}
```

1.18.10 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

```
#include <iostream>
#include <cstring>

int main()
{
  char str[] = "almost every programmer should know memset!";
  memset (str,'-',6);
  cout << str << endl; // ------ every programmer should know memset!

  int a[10][10];
  memset(a, -1, sizeof(a)); // 或者 10*10*sizeof(int)，全部赋值为-1
  for(int e:a) cout << bitset<sizeof(int)*8>(e) << endl; // 11111111  11111111  11111111  11111111
(补码)

  int b[5];
  memset(b, 1, sizeof(b));// 或者 5*sizeof(int)，全部赋值为 16843009
  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)

  return 0;
}
```

1.18.11 附：头文件

- cmath
  - pow()
  - sqrt()
  - floor()

  - ceil()
  - round()
- `cstdlib`
  - abs()
  - fabs()
- `limits`
  - INT_MIN: `(signed int)0x80000000`
  - INT_MAX: `0x7fffffff`
- `algorithm`
  - min()
  - max()
- `utility`
  - pair
  - move
- `functional`
  - less< *TYPE* >()
  - greater< *TYPE* >()
- `cassert`
  - assert()

1.18.12 参考资料

1. C++ reference

   http://www.cplusplus.com/reference/algorithm/lower_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/sort

   http://www.cplusplus.com/reference/algorithm/reverse

   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/iota

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

1.19 常用 STL 类及容器

STL: Standard Template Library，标准模板库。

以下基于 C++11 标准。

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

一个容器适配器接受一种已有的容器类型，使其行为看起来像一种不同的类型。默认情况下，stack 和 queue 基于 deque 实现，priority_queue 基于 vector 实现。

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：capcity()
- 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)，emplace_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);

template <class... Args>
iterator emplace (const_iterator position, Args&&... args);
```

- 尾部删除：pop_back()
- 申请空间：至少能容纳 n 个元素（capcity() 为 n）。

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

- 改变大小：将元素个数变为 n。如果指定 val 且 n 大于原来的 size，则使用 val 填充新元素，原来的元素不变；如果 n 小于原来的 size，则丢弃尾部元素。

```
void resize (size_type n);
void resize (size_type n, const value_type& val);
```

- 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.capcity() 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 capcity() 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)，emplace_front(x)，emplace_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);

template <class... Args>
iterator emplace (const_iterator position, Args&&... args);
```

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。
- 关系运算：支持 ==，!=，<，<=，>，>=，从而可以直接排序

```
// 如果 first 相等，则比较 second
template <class T1, class T2>
bool operator<  (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs)
{ return lhs.first<rhs.first || (!(rhs.first<lhs.first) && lhs.second<rhs.second); }

template <class T1, class T2>
bool operator>  (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs)
{ return rhs<lhs; }
```

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

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  // first insert function version (single parameter):
  mymap.insert ( std::pair<char,int>('a',100) );
```

(continues on next page)

(continued from previous page)

```
  mymap.insert ( std::map<char,int>::value_type('z',200) );

  std::pair<std::map<char,int>::iterator,bool> ret;
  ret = mymap.insert ( std::pair<char,int>('z',500) );
  if (ret.second==false)
  {
    std::cout << "element 'z' already existed";
    std::cout << " with a value of " << ret.first->second << '\n';
  }

  return 0;
}
```

1.19.7 stack

```
#include<stack>
```

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

```
void pop();
```

1.19.9 priority_queue

```
#include<queue>
```

实现 priority_queue 的底层容器默认是 vector，同时默认功能是大顶堆（值越大，优先级越高；队首元素值最大）。

```
template <class T, class Container = vector<T>,
class Compare = less<typename Container::value_type> > class priority_queue;
```

- 大小：size()，empty()
- 最高优先级元素：top()
- 入队：push(x)，emplace(x)
- 最高优先级元素出队：pop()

```
#include <iostream>
#include <queue>
using namespace std;

template<class T>
class comparator
{
public:           // 必须是 public
  bool operator()(T a, T b)
  {
    return a > b; // 相当于 greater<T>, 小顶堆
  }
};

int main(int argc, char ** argv)
{
  priority_queue<string, vector<string>, comparator<string> > mypq;

  mypq.emplace("orange");
  mypq.emplace("strawberry");
  mypq.emplace("apple");
  mypq.emplace("pear");

  cout << "Popping out elements...";
  while (!mypq.empty())
  {
    cout << ' ' << mypq.top();
    mypq.pop();
```

(continues on next page)

(continued from previous page)

```
  }
  cout << '\n';
  return 0;
}

// 输出结果
// Popping out elements... apple orange pear strawberry
```

Note: C++11 标准引入了 **emplace_front**，**emplace**，**emplace_back** 这些操作构造而不是拷贝元素，分别对应 **push_front**，**insert/push**，**push_back**。

调用 push 或 insert 时，先创建一个元素类型的 **临时对象**，这个对象被 **拷贝**到容器中。

调用 emplace 时，将 **参数**传递给元素类型的 **构造函数**，在容器管理的内存空间中使用这些参数直接构造元素。传递给 emplace 的参数必须与构造函数匹配。

1.19.10 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);
```

1.19.11 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 );
```

1.19.12 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>
- <cwchar>

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。

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

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 = 2^32 - 1 或 2^64 - 1，而不是预想的 -1，陷入死循环。

1.19.13 参考资料

1. C++ reference

   http://www.cplusplus.com/reference/string/string

   http://www.cplusplus.com/reference/string/to_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/utility/pair/operators

   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/queue/priority_queue

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

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 AspectRation = 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，AspectRation 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。

还可以在类内声明 static const 成员。

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

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。

```
class Player
{
private:
  enum {NumTurns = 5}; // NumTurns 成为数值 5 的一个记号。
```

(continues on next page)

(continued from previous page)

```
  int scores[NumTurns]; // 使用该常量
}
```

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。

```
#include <iostream>
using namespace std;

#define CALL_WITH_MAX(a, b) f((a)>(b)? (a):(b))

#define MAX_COMP_1(a, b) (a)>(b)? (a):(b)

#define MAX_COMP_2(a, b) ((a)>(b)? (a):(b)) // 有外层括号

template<class T>
void f(T elem)
{
  cout << "max out: " <<  elem << endl;
}

template<class T>
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
```

(continues on next page)

(continued from previous page)

```
{
  f(a > b ? a : b);
}

int main(int argc, char ** argv)
{
  int a = 5, b = 0;
  CALL_WITH_MAX(++a, b); // a 自增 2 次，变为 7（++a > b => ++a）
  cout << a << endl;
  CALL_WITH_MAX(++a, b+10); // a 自增 1 次，变为 8（++a < b+10 => b）
  cout << a << endl;

  f(-10 + MAX_COMP_1(a, b)); // -10 + a > b ? a : b; 结果为 0
  f(-10 + MAX_COMP_2(a, b));// -10 + (a > b ? a : b); 结果为 -10 + 8 = -2

  CallWithMax(a, b); // 8

  return 0;
}
```

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

Continued on next page

Table 6 – continued from previous page

priority	operator	meaning
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。

```
int i = 0;
int *const p1 = &i; // 顶层 const，不能改变 p1 的值

const int ci = 42; // 顶层 const，不能改变 ci 的值
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。
const int *const p3 = p2;  // 顶层 const + 底层 const
```

(continues on next page)

(continued from previous page)

```

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。

```
// 承接上例的定义

i = ci; // 正确
p2 = p3; // 正确

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。
p2 = &i; // 正确，int* 能转换为 const int*

int &r1 = ci; // 错误，r1 没有底层 const
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，也可传入 const int 对象（传值调用，函数拷贝了实参）。

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
{
```

(continues on next page)

(continued from previous page)

```
  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。

1.23 拷贝控制

拷贝控制（copy control）

- 拷贝构造函数（copy constructor）
- 拷贝赋值运算符（copy-assignment operator）
- 移动构造函数（move constructor）
- 移动赋值运算符（move-assignment operator）
- 析构函数（destructor）

1.23.1 拷贝构造函数

拷贝构造函数的第一个参数必须是引用类型。

在函数调用中，具有非引用类型的参数要进行拷贝初始化。类似地，当一个函数具有非引用类型的返回类型时，返回值会被用来初始化调用方的结果。

拷贝构造函数被用来初始化 **非引用类类型**（被初始化的是类的非引用对象）参数，如果拷贝构造函数的参数不是引用类型，为了调用拷贝构造函数，我们必须拷贝它的实参，然而拷贝实参又需要调用拷贝构造函数，如此无限循环。

如果我们没有为类定义拷贝控制函数，编译器会为我们定义一个。与合成默认构造函数不同（如果定义了其他构造函数，则需要我们再显式定义默认构造函数），即使我们定义了其他构造函数，编译器也会为我们合成一个拷贝构造函数。

1.23.2 default 和 delete

使用 `=default` 将拷贝控制成员定义为 `=default` ，显式要求编译器生成合成的版本。

- 类内使用 =default 修饰成员的声明，则合成的函数隐式地声明为内联函数（注：定义在类内的函数自动为内联函数）。
- 类外使用 =default 修饰成员的定义，则合成的成员不是内联函数。
- 只能对默认构造函数或拷贝控制成员使用 =default 。

使用 =delete 在函数参数列表之后加上 =delete 定义为 **删除的函数**：虽然有声明，但是不能以任何形式使用它们。

将拷贝构造函数和拷贝赋值运算符定义为删除的函数，从而阻止拷贝操作。

1.23.3 直接初始化和拷贝初始化

直接初始化直接调用与实参匹配的构造函数，拷贝初始化总是调用拷贝构造函数。

```
string dots(10, '.');          // 直接初始化
string s(dots);                // 直接初始化

string s2 = dots;              // 拷贝初始化
string s3 = "999-9999";        // 拷贝初始化
string s4 = string(100, '9'); // 拷贝初始化
```

```
class ClassTest
{
public:
  ClassTest()
  {
    c[0] = '\0';
    cout << "ClassTest()" << endl;
  }

  ClassTest& operator=(const ClassTest &ct)
  {
    strcpy(c, ct.c);
    cout << "ClassTest& operator=(const ClassTest &ct)" << endl;
    return *this;
  }

  ClassTest(const char *pc)
  {
    strcpy(c, pc);
    cout << "ClassTest (const char *pc)" << endl;
  }

// private:
```

(continues on next page)

(continued from previous page)

```
  ClassTest(const ClassTest& ct)
  {
    strcpy(c, ct.c);
    cout << "ClassTest(const ClassTest& ct)" << endl;
  }
private:
  char c[256];
};
```

调用:

```
ClassTest ct1("ab");          // 直接初始化
// 输出:  ClassTest (const char *pc)

ClassTest ct2 = "ab";         // 拷贝初始化
// 输出:  ClassTest (const char *pc)
// 首先调用构造函数 ClassTest(const char *pc) 函数创建一个临时对象；然后调用拷贝构造函数，把这个临时对象作为参数，构造对象 ct2
// 然而结果并没有输出 ClassTest(const ClassTest& ct)。有说法是编译器优化之后，直接匹配了ClassTest(const char *pc)，不再调用拷贝构造函数

//ClassTest ct3 = ct1;        // 拷贝初始化
// 输出:  ClassTest(const ClassTest& ct)
// ct1 已经存在，直接调用拷贝构造函数

//ClassTest ct4(ct1);         // 直接初始化
// 输出:  ClassTest(const ClassTest& ct)
// ct1 已经存在，直接调用拷贝构造函数

//ClassTest ct5 = ClassTest();// 拷贝初始化
// 输出:  ClassTest()
// 首先调用默认构造函数产生一个临时对象；然后调用拷贝构造函数，把这个临时对象作为参数，构造对象 ct5

//ct3 = ct2;                  // 赋值
// 输出:  ClassTest& operator=(const ClassTest &ct)
```

当把拷贝构造函数设置为 `private` ，ct3、ct4、ct5 的初始化都无法完成。

1.23.4 push 和 emplace

在 18 章提到了 push 和 emplace 的区别，这里用一个例子解释。

Example

```cpp
#include <iostream>
#include <utility>  // std::move

class Foo
{
public:
  Foo(std::string str) : name(str)
  {
    std::cout << "constructor" << std::endl;
  }

  Foo(const Foo& f) : name(f.name)
  {
    std::cout << "copy constructor" << std::endl;
  }

  Foo(Foo&& f) : name(std::move(f.name))
  {
    std::cout << "move constructor" << std::endl;
  }

private:
  std::string name;
};

int main(int argc, char ** argv)
{
  std::vector<Foo> v;
  int count = 10000000;
  v.reserve(count);

  {
    Foo temp("test");
    // constructor
    v.push_back(temp);// push_back(const T&), 参数是左值引用
    // copy constructor
  }

  v.clear();
  {
    Foo temp("test");
    // constructor
    v.push_back(std::move(temp));// push_back(T &&), 参数是右值引用
    // move constructor
```

(continues on next page)

(continued from previous page)

```
  }

  v.clear();
  {
    v.push_back(Foo("test"));// push_back(T &&), 参数是右值引用
    // constructor
    // move constructor
  }

  v.clear();
  {
    std::string temp = "test";
    v.push_back(temp);// push_back(T &&), 参数是右值引用
    // constructor
    // move constructor
  }

  v.clear();
  {
    std::string temp = "test";
    v.emplace_back(temp);// 只有一次构造函数，不调用拷贝构造函数，速度最快
    // constructor
  }

  return 0;
}
```

Note: 我们可以销毁一个移动之后的源对象（moved-from），也可以赋予它新值，但是不能使用一个移后源对象的值。

如：上例中的 temp 被移动后，就不能再取它的值来使用。

1.23.5 参考资料

1.《C++ Primer 第 5 版中文版》Page 440 – 442，449，470 – 475。

2. 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

3. C++11 使用 emplace_back 代替 push_back

   https://blog.csdn.net/yockie/article/details/52674366

CHAPTER

TWO

PYTHON

2.1 in-place 运算

2.1.1 += 运算

+= 是一个 in-place 运算符，看如下例子：

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

结果如下：

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

如果改变成如下形式：

```
a = []
b = a
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。

```
import operator

x = 5
y = 6
a = 5
b = 6

z = operator.add(a, b)
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。

```
import operator

a = [1,2,4,5]
b = [1,2,4,5]

z = operator.add(a, [1,2,3])
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:

```
from func import x, foo
# 假设 x 是一个变量，foo 是一个函数

__all__ = ['x', 'foo']
```

test.py 中的内容如下:

```
from myPack import *

print x

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。

```
>>> a = "hello"
>>> b = "hello"
>>> a==b
True
>>> a is b
True
>>> id(a)
140568052594368
>>> id(a)
140568052594368

>>> a = "hello world"
>>> b = "hello world"
>>> a==b
True
>>> a is b
False
>>> id(a)
140568052594752
>>> id(a)
140568052594320

>>> a = [1,2,3]
>>> b = a
>>> a==b
True
>>> a is b
True
```

Note: Python 缓存了整数和短字符串，因此每个对象在内存中只存有一份。

Python 没有缓存长字符串、列表及其他对象，可以有多个相同的对象。

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。

```
>>> class Foo(object):
...     def __eq__(self, other):
...         return True

>>> foo = Foo()
>>> foo == None
True
>>> 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

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

```
from functools import wraps
import time

def timer(func):
    @wraps(func)
    def function_timer(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print "time (s): ", end_time - start_time
        return result
    return function_timer

@timer
def foo():
    print "hello"
    return 0

print foo.__name__
print foo.__doc__
```

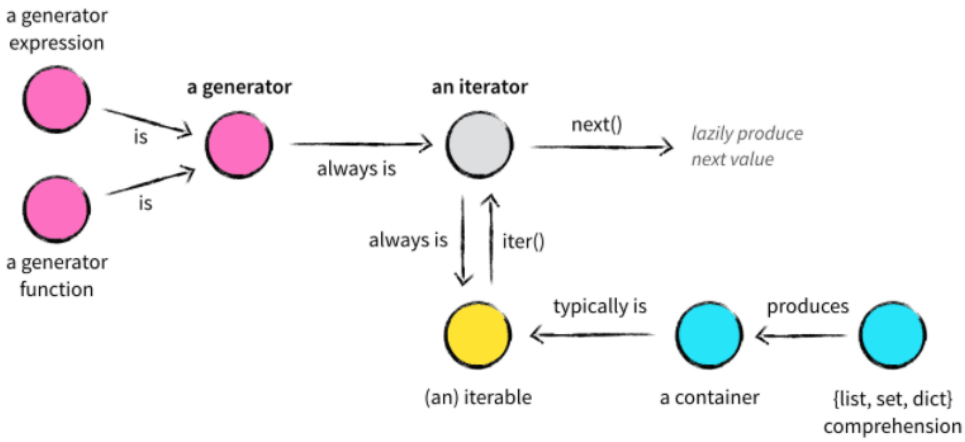
使用 wraps 可以保持函数 foo() 的属性 __name__ 和 __doc__ ，而不变成函数 function_timer 的相关属性。

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。

```
## enumerate 返回迭代器
a = enumerate(['a','b'])

for i in range(2): ## 迭代两次 enumerate 对象
    for x, y in a:
        print x, y
    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。

```
class Iterator_test(object):
  def __init__(self, data):
      self.data = data
      self.index = len(data)

  def __iter__(self):
      return self

  def next(self):
      if self.index <= 0:
          raise StopIteration
      self.index -= 1
      return self.data[self.index]

iterator_winter = Iterator_test('abcde')
for item in iterator_winter:
    print item
## 打印 e d c b a

class Iterator_test2(object):
    def __init__(self, data):
        self.data = data
    def __getitem__(self, it):
        return self.data[it]
no_iter = Iterator_test2('abcde')
for item in no_iter:
    print item
## 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：

```
from collections import Iterable
a = [1,2,3]
isinstance(a, Iterable) # True

a = iter(a)
next(a) # 或 a.next(), 返回 1
next(a) # 返回 2
next(a) # 返回 3
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 `yiled` 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。

```
def generator_winter():
  i = 1
  while i <= 3:
      yield i
      i += 1

generator_iter = generator_winter() ## 函数体中的代码并不会执行，只有显示或隐示地调用 next 的时候才会真正执行里面的代码。
print generator_iter.next() ## 1
print generator_iter.next() ## 2
print generator_iter.next() ## 3
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 ()）。

```
gen = (x for x in range(10)) ## <generator object <genexpr> at 0x0000000012BC4990>
for item in gen:
  print item

## fibonacci 数列
def fibonacci(n):
```

(continues on next page)

(continued from previous page)

```
  a, b = 0, 1
  while b <= n:
      yield b
      a, b = b, a+b
f = fibonacci(10)
for item in f:
    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。

```
plus = lambda x, y: x+y
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)`

```
a = [1, 2, 6, 5, 2, -8, -5, -1, -10]

## 每个元素加 1
b = map(lambda x: x+1, a) # [2, 3, 7, 6, 3, -7, -4, 0, -9]

## 提取序列中大于 0 的数
c = filter(lambda x:x>0, a) # [1, 2, 6, 5, 2]

## 返回所有元素相乘的结果
d = reduce(lambda x, y: x*y, a) # 48000

## 负数排在正数前面，同时绝对值大的排在后面
## 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
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：

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

```
>>> foo(1, 100, '200k', 300)
x: 1
args[0]: 100
args[1]: 200k
args[2]: 300

>>> args = [1,2,'abc']
>>> foo('A', *args)
x: A
args[0]: 1
args[1]: 2
args[2]: abc

>>> foo('A', args) ## 注：此时把 args 当做一个参数，参数类型为列表
x: A
args[0]: [1, 2, 'abc']
```

例二：

```
def foo(x, var1, var2, var3):
  print "x:", x
  print "var1:", var1
  print "var2:", var2
  print "var3:", var3
```

```
>>> args = [1,2,'A'] # list
>>> foo(1, args)
TypeError: foo() takes exactly 4 arguments (2 given)
>>> foo(1, *args)
x: 1
var1: 1
var2: 2
var3: A

>>> args = (1,2,'A') # tuple
>>> foo(1, args)
TypeError: foo() takes exactly 4 arguments (2 given)
>>> foo(1, *args)
x: 1
var1: 1
var2: 2
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:

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

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

例二:

```
def foo(x, var1=2, var2='a'):
  print "x:", x
  print "var1:", var1
  print "var2:", var2
```

```
>>> dict_input = {"var1": 10, "var2": "A"}
>>> foo(1, dict_input)
x: 1
var1: {'var1': 10, 'var2': 'A'}
var2: a

>>> foo(1, **dict_input)
x: 1
var1: 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)。

```
def foo(arg, *args, **kwargs):
  print "arg:", arg
  print "args:", args
  print "kwargs:", kwargs
```

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

>>> foo(1, x=1, y='b', 2, 3, 4)
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）。

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

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

2.7.4 参考资料

1. 大话 Python 中 *args 和 **kargs 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()

```
>>> s = "abcde"
>>> "-".join(s)
a-b-c-d-e
>>> s = ['abc', 'def', 'ghi']
>>> "-".join(s)
abc_def_ghi
>>> s
['abc', 'def', 'ghi']

>>> s = "a-b-c-d-e"
>>> s.partion('-') ## 只能分割为 3 部分
('a', '-', 'b-c-d-e')
>>> s.split('-')
['a', 'b', 'c', 'd', 'e']
>>> s
"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()

```
>>> a = [1,2,3]
>>> a.append(4)
```

(continues on next page)

(continued from previous page)

```
>>> a
[1, 2, 3, 4]
>>> a.extend([10,20,30])
>>> a
[1, 2, 3, 4, 10, 20, 30]

>>> a.insert(1, 5) ## 在第一个元素之后插入
>>> a
[1, 5, 2, 3, 4, 10, 20, 30]

>>> a.remove(2)
>>> a
[1, 5, 3, 4, 10, 20, 30]
>>> del a[3]
>>> a
[1, 5, 3, 10, 20, 30]

>>> a.sort(reverse=True)
>>> a
[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。

>>> a2 = a.pop(2)
>>> a2
10
>>> a
[30, 20, 5, 3, 1]
```

- **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()

```
>>> info ={
...       "name":"Tom",
...        "age":25,
...        "sex":"man",
...       }
>>> info.keys()
['age', 'name', 'sex']
```

(continues on next page)

(continued from previous page)

```
>>> info.values()
[25, 'Tom', 'man']
>>> info.items()
[('age', 25), ('name', 'Tom'), ('sex', 'man')]

>>> info.get(age)
25
>>> new = {"weight": 60}
>>> info.update(new)
>>> info
{'age': 25, 'name': 'Tom', 'weight': 60, 'sex': 'man'}
>>> info.clear()
>>> info
{}
```

  - **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)。

```
>>> from collections import defaultdict
>>> dd = defaultdict(list) ## 使用 list 作为 value type
defaultdict(<type 'list'>, {})
>>> dd['a']
[]
>>> dd['b'].append("hello")
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：&，|，-，^（交差补集，去除交集后剩下元素的并集），issubset() 、isupperset()。

```
>>> s1 = {'a', 'b', 'c'} ## 或者 s1 = set(['a', 'b', 'c'])
>>> s1.update({'e','d'})
>>> s1
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 <= 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。

```
>>> alist = [1,2,3]
>>> b = alist ## 引用
>>> c = alist[:] ## 复制
>>> alist.append(5)
>>> alist
[1, 2, 3, 5]
>>> b
[1, 2, 3, 5]
>>> c
[1, 2, 3]
>>> b[0] = -1
>>> a
[-1, 2, 3, 5]
>>> b
[-1, 2, 3, 5]
>>> c
[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。

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

(continues on next page)

(continued from previous page)

```
>>> alist
[1, 2, 3, ['a', 'b'], 5]
>>> a_copy
[1, 2, 3, ['a', 'b']]
>>> a_copy[0] = -1
>>> alist
[1, 2, 3, ['a', 'b'], 5]
>>> a_copy
[-1, 2, 3, ['a', 'b']]

>>> alist[3].append('c') ## 子对象的修改
>>> alist
[1, 2, 3, ['a', 'b', 'c'], 5]
>>> a_copy
[-1, 2, 3, ['a', 'b', 'c']]
>>> a_copy[3].append('d')
>>> alist
[1, 2, 3, ['a', 'b', 'c', 'd'], 5]
>>> a_copy
[-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。

```
>>> from copy import deepcopy
>>> alist = [1,2,3,['a','b']] ## ['a','b'] 是列表，是一个子对象
>>> a_copy = deepcopy(alist)
>>> alist[3].append('c') ## 子对象的修改
>>> alist
[1, 2, 3, ['a', 'b', 'c']]
>>> a_copy
[1, 2, 3, ['a', 'b']]
>>> a_copy[3].append('d')
>>> alist
[1, 2, 3, ['a', 'b', 'c']]
>>> a_copy
[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**。

```
>>> x = 'abcd'
>>> id(x)
313010056L
>>> y = 'abcd'
>>> id(y)
313010056L
## x 和 y 都是 str 对象的引用，值相同，占用同一块内存。

>>> a = [5, 3, 4, 3]
>>> id(a)
314009096L
>>> b = [5, 3, 4, 3] ## b = a[:]
>>> id(b)
314011080L
## a 和 b 的 id 不同，尽管值相同
>>> b.append(1)
>>> b
[5, 3, 4, 3, 1]
>>> id(b)
314011080L
## 改变 b，仍然是同一个对象，因此是可变对象
```

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([** $d_0, d_1, ..., d_n$ **])**
  - 生成 0~1 的随机浮点数，维度为 $d_0 \times d_1 \times ... \times d_n$ （缺省为 1）。
- **numpy.random.rand(** $d_0, d_1, ..., d_n$ **)**
  - 生成 0~1 的随机浮点数，维度为 $d_0 \times d_1 \times ... \times d_n$ （缺省为 1）。
- **numpy.random.randn(** $d_0, d_1, ..., d_n$ **)**
  - 标准正态分布。
- **numpy.random.randint(low, high=None, size=None, dtype='l')**
  - 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。

```
## 注：生成的数组都是 numpy array 类型

>>> 2.5 * np.random.randn(2, 4) + 3
[[-0.52410303  1.68461615 -0.04895917  2.81907944]
 [ 6.89754303  2.95949232  1.85296809  1.56361545]]

>>> np.random.randint(5, size=(2, 4))
[[0 4 2 3]
 [0 0 4 4]]

## 从 np.arange(4) 选取 3 个元素
>>> np.random.choice(4, 3)
[3 1 2]
>>> np.random.choice([1,3,9,0], 3)
[9 3 0]
>>> np.random.choice(4, 3, p=[0.1, 0.2, 0, 0.7])
[1 3 3]

>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> print arr
[6 9 0 8 1 7 4 5 3 2]

>>> np.random.permutation(10)
[5 8 9 7 3 1 0 2 6 4]
>>> np.random.permutation([1, 4, 9, 12, 15])
[ 9  1  4 12 15]
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
[[3 4 5]
 [6 7 8]
 [0 1 2]]

```

(continues on next page)

(continued from previous page)

```
>>> np.random.seed(1)
>>> np.random.random()
0.417022004702574
>>> np.random.seed(1)
>>> np.random.random()
0.417022004702574
>>> np.random.random()
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)
```

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

(continues on next page)

(continued from previous page)

```
>>> norm = la.norm(arr, axis=0, keepdims=True)
>>> print norm
[[ 2.82842708  1.41421354  2.23606801]]
>>> print arr / np.tile(norm,(2,1))
[[ 0.70710677  0.70710677  0.89442718]
 [ 0.70710677  0.70710677  0.44721359]]

>>> from sklearn import preprocessing
>>> print preprocessing.normalize(arr, axis=0, norm='l2')
[[ 0.70710677  0.70710677  0.89442718]
 [ 0.70710677  0.70710677  0.44721359]]

>>> import torch.nn.functional as F
>>> print F.normalize(torch.from_numpy(arr), p=2, dim=0)
0.7071  0.7071  0.8944
0.7071  0.7071  0.4472
[torch.FloatTensor of size 2x3]
```

2.10.4 k-means 实现

```
import numpy as np

## feature initialization
np.random.seed(1)
n = 10000
d = 3
K = 50
data = np.random.randn(n, d) ## n x d

## feature  normalization
data = data / np.tile(np.linalg.norm(data, axis=1, keepdims=True), (1, data.shape[1]))

## center initialization
center = data[np.random.permutation(n)[0:K]]

itr = 0
## loop
while itr < 20:
    itr += 1
    ## quantization
    similarity = np.dot(data, center.T)
    quan_id = np.argsort(-similarity, axis=1)[:, 0]
```

(continues on next page)

(continued from previous page)

```
23
24      ## update center
25      new_error = 0.0
26      for c in range(K):
27          data_c = data[quan_id == c]
28          if data_c.shape[0] != 0:
29            center[c] = np.mean(data_c, axis=0)
30            new_error += np.sum((data_c - center[c])**2)
31
32      if itr > 1 and abs(1 - new_error/old_error) < 1e-3:
33          break
34      old_error = new_error
```

2.10.5 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:

```
from Queue import PriorityQueue
que = PriorityQueue()
que.put((1,'apple'))
que.put((10,'app'))
que.put((5,'banana'))

while not que.empty():
    print que.get(), que.qsize()
```

(continues on next page)

(continued from previous page)

```
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])
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。

  如果插入元素是元组（tuple），则元组的第一项自动成为优先级，值越小，优先级越高。堆顶元素优先级最高，值最小。

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

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

```
>>> h1 = [32, 3, 5, 34, 54, 23, 132]
>>> h2 = [23, 2, 12, 656, 324, 23, 54]
>>> h1 = sorted(h1)
>>> h2 = sorted(h2)
>>> h = heapq.merge(h1, h2)
```

(continues on next page)

(continued from previous page)

```
>>> print type(h), list(h)
<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。

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

```
>>> nums = [1, 3, 4, 5, 2]
>>> print heapq.nlargest(3, nums)
[5, 4, 3]
>>> print heapq.nsmallest(3, nums)
[1, 2, 3]

>>> info = [
...     {'name': 'IBM', 'price': 91.1},
...     {'name': 'AAPL', 'price': 543.22},
...     {'name': 'FB', 'price': 21.09},
...     {'name': 'HPQ', 'price': 31.75},
...     {'name': 'YHOO', 'price': 16.35},
...     {'name': 'ACME', 'price': 115.65}
... ]
>>> cheap = heapq.nsmallest(2, info, key=lambda x:x['price'])
>>> expensive = heapq.nlargest(2, info, key=lambda x:x['price'])
>>> print cheap
[{'price': 16.35, 'name': 'YHOO'}, {'price': 21.09, 'name': 'FB'}]
>>> print expensive
[{'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 $\mathcal{O}(NlogK)$。

```
import heapq as hq

class TopKHeap(object):
  def __init__(self, k=3):
    self.k = k
    self.data = []

  def push(self, x):
    if len(self.data) < self.k:
      hq.heappush(self.data, x)
    else:
      min_number = self.data[0]
      if x > min_number:
        hq.heapreplace(self.data, x)

  def topk(self):
    return list(reversed([hq.heappop(self.data) for _ in range(len(self.data))]))

def main():
  nums = range(1, 10)
  tkh = TopKHeap(3)
  for n in nums:
    tkh.push(n)
  print tkh.topk() ## [9, 8, 7]

if __name__ == '__main__':
  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。

```
>>> cnt = Counter() ## 空计数器
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
>>> cnt = ['red', 'blue', 'red', 'green', 'blue', 'blue']
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> cnt.most_common(2) ## 返回出现频率最高的两个元素
[('blue', 3), ('red', 2)]

>>> c = Counter('gallahad')
>>> c
Counter({'a': 3, 'l': 2, 'h': 1, 'g': 1, 'd': 1})

>>> c = Counter({'red': 4, 'blue': 12})
>>> c
Counter({'blue': 12, 'red': 4})
>>> c['green'] ## 访问不存在关键字，可使用 c.get('green')
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

2.12 逻辑运算与布尔测试

2.12.1 逻辑运算

Table 1: 逻辑运算

运算符	描述
and	与，x and y：如果 x 为 False，返回 False，否则返回 y 的 **计算值**
or	或，x or y：如果 x 为 True，返回 True，否则返回 y 的 **计算值**
not	非，not x：如果 x 为 True，返回 False，否则返回 True

```
>>> print 'a'<'b' and 'c'
c
>>> print 'a'>'b' and 'c'
False

>>> print 'a'>'b' or 'c'
c
>>> print 'a'<'b' or 'c'
True
```

2.12.2 成员运算

Table 2: 成员运算

运算符	描述
in	如果在指定列表/元组/字符串/字典中找到值，返回 True，否则返回 False
not in	如果未在指定列表/元组/字符串/字典中找到值，返回 True，否则返回 False

```
>>> 1 in [1,2,3]
True
>>> 1 in (1,2)
True
```

(continues on next page)

(continued from previous page)

```
>>> 'a' in 'abc'
True

>>> cnt = {'a':1}
>>> 'a' in cnt
True
>>> {'a':1} in cnt
TypeError: unhashable type: 'dict'
```

Note: 查找的值必须是可哈希的，也就是不可变类型。

2.12.3 布尔测试

下面对象的布尔值都是 `False`：

- False（布尔类型）
- None
- “”（空字符串）
- []（空列表）
- {}（空字典）
- ()（空元组）
- 所有值为零的数
  - 0（整型）
  - 0.0（浮点型）
  - 0L（长整型）
  - 0.0 + 0.0j（复数）

```
flag = None
if not flag:
    print "flag is none:", flag
    print flag==False

flag = {}
if not flag:
    print "flag is empty:", flag
    print flag == False
```

(continues on next page)

(continued from previous page)

```
flag = {'a':1}
if flag:
    print "flag is not empty:", flag

flag = []
if not flag:
    print "flag is empty:", flag
    print flag == False
flag = [1]
if flag:
    print "flag is not empty:", flag

flag = ""
if not flag:
    print "flag is empty:", flag
    print flag == False
flag = "a"
if flag:
    print "flag is not empty:", flag

flag = 0.0
if not flag:
    print "flag is zero:", flag
    print flag == False
flag = 1
if flag:
    print "flag is not empty:", flag
```

输出结果为:

```
flag is none: None
False

flag is empty: {}
False
flag is not empty: {'a': 1}

flag is empty: []
False
flag is not empty: [1]

flag is empty:        ## ""
False
flag is not empty: a
```

(continues on next page)

(continued from previous page)

```
flag is zero: 0.0
True                    ## 只有这一个是 True: 0.0 == False
flag is not empty: 1
```

Note: 布尔值是 False，不代表等于 False。

零的布尔值是 False，同时也等于 False。

2.12.4 参考资料

1. Python 基本数据类型

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

2.13 命名规范

2.13.1 后缀单下划线

避免与关键字冲突。

```
class_ = 1
```

2.13.2 前缀单下划线

不能被 `from module_name import *` 导入。

对于类的成员变量和成员函数

- `_var`： 保护成员（protected），类对象可以在外部访问
- `_func`： 保护成员（protected），类对象可以在外部访问

2.13.3 前缀双下划线

不能被 `from module_name import *` 导入。

对于类的成员变量和成员函数

- `__var`： 私有成员（private），类对象不可以在外部访问
- `__func`： 私有成员（private），类对象不可以在外部访问

```
## ac.py
_global = 10
def _func():
    print "_func"

class A:
  var = 100         ## 类变量 (类对象共有)
  def __init__(self):
      self._a = 0  ## protected
      self.__b = 1 ## private
      self.c = 2

  def _foo(self):
      print "_foo"

  def __foo(self):
      print "__foo"

  def foo(self):
      print "foo"
      print "__b:", self.__b
```

```
>>> from ac import *
>>> obj = A()
>>> print obj.var
100
>>> print obj.c
2
>>> obj.foo()
foo
__b: 1

>>> print obj._a
0
>>> print _foo()
_foo

>>> print obj.__b
AttributeError: A instance has no attribute '__b'

>>> print _global
NameError: name '_global' is not defined
```

2.13.4 前后双下划线

内建方法，如:

```
__doc__
__name__
```

2.13.5 参考资料

1. python 命名规范

   https://www.jianshu.com/p/a793c0d960fe

2.14 类变量与类方法

2.14.1 类变量

定义在类的开始。

- 类和实例都可以访问类变量
- 实例只可以访问，不可以修改

2.14.2 实例变量

类实例（对象）可使用的变量，以 `self.` 开头。

2.14.3 实例方法

类实例（对象）可调用的函数，形参包括 `self` 。

2.14.4 静态方法

使用 `@staticmethod` 装饰，是定义在类内的普通函数。

- 静态方法不能访问类变量、实例变量、实例方法
- 类和实例都可以访问静态方法

2.14.5 类方法

使用 `@classmethod` 装饰，形参包括 `cls` 。

- 类方法可以访问和修改类变量，不能访问实例变量、实例方法
- 类和实例都可以访问类方法

```
class A:
  var = 100        ## 类变量
  def __init__(self):
      self._a = 0  ## 实例变量
      self.__b = 1 ## 实例变量
      self.c = 2   ## 实例变量

  def _foo(self):
      print "_foo"

  def __foo(self):
      print "__foo"

  def foo(self):
      self.__foo()
      print "__b:", self.__b

  @staticmethod
  def static_func(a, b):
      print "static_method"
      return a + b

  @classmethod
  def class_func(cls, num):
      print "class_method"
      cls.var = num
      print cls.static_func(-1, 1)
```

```
>>> obj = A()
>>> print obj.static_func(1,3)
static_method
4
>>> A.class_func(200)
class_method
static_method
0

```

(continues on next page)

(continued from previous page)

```
>>> print A.var
200
```

2.14.6 参考资料

1. Python-类变量，成员变量，静态变量，类方法，静态方法，实例方法，普通函数

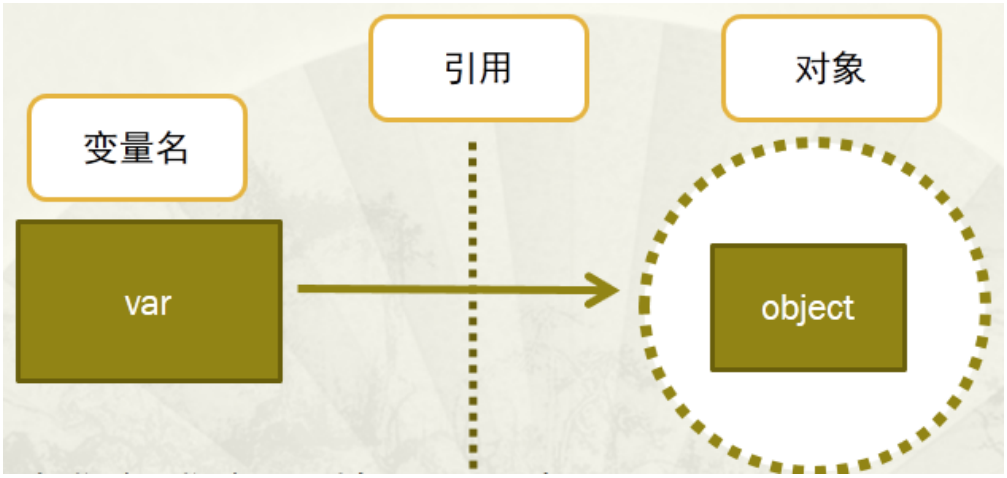
   https://www.cnblogs.com/20150705-yilushangyouni-Jacksu/p/6238187.html

2. 一张图了解 python 类方法与类变量类变量与实例变量

   https://blog.csdn.net/cgqdtc/article/details/80555319

2.15 内存管理

2.15.1 变量与对象



(图片来源：https://www.cnblogs.com/geaozhang/p/7111961.html)

变量 通过变量指针引用对象，变量指针指向具体对象的内存空间，取对象的值。

对象 类型已知，每个对象都包含一个头部信息（类型标识符和引用计数器）

变量名没有类型，类型属于对象。

```
>>> a = "hello"
>>> b = "hello"
>>> a==b
True
>>> a is b
```

(continues on next page)

(continued from previous page)

```
True
>>> id(a)
140568052594368
>>> id(a)
140568052594368

>>> a = "hello world"
>>> b = "hello world"
>>> a==b
True
>>> a is b
False
>>> id(a)
140568052594752
>>> id(a)
140568052594320

>>> a = [1,2,3]
>>> b = a
>>> a==b
True
>>> a is b
True
```

Note: Python 缓存了整数和短字符串，因此每个对象在内存中只存有一份，赋值语句只是创造新的引用，而不是对象。

Python 没有缓存长字符串、列表及其他对象，可以有多个相同的对象，赋值语句创建出新的对象。

变量的改变

不可变对象 赋值、加减乘除这些操作实际上导致变量指向的对象发生了改变（已经不是指向原来的那个对象了），并不是通过这个变量来改变它指向的对象的值。

```
>>> a = 10
>>> id(a)
21856416
>>> a = a - 1
>>> id(a)
21856440
>>> a *= 2
```

(continues on next page)

(continued from previous page)

```
>>> id(a)
21856224
```

可变对象 对于 list、dict 对象，此时变量的指向没有改变。

```
>>> a = []
>>> id(a)
140568052448936
>>> a.append(1)
>>> id(a)
140568052448936
```

2.15.2 引用计数

```
from sys import getrefcount
```

使用 sys 包中的 getrefcount()，来查看某个对象的引用计数。需要注意的是，当使用某个引用作为参数，传递给 getrefcount() 时，参数实际上创建了一个临时的引用。因此，getrefcount() 所得到的结果，会比期望的多 1。

普通引用

```
>>> a = [1,2,3]
>>> getrefcount(a)
2
>>> b = a
>>> getrefcount(a)
3
>>> getrefcount(b)
3
>>> del b
>>> getrefcount(a)
2


>>> getrefcount(1)
2418
>>> n = 1
>>> getrefcount(1)
2419
>>> m = n
```

(continues on next page)

(continued from previous page)

```
>>> getrefcount(1)
2420
>>> del n
>>> getrefcount(1)
2419
>>> n = [1,2,3]
>>> getrefcount(1)
2420
>>> m = 2
>>> getrefcount(1)
2419
```

容器对象

Python 的容器对象 (container)，比如列表、元组、字典等，可以包含多个对象。**容器对象中包含的并不是元素对象本身，是指向各个元素对象的引用**。

```
>>> a = [1,2,3]
>>> getrefcount(a)
2
>>> b = [a, a]
>>> getrefcount(a)
4
```

循环引用

只有容器对象才会产生循环引用的情况，比如列表、字典、用户自定义类的对象、元组等。而像数字、字符串这类简单类型不会出现循环引用。

```
>>> a = []
>>> t = [a]
>>> getrefcount(a)
3
>>> a.append(t)
>>> getrefcount(a)
9
```

2.15.3 垃圾回收

```
>>> import gc
>>> print gc.get_threshold()
(700, 10, 10)
## 700 是垃圾回收启动的阈值，10 是与分代回收相关的阈值
```

当 Python 的某个对象的引用计数降为 0 时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。频繁的垃圾回收（garbage collection），将大大降低 Python 的工作效率。如果内存中的对象不多，就没有必要总启动垃圾回收。所以，Python 只会在特定条件下，自动启动垃圾回收。

当 Python 运行时，会记录其中分配对象（object allocation）和取消分配对象（object deallocation）的次数。当两者的差值高于某个阈值时，垃圾回收才会启动，清除那些引用计数为 0 的对象。

垃圾检查

`gc.get_count()` 获取一个三元组，如 `(488, 3, 0)`。

- `488` 是指距离上一次 0 代垃圾检查，Python 分配内存的数目减去释放内存的数目。
- `3` 是指距离上一次 1 代垃圾检查，0 代垃圾检查的次数。
- `0` 是指距离上一次 2 代垃圾检查，1 代垃圾检查的次数。

分代回收

Python 将所有的对象分为 0，1，2 三代。所有的新建对象都是 0 代对象。当某一代对象经历过垃圾回收，依然存活，那么它就被归入下一代对象。垃圾回收启动时，一定会扫描所有的 0 代对象。如果 0 代经过一定次数垃圾回收，那么就启动对 0 代和 1 代的扫描清理。当 1 代也经历了一定次数的垃圾回收后，那么会启动对 0，1，2，即对所有对象进行扫描。

`(700, 10, 10)` 表明：每 10 次 0 代垃圾回收，会配合 1 次 1 代的垃圾回收；每 10 次 1 代的垃圾回收，才会有 1 次的 2 代垃圾回收。

标记-清除

Python 采用了“标记-清除”(Mark and Sweep) 算法，解决容器对象可能产生的循环引用问题。

- 标记阶段：遍历所有的对象，如果是可达的（reachable），也就是还有对象引用它，那么就标记该对象为可达；
- 清除阶段：再次遍历对象，如果发现某个对象没有标记为可达，则就将其回收。

2.15.4 参考资料

1. Python 内存管理机制

   https://www.cnblogs.com/geaozhang/p/7111961.html

2. Python 的内存管理

   https://www.cnblogs.com/vamei/p/3232088.html

3. Python 垃圾回收机制详解

   https://www.cnblogs.com/Xjng/p/5128269.html

4. 聊聊 Python 内存管理

   https://andrewpqc.github.io/2018/10/08/python-memory-management/

5. [Python] 内存管理

   https://chenrudan.github.io/blog/2016/04/23/pythonmemorycontrol.html

2.16 __new__ 和 __init__

2.16.1 老式类与新式类

Python 2.x 中类的定义分为新式定义和老式定义两种。老式类定义时默认是继承自 `type` ，而新式类在定义时显示地继承 `object` 类。

```
class A: ## 老式类
  pass

class B(object): ## 新式类
  pass
```

```
>>> print A.__bases__
()
>>> print dir(A)
['__doc__', '__module__']

>>> print B.__bases__
(<type 'object'>,)
>>> print dir(B)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__',
↪'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '_
↪_sizeof__', '__str__', '__subclasshook__', '__weakref__']
>>> print B.__class__
<type 'type'>
```

Python 3.x 中没有新式类和老式类之分，它们都继承自 `object` 类，因此可以不用显示地指定其基类。

2.16.2 老式类

老式类中其实并没有 `__new__` 方法，因为 `__init__` 就是它的构造方法（函数）。即使重写 `__new__` 方法，也永远不会执行。

`__init__` 只能返回 `None`。

2.16.3 新式类

功能

新式类中，`__new__`（构造函数）单独地 **创建**一个对象，而 `__init__`（初始化函数）负责 **初始化**这个对象。

- `__new__` 至少要有一个参数 `cls`，代表要实例化的类，此参数在实例化时由 Python 解释器自动提供。
- `__init__` 有一个参数 `self`，就是 `__new__` 返回的实例，`__init__` 在 `__new__` 的基础上可以完成一些其它初始化的动作，`__init__` 不需要返回值（或者说返回 `None` ）。

返回值

`__init__` 只能返回 `None`。

`__new__` 返回创建的实例对象并传递给 `__init__` 的 `self` 参数。如果 `__new__` 没有返回值，或者没有正确返回 **当前类** `cls` 的实例，则 `__init__` 不会被调用。

```
class A(object):

  def __new__(cls):
      print "A.__new__ called"
      print cls
      return super(A, cls).__new__(cls)

  def __init__(self):
      print "A.__init__ called"
```

```
>>> a = A()
A.__new__ called
<class '__main__.A'> ## cls
A.__init__ called

>>> a.__class__      ## type(a)
<class '__main__.A'>
>>> A.__class__
<type 'type'>
```

`__new__` 返回父类的对象：

```
class A(object):
  pass

class B(A):
  def __new__(cls):
    print "B.__new__ called"
    return A() ## 或者写为:  return super(B,cls).__new__(A)

  def __init__(self):  ## 不会被调用
    print "B.__init__ called"
```

```
>>> b = B()
B.__new__ called
>>> print type(b)
<class '__main__.A'>
```

___new___ 实现单例

单例（singleton）：类只有一个对象。None 就是一个单例，所有的变量只要是 None ，它一定和 None 指向同一个内存地址。

```
class Singleton(object):
  _instance = None
  def __new__(cls, *args, **kwargs):
      if cls._instance is None:
          cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)

      return cls._instance
```

```
>>> s1 = Singleton()
>>> print id(s1)
317973448
>>> s2 = Singleton()
>>> print id(s2)
317973448
```

2.16.4 附：___repr___ 和 ___str___

```
class Base(object):
    def __init__(self, name="fong"):
        self.name = name
```

(continues on next page)

(continued from previous page)

```

class A(Base):
    def __repr__(self):
        return "Class A(%s)" % self.name

class B(Base):
    def __str__(self):
        return "Class B(%s)" % self.name
```

```
>>> a = A()
>>> a
Class A(fong)
>>> print a
Class A(fong)

>>> b = B()
>>> b
<B object at 0x0000000012B7FB70>
>>> print b
Class B(fong)
```

2.16.5 参考资料

1. 深入理解 Python 中的 ___new___ 和 ___init___

   https://blog.csdn.net/luoweifu/article/details/82732313

2. 详解 Python 中的 ___init___ 和 ___new___（静态方法）

   https://www.cnblogs.com/nyist-xsk/p/8286941.html

3. Python 面试之理解 ___new___ 和 ___init___ 的区别

   https://juejin.im/post/5add4446f265da0b8d4186af

4. Python 中 ___repr___ 和 ___str___ 区别

   https://blog.csdn.net/luckytanggu/article/details/53649156

2.17 pip

`pip` 是 python 的包管理工具。

2.17.1 基本指令

```
Commands:
install                     Install packages.
download                    Download packages.
uninstall                   Uninstall packages.
freeze                      Output installed packages in requirements format.
list                        List installed packages.
show                        Show information about installed packages.
check                       Verify installed packages have compatible dependencies.
search                      Search PyPI for packages.
wheel                       Build wheels from your requirements.
hash                        Compute hashes of package archives.
completion                  A helper command used for command completion.
help                        Show help for commands.
```

- 安装

```
pip install <包名>
pip install -r requirements.txt
```

requirements.txt 使用 == >= <= > < 来指定版本，不写则默认为最新版本，格式如:

```
APScheduler==2.1.2
Django==1.5.4
MySQL-Connector-Python==2.0.1
MySQL-python==1.2.3
PIL==1.1.7
South==1.0.2
```

- 卸载

```
pip uninstall <包名>
pip uninstall -r requirements.txt
```

- 升级包

```
pip install -U <包名>
pip install <包名> --upgrade
```

- 升级 pip

```
pip install -U pip
python -m pip install --upgrade pip
```

- freeze：查看已经安装的包及版本信息

- list：列出已安装的包

```
pip list -o ## 查询可升级的包
```

- search：在 PyPI 查询包

```
pip search <包名>
```

- show：显示已安装的包的信息

```
pip show <包名>
```

Note: 如果不能直接使用 pip 命令，可能是因为安装目录不在系统的 PATH 中，此时可以执行:

```
python -m pip <pip arguments>
py -m pip <pip arguments> ## Windows
```

使用清华源加速安装:

```
pip install <包名> -i https://pypi.tuna.tsinghua.edu.cn/simple
```

2.17.2 参考资料

1. Python pip 常用命令

   https://www.cnblogs.com/BlueSkyyj/p/8268621.html

2. User Guide

   https://pip.pypa.io/en/stable/user_guide/

CHAPTER

THREE

LINUX/SHELL

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

```
## 将用户输入的数据同时保存到 out1 out2
$ tee out1 out2
a b c d e f g # 输入
a b c d e f g # 反馈
^C            # 结束输入
$ cat out1
a b c d e f g
$ cat out2
a b c d e f g

## 管道：将屏幕的输出保存到文件
$ echo "hello world" | tee out
hello world
$ cat out
hello world

## 把 python 程序打印到屏幕的内容保存到文件
$ 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

```
$ which python
/usr/bin/python

$ whereis -s -S /usr/lib -f python
python: /usr/lib/python3.5 /usr/lib/python2.7

$ locate /usr/bin/pytho ## 以 pytho 开头的文件
/usr/bin/python
/usr/bin/python-config
/usr/bin/python2
...
```

(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，

```
$ umask
0002

## 临时修改默认权限 （若要长期生效，把 umask 0022 写进 ~/.bashrc 并 source）
$ umask 0022
$ umask
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（即以递回的方式逐个变更）

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。

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

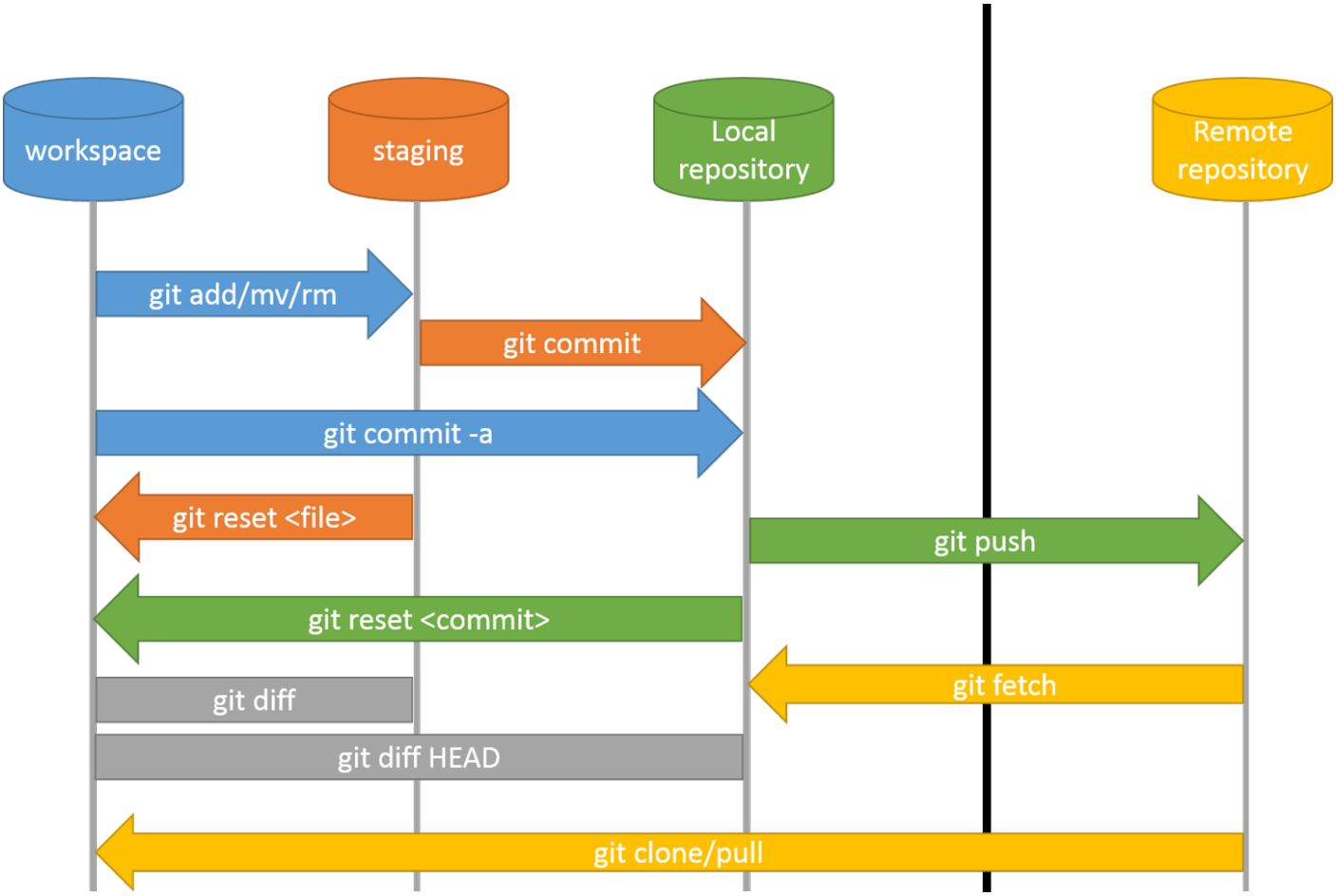
CHAPTER

FOUR

GIT

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。

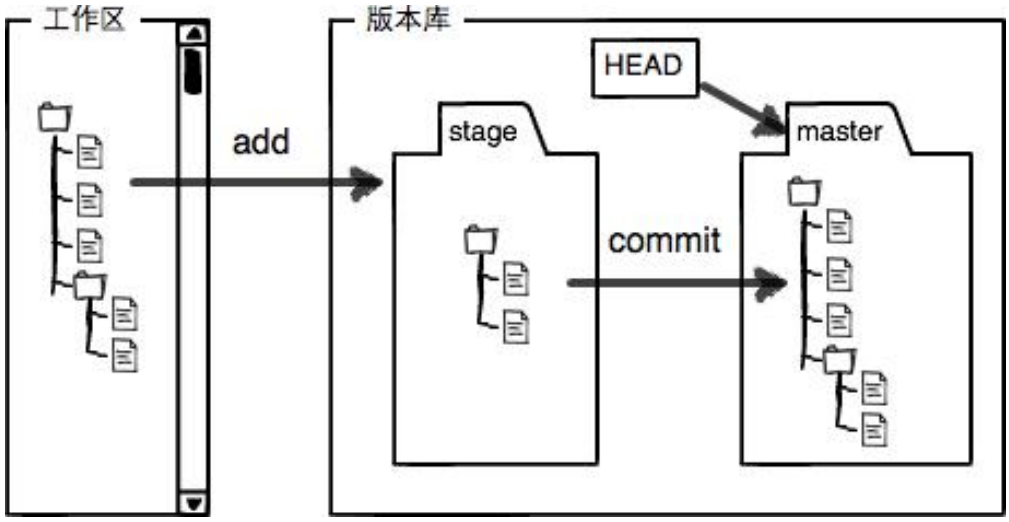
git add -u ： git add - -update，仅监控已经被 add 的文件（即 **tracked file** ），他会将被修改的文件提交到暂存区。不会提交新文件（untracked file）。

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。

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

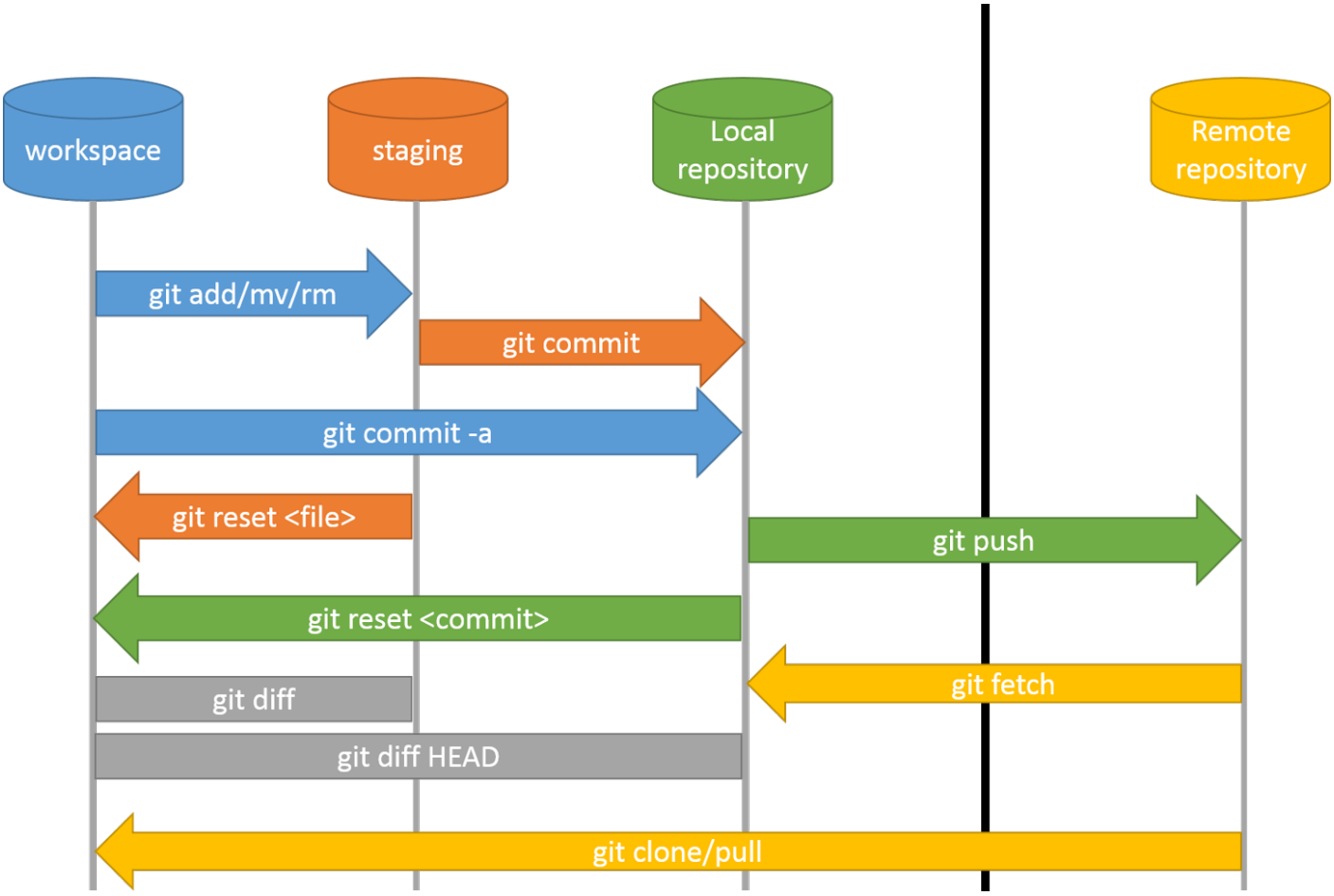
4. Git 操作详解

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

5. git 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。

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

(continues on next page)

(continued from previous page)

```
origin  git@github.com:******/******.git (push)
## 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/schaepher/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

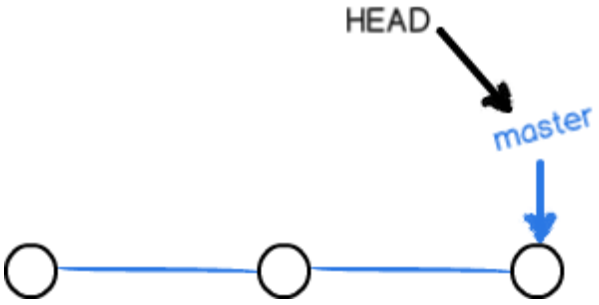
- `git checkout -b dev` 创建分支 dev，并切换到 dev 分支（此时可以正常 add，commit 等）。相当于下面两条指令：
  - `git branch dev` 创建分支
  - `git checkout dev` 切换分支

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。

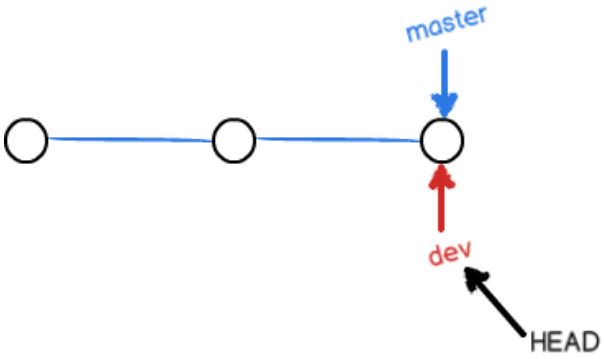
- `git branch` 查看当前分支。
- `git merge dev` 把 dev 分支的内容合并到当前分支（如，master 分支）。
- `git branch -d dev` （合并之后）删除 dev 分支。

流程如下（图片来源于 Git 教程 - 廖雪峰的官方网站 ）：

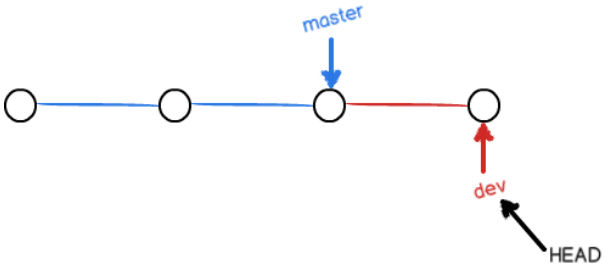
1. 初始状态：**master** 分支。



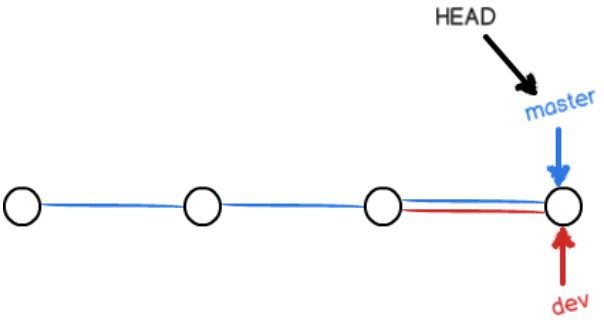
2. 创建并切换到 **dev** 分支。



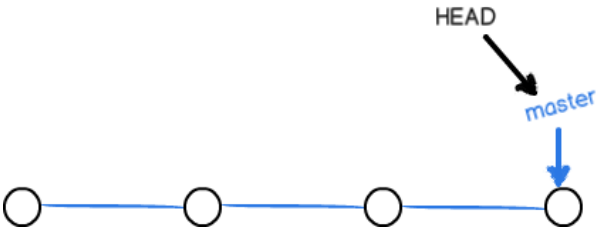
3. 更新 **dev** 分支。



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/schaepher/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

CHAPTER

FIVE

机器学习

5.1 Logistic Regression

模型：

$$h_\theta(\mathbf{x}) = g(\theta^\top \mathbf{x}),$$

$$g(z) = \frac{1}{1+e^{-z}},$$

$$g'(z) = (1-g(z))g(z) \in (0, 0.25].$$

对数损失函数（极大似然）：

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

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 $y \sim Bernoulli(\phi)$
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：

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

  只与 $y^{(i)}, \mathbf{x}^{(i)}$ 有关，与 h_θ 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，θ 更新与 h_θ 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 θ_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点总结

   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

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（几何间隔，geometric margin）为：

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

函数间隔（functional margin）：

$$y(w^\top x + b).$$

原始问题：

$$\begin{aligned} &\underset{w,b}{min} \\ &\frac{1}{2}\|w\|^2 \\ &s.t. \\ &y_i(w^\top x_i + b) \geqslant 1, i = 1, 2, ..., m \end{aligned}$$

拉格朗日函数：

$$L(w, b, \alpha) = \frac{1}{2} w^\top w + \sum_{i=1}^{m} \alpha_i (1 - y_i(w^\top x_i + b))$$

目标函数：

$$\underset{w,b}{min}(\underset{\alpha \geqslant 0}{max} L(w, b, \alpha))$$

对偶问题：

$$\underset{\alpha \geqslant 0}{max}(\underset{w,b}{min} L(w, b, \alpha))$$

令 L 对 w 和 b 的偏导为 0 得：

$$\begin{aligned} w = &\sum_{i=1}^{m} \alpha_i y_i x_i, \\ 0 = &\sum_{i=1}^{m} \alpha_i y_i. \end{aligned}$$

对偶问题变成：

$$\max_{\alpha \geqslant 0} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j x_i^\top x_j,$$

$$s.t.$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0,$$

$$\alpha \geqslant 0,\ i = 1, 2, ..., m.$$

KKT 条件：

$$y_i(w^\top x_i + b) \geqslant 1,$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0,$$

$$\alpha_i(1 - y_i(w^\top x_i + b)) = 0.$$

5.2.1 核函数

核函数 $\mathcal{K}$

- 对称半正定。($\mathcal{K} \geqslant 0 : \forall z,\ z^\top \mathcal{K} z \geqslant 0.$)
- 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 σ 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 α 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/fengzhizizhizizhizi/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

$$\max_{W} \quad Tr(W^\top X X^\top W) \quad s.t. \quad W^\top W = I.$$

$$X \in \mathbb{R}^{d \times m},$$

$$W \in \mathbb{R}^{d \times d'},$$

$$d' < d.$$

5.3.2 求解

1. 计算样本的协方差矩阵 $C = XX^\top$ ；
2. 对协方差矩阵做特征值分解（EVD）；
3. 取最大的 d' 个特征值 $(\lambda_1, \lambda_2, ..., \lambda_{d'})$ 对应的特征向量：

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

5.3.3 PCA-Whitening

白化（Whitening）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

$$x_{rot} = W^\top x,$$

$$x_{pca\ white,i} = \frac{x_{rot,i}}{\sqrt{\lambda_i}}$$

5.3.4 SVD 分解

$$A = U\Sigma V^{\top}, A \in \mathbb{R}^{m\times n}, r = rank(A), U \in \mathbb{R}^{m\times r}, \Sigma \in \mathbb{R}^{r\times r}, V \in \mathbb{R}^{n\times r}.$$

其中 U 是 $AA^{\top}$ 的特征向量矩阵，V 是 $A^{\top}A$ 的特征向量矩阵。

当 d 很大时，$C = XX^{\top}$ 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。

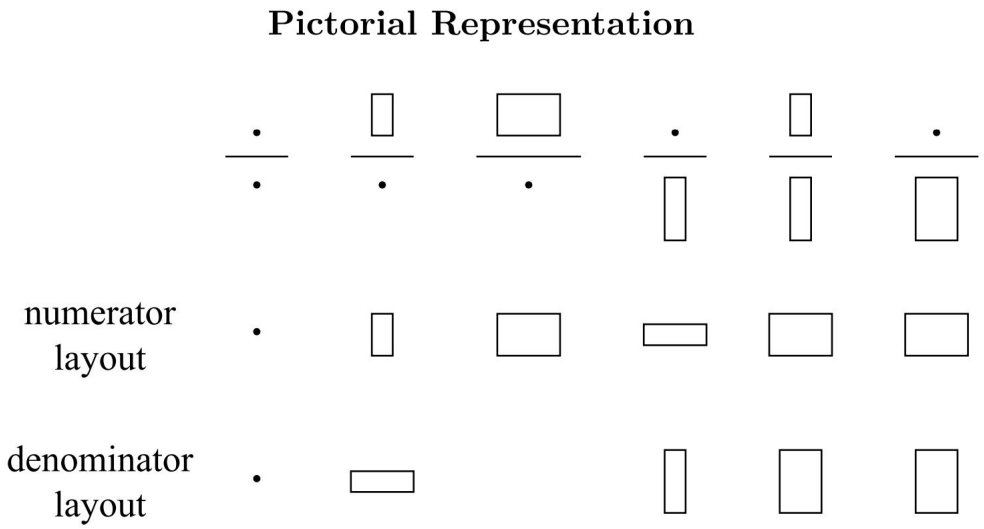
5.3.5 参考资料

1. 周志华《机器学习》Page 229 – 232。
2. ufldl

   http://ufldl.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：



这里使用 **denominator layout** 。

$\mathbf{x}, \mathbf{a}, \mathbf{b}$ 均为 **列向量**。

5.4.1 一阶

$$\frac{\partial(\mathbf{x}^\top \mathbf{a})}{\partial \mathbf{x}} = \frac{\partial(\mathbf{a}^\top \mathbf{x})}{\partial \mathbf{x}} = \mathbf{a}$$

$$\frac{\partial(\mathbf{a}^\top \mathbf{X} \mathbf{b})}{\partial \mathbf{X}} = \mathbf{a}\mathbf{b}^\top$$

$$\frac{\partial(\mathbf{a}^\top \mathbf{X}^\top \mathbf{b})}{\partial \mathbf{X}} = \mathbf{b}\mathbf{a}^\top$$

$$\frac{\partial(\mathbf{a}^\top \mathbf{X} \mathbf{a})}{\partial \mathbf{X}} = \mathbf{a}\mathbf{a}^\top$$

5.4.2 二阶

$$\frac{\partial(\mathbf{x}^\top \mathbf{x})}{\partial \mathbf{x}} = 2\mathbf{x}$$

$$\frac{\partial(\mathbf{x}^\top \mathbf{B}\mathbf{x})}{\partial \mathbf{x}} = (\mathbf{B} + \mathbf{B}^\top)\mathbf{x}$$

$$\frac{\partial(\mathbf{a}^\top \mathbf{X}^\top \mathbf{X}\mathbf{b})}{\partial \mathbf{X}} = \mathbf{X}(\mathbf{a}\mathbf{b}^\top + \mathbf{b}\mathbf{a}^\top)$$

5.4.3 迹

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

$$\frac{\partial(Tr(\mathbf{X}))}{\partial \mathbf{X}} = \mathbf{I}$$

$$\frac{\partial(Tr(\mathbf{XA}))}{\partial \mathbf{X}} = \mathbf{A}^\top$$

$$\frac{\partial(Tr(\mathbf{X}^\top \mathbf{A}))}{\partial \mathbf{X}} = \frac{\partial(Tr(\mathbf{A}\mathbf{X}^\top))}{\partial \mathbf{X}} = \mathbf{A}$$

$$\frac{\partial(Tr(\mathbf{AXB}))}{\partial \mathbf{X}} = \mathbf{A}^\top \mathbf{B}^\top$$

$$\frac{\partial(Tr(\mathbf{A}\mathbf{X}^\top \mathbf{B}))}{\partial \mathbf{X}} = \mathbf{BA}$$

$$\frac{\partial(Tr(\mathbf{X}\mathbf{X}^\top))}{\partial \mathbf{X}} = 2\mathbf{X}$$

$$\frac{\partial(Tr(\mathbf{X}^2))}{\partial \mathbf{X}} = 2\mathbf{X}^\top$$

$$\frac{\partial(Tr(\mathbf{X}\mathbf{B}\mathbf{X}^\top))}{\partial \mathbf{X}} = \mathbf{X}\mathbf{B}^\top + \mathbf{XB}$$

$$\frac{\partial(Tr(\mathbf{X}^\top \mathbf{B}\mathbf{X}))}{\partial \mathbf{X}} = \mathbf{B}\mathbf{X} + \mathbf{B}^\top \mathbf{X}$$

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

5.5 牛顿方法

牛顿方法（Newton’s method, Newton-Raphson method）可以有效地近似实值函数的根。

5.5.1 一维

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

5.5.2 高维

$$x_{n+1} = x_n - J^{-1}F(x_n).$$

$x \in \mathbb{R}^k$, $F : \mathbb{R}^k \to \mathbb{R}^k$, $J \in \mathbb{R}^{k\times k}$, [Jacobian matrix]

$$J_{ij} = \frac{\partial F_i}{\partial x_j}.$$

由于求解 Jacobian matrix 的逆复杂度较高，因此可以通过求解以下等式来节省时间开销：

$$J \cdot (x_{n+1} - x_n) = -F(x_n).$$

5.5.3 收敛问题

- 初始点问题

- 导数为 0，出现除 0 操作

$$f(x) = 1 - x^2, x_0 = 0, f'(x_0) = 0.$$

- 死循环

$$f(x) = x^3 - 2x + 2, x_0 = 0, x_1 = 1, x_2 = 0, x_3 = 1, ...$$

- 根的导数不存在或不连续
- 有时候收敛速度慢

5.5.4 应用

- 最大/最小化问题
  - 一维

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

  - 高维

$$x_{n+1} = x_n - H^{-1}\nabla F(x), H_{ij} = \frac{\partial^2 F}{\partial x_i \, \partial x_j}.\text{[Hessian matrix]}$$

- 求倒数（reciprocal）

$$f(x) = a - \frac{1}{x},$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n(2 - ax_n).$$

- 开根号（sqrt）

$$f(x) = x^2 - a,$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x^2 - a}{2x}.$$

```
def Sqrt(a):
  x = a
  while abs(x*x - a) > 1e-3:
      x = x - (x*x - a) / float(2 * x)
  return x
```

5.5.5 参考资料

1. Wikipedia: Newton’s method

   https://en.wikipedia.org/wiki/Newton%27s_method

CHAPTER

SIX

深度学习

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：

```
# 在 GPU 上运行
model.cuda()
# 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
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：

```
# 定义网络结构
self.layer1 = nn.Linear(227, 128)
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

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

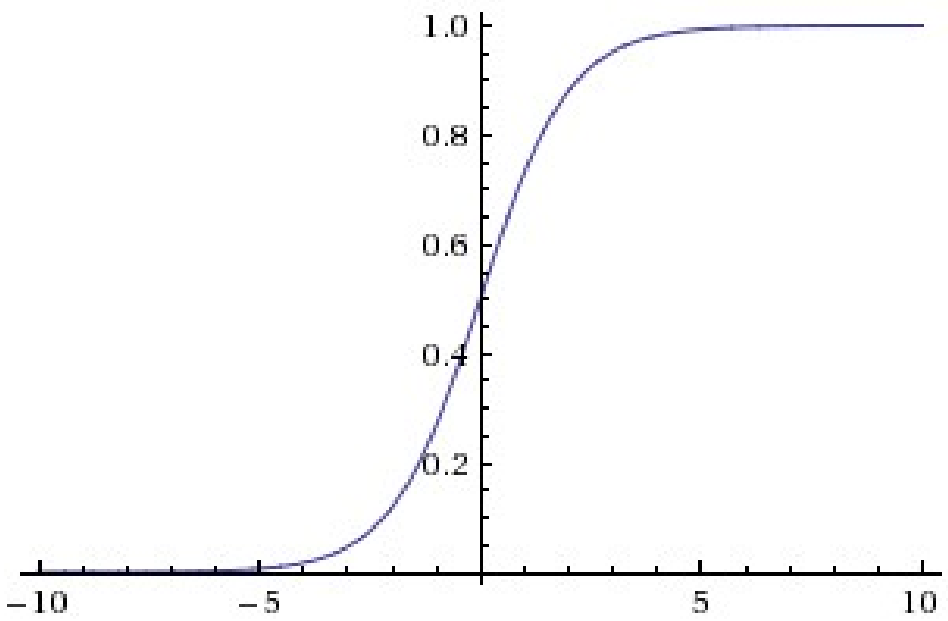
sigmoid 函数的数学表达式如下：

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

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：

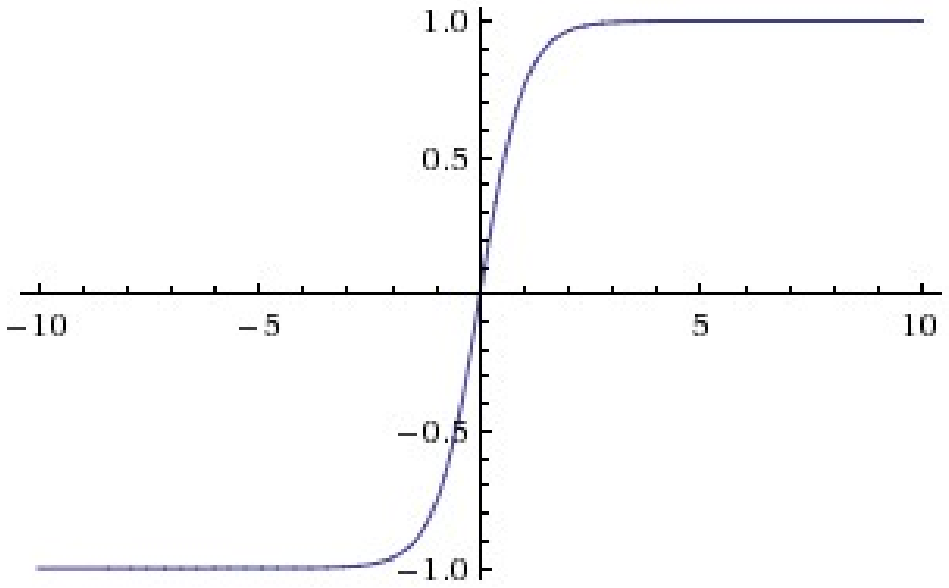
- 容易饱和。当输入很大/很小时（saturation, 饱和），神经元的梯度接近 0，出现“梯度消失”（gradient vanishing），导致无法完成深层网络的训练。
- 输出不是零均值的（not zero-centered）。假设某个神经元的输入一直是正的，即 $x > 0$. 对于 $f(x) = w^T x + b$ ，则 w 获得的梯度将是恒正或者恒负（取决于 f 得到的梯度的符号），导致 w 的更新非常“曲折”（zig-zagging）。当然，如果是按 batch 训练，最终梯度是各个样本下梯度的和，而每个样本下的梯度可能是符号各异的，因此在一定程度上可以缓解这个问题。

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：



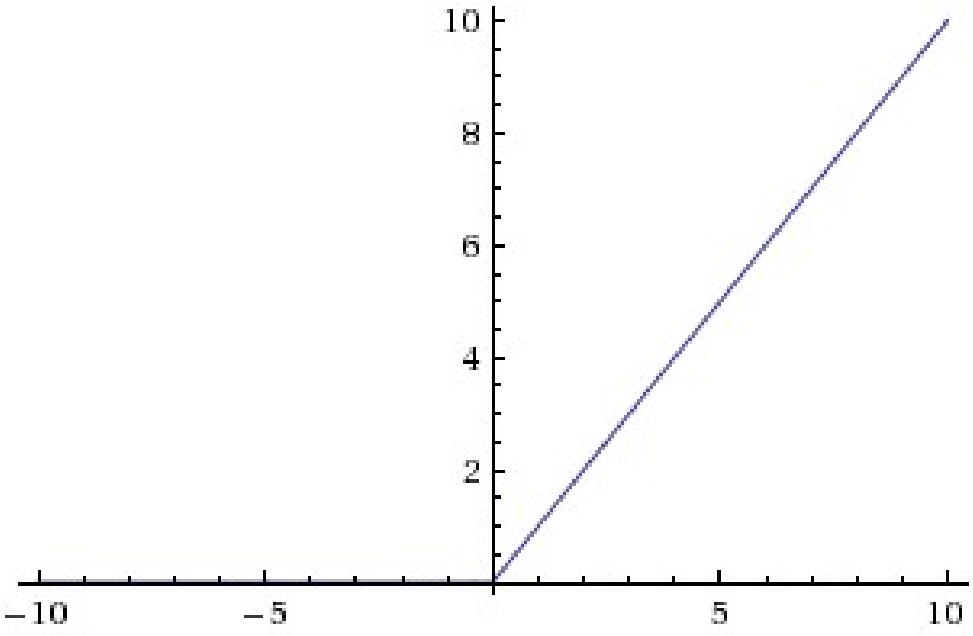
与 **sigmoid** 一样，**tanh** 也会产生饱和现象，但是 **tanh** 的输出是零均值的（zero-centered）。

6.2.3 ReLU

ReLU 函数的数学表达式如下：

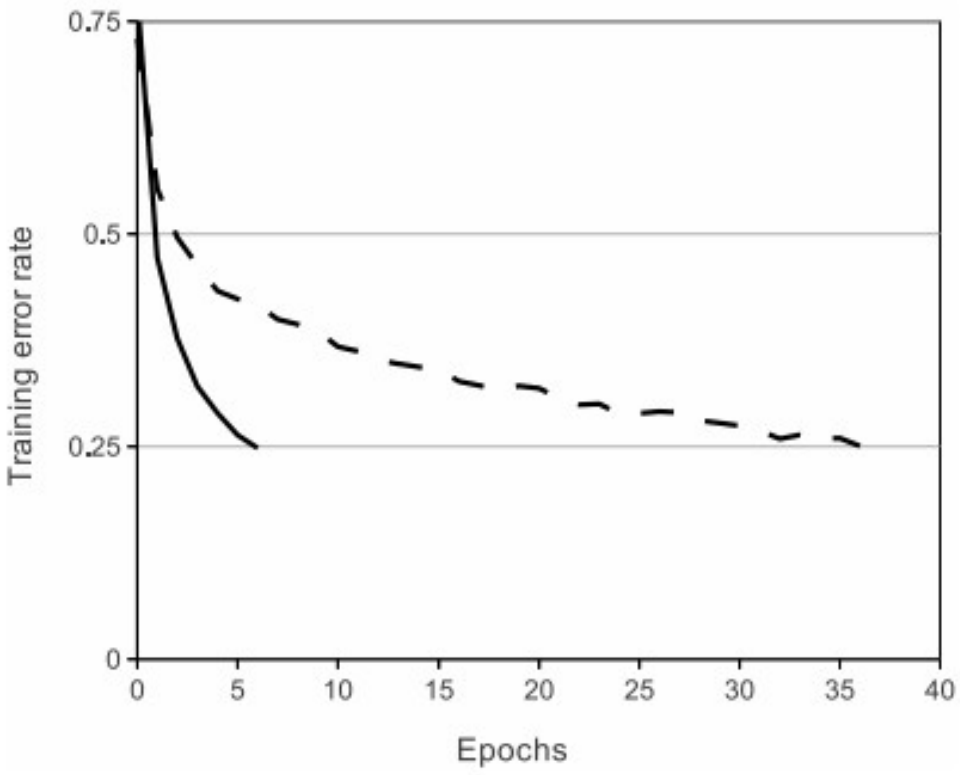
$$relu(z) = max(0, z).$$

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

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

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

6.3.1 加速训练

- **增大学习率**。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。
- **移除 dropout** 。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（BN 提供了正则化的作用）。
- **减小 L_2 正则化损失的权重**。
- **加速学习率衰减**。

6.3.2 BN 消除

如果网络发现这种 normalization 是多余的，可以通过学习使得：

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

$$\beta^{(k)} = E[x^{(k)}]$$

从而消除 BN 的作用。

6.3.3 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。

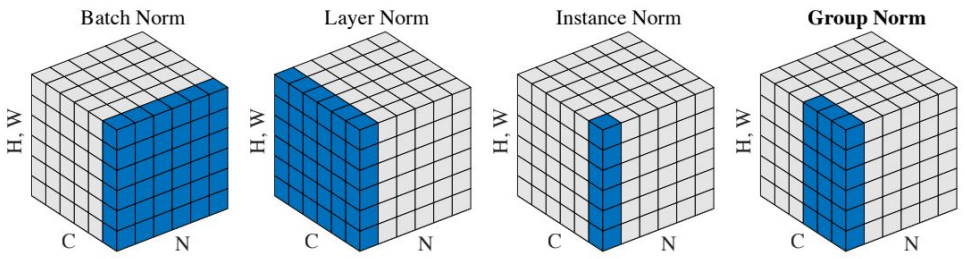


Figure 2. Normalization methods. Each subplot shows a feature map tensor. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels. Group Norm is illustrated using a group number of 2.

6.3.4 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

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。

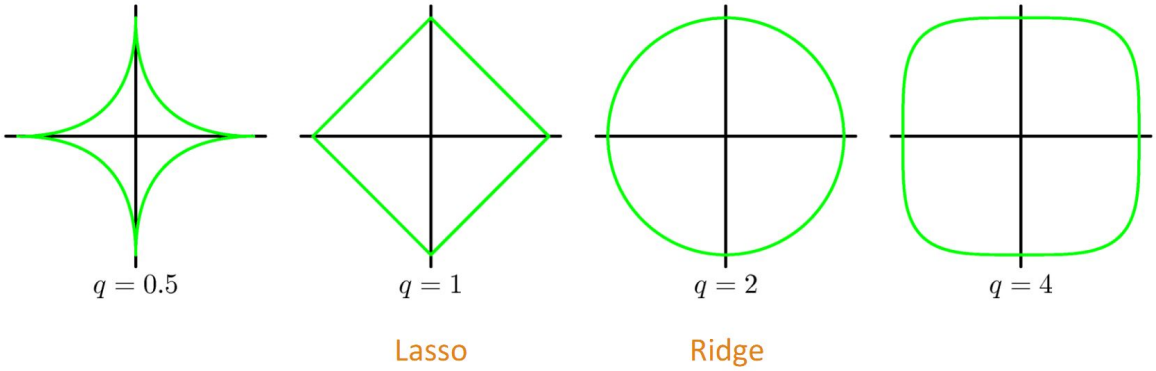
5. 集成学习。Bagging：并行化模型生成，减小模型 variance。Boosting：串行化模型生成，减小模型 bias。

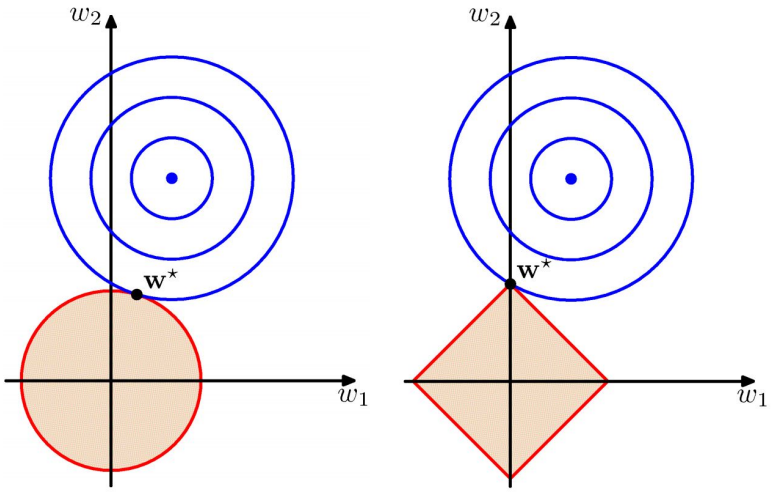
6. 正则化。

- L0 正则化（非 0 元素个数），难以优化求解（NP-Hard）。
- 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。
- L2 正则化（元素平方和，Ridge regression / weight dacay），使权值分布均匀且值较小。

6.4.4 附：正则化

$$L_q\text{-norm}: \ \|w\|_q^q = \sum_j |w_j|^q.$$





6.5 pytorch：模型保存与读取

6.5.1 简单

```
import torch
## save
torch.save(model, 'model.pkl')
## 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，一般存取的 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。

```
import torch
import shutil

## save
def save_checkpoint(state, is_best, save_path, filename):
  filename = os.path.join(save_path, filename)
  torch.save(state, filename)
  if is_best:
    bestname = os.path.join(save_path, 'model_best.pth.tar')
    shutil.copyfile(filename, bestname)

```

(continues on next page)

(continued from previous page)

```
save_checkpoint({
        'epoch': cur_epoch,
        'state_dict': model.state_dict(),
        'best_prec': best_prec,
        'loss_train': loss_train,
        'optimizer': optimizer.state_dict(),
      }, is_best, save_path, 'epoch-{}_checkpoint.pth.tar'.format(cur_epoch))

## load
def load_checkpoint(checkpoint, model, optimizer):
  """ loads state into model and optimizer and returns:
      epoch, best_precision, loss_train[]
      e.g., model = alexnet(pretrained=False)
  """
  if os.path.isfile(load_path):
      print("=> loading checkpoint '{}'".format(load_path))
      checkpoint = torch.load(load_path)
      epoch = checkpoint['epoch']
      best_prec = checkpoint['best_prec']
      loss_train = checkpoint['loss_train']
      model.load_state_dict(checkpoint['state_dict'])
      optimizer.load_state_dict(checkpoint['optimizer'])
      print("=> loaded checkpoint '{}' (epoch {})"
            .format(epoch, checkpoint['epoch']))
      return epoch, best_prec, loss_train
  else:
      print("=> no checkpoint found at '{}'".format(load_path))
      # epoch, best_precision, loss_train
      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：

```
# args has the model name, num classes and other irrelevant stuff
pretrained_state = model_zoo.load_url(model_names[args.arch])
model_state = my_model.state_dict()
pretrained_state = { k:v for k,v in pretrained_state.iteritems() if k in model_state and v.size()␣
↪== model_state[k].size() }
model_state.update(pretrained_state)
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

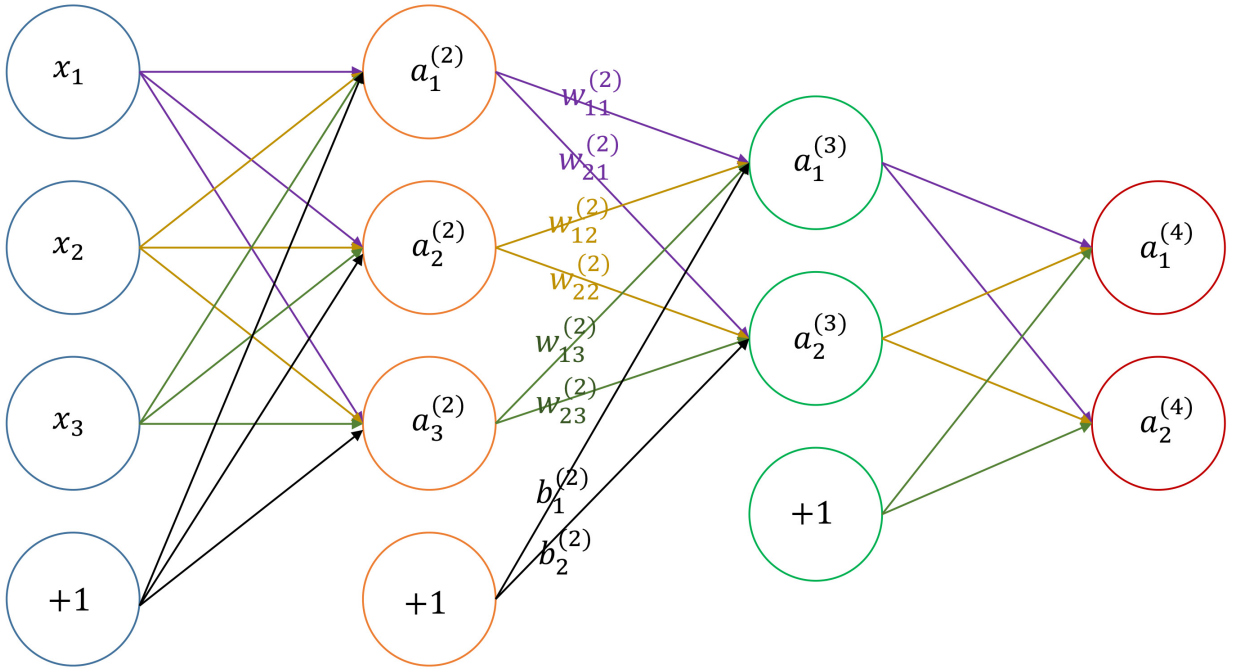
   https://www.cnblogs.com/darkknightzh/p/6836568.html

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



l_1层（输入） l_2层 l_3层 l_4层（输出）

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

$$z_i^{(l+1)} = b_i^{(l)} + \sum_{j=1}^{C_l} w_{ij}^{(l)} a_j^{(l)},$$

$$g(t) = \frac{1}{1+e^{-t}},$$

$$g'(t) = (1-g(t))g(t),$$

$$a_i^{(l)} = g(z_i^{(l)}).$$

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：

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

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：

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

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）：

$$\frac{\partial \mathcal{L}}{\partial z_i^{(n)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(n)}} \frac{\partial a_i^{(n)}}{\partial z_i^{(n)}}$$

$$\frac{\partial \mathcal{L}}{\partial a_i^{(n)}} = \sum_{j=1}^{C_n} \frac{\partial \mathcal{L}}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial a_i^{(n)}}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_j} = -\frac{y_j}{\hat{y}_j}$$

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：

$$\delta_i^{(n)} = \frac{\partial \mathcal{L}}{\partial z_i^{(n)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(n)}} \frac{\partial a_i^{(n)}}{\partial z_i^{(n)}} = (-y_i + \hat{y}_i) g'(z_i^{(n)})$$

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

2. 神经网络–反向传播详细推导过程

   https://blog.csdn.net/qq_29762941/article/details/80343185

6.8 优化算法

6.8.1 可视化

等高线

鞍点

6.8.2 SGD

这里 **SGD** 指小批量梯度下降算法。

小批量损失：$\mathcal{L}$ ，学习率：η ，Hadamard 积：$\odot$ 。

$$g \leftarrow \nabla_\theta \mathcal{L}(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \text{ [计算梯度]}$$

$$\theta \leftarrow \theta - \eta g \text{ [参数更新]}$$

特点

- 相比于单样本 SGD，可以降低参数更新时的方差，收敛更稳定；可以充分地利用深度学习库中高度优化的矩阵操作来进行更有效的梯度计算
- 不能保证很好的收敛性：如果选择的太小，收敛速度会很慢；如果太大，损失函数就会在极小值处不停地震荡甚至偏离；容易困在鞍点。
- 选择合适的学习率比较困难：对所有的参数更新使用同样的学习率。对于稀疏数据或者特征，有时我们可能想更快更新对应于不经常出现的特征的参数，对于常出现的特征更新慢一些。

6.8.3 Momentum

$$
\begin{aligned}
v \leftarrow \\
\gamma v - \eta \nabla_\theta \mathcal{L}(\theta) \text{ [速度更新]} \\
\theta \leftarrow \\
\theta + v \text{ [参数更新]}
\end{aligned}
$$

动量（momentum）方法旨在加速学习，特别是处理高曲率、小但一致的梯度，或是带噪声的梯度。动量算法积累了之前梯度指数级衰减的移动平均，并且沿该方向继续移动。当许多连续的梯度指向相同的方向时，步长最大。

特点

- 下降初期，使用上一次参数更新；下降方向一致，乘上较大的 γ 能够进行很好的加速。
- 下降中后期，在局部最小值来回震荡的时候，梯度接近 0，γ 能够使得更新幅度增大，跳出陷阱。
- 梯度改变方向时，能够减少更新。

6.8.4 Adagrad

$$
\begin{aligned}
g \leftarrow \\
\nabla_\theta \mathcal{L}(\theta) \text{ [计算梯度]} \\
r \leftarrow \\
r + g \odot g \text{ [累计平方梯度]} \\
\Delta\theta \leftarrow \\
-\frac{\eta}{\sqrt{r+\epsilon}} \odot g \text{ [梯度除以累计平方梯度的平方根]} \\
\theta \leftarrow \\
\theta + \Delta\theta \text{ [参数更新]}
\end{aligned}
$$

特点

- 独立地适应所有模型参数的学习率，适合处理稀疏数据。对于梯度 g 较大的参数（这些参数关联着频繁出现的特征），有一个快速下降的学习率；对于梯度 g 较小的参数（这些参数关联着不频繁出现的特征），学习率有相对较小的下降。
- 从训练开始累积平方梯度，导致有效学习率过早和过量减小，导致训练过早停止。

6.8.5 Adadelta

$$
\begin{aligned}
g &\leftarrow \\
\nabla_\theta \mathcal{L}(\theta) &\text{ [计算梯度]} \\
E[g^2] &\leftarrow \\
\gamma E[g^2] + (1-\gamma) g \odot g &\text{ [累计平方梯度：指数衰减平均]} \\
RMS[g] &\leftarrow \\
\sqrt{E[g^2] + \epsilon} &\text{ [梯度均方根]} \\
E[\Delta\theta^2] &\leftarrow \\
\gamma E[\Delta\theta^2] + (1-\gamma) \Delta\theta \odot \Delta\theta &\text{ [平方参数增量平滑]} \\
RMS[\Delta\theta] &\leftarrow \\
\sqrt{E[\Delta\theta^2] + \epsilon} &\text{ [参数增量均方根]} \\
\Delta\theta &\leftarrow \\
-\frac{RMS[\Delta\theta]}{RMS[g]} \odot g &\text{ [参数增量]} \\
\theta &\leftarrow \\
\theta + \Delta\theta &\text{ [参数更新]}
\end{aligned}
$$

Adadelta 是 Adagrad 的改进。

特点

- 使用指数衰减平均值，使得能够在找到凸碗状结构后快速收敛。
- 不用依赖于全局学习率，然而引入了新的超参：衰减系数 γ 。
- 训练初中期，加速效果很快。

6.8.6 RMSprop

$$
\begin{aligned}
g &\leftarrow \\
&\mathcal{L}(\theta) \text{ [计算梯度]} \\
r &\leftarrow \\
&\gamma r + (1-\gamma) g \odot g \text{ [累计平方梯度：指数衰减平均]} \\
\Delta\theta &\leftarrow \\
&-\frac{\eta}{\sqrt{r+\epsilon}} \odot g \text{ [参数增量]} \\
\theta &\leftarrow \\
&\theta + \Delta\theta \text{ [参数更新]}
\end{aligned}
$$

RMSprop 趋于 Adagrad 和 Adadelta 之间。

特点

- 使用指数衰减平均值，使得能够在找到凸碗状结构后快速收敛。
- 仍然依赖于全局学习率。

6.8.7 Adam

$$
\begin{aligned}
g \leftarrow & \\
\mathcal{L}(\theta) & \text{ [计算梯度]} \\
t \leftarrow & \\
t+1 & \text{ [迭代次数]} \\
m \leftarrow & \\
\beta_1 m + (1-\beta_1) g & \text{ [有偏一阶矩]} \\
n \leftarrow & \\
\beta_1 n + (1-\beta_2) g \odot g & \text{ [有偏二阶矩]} \\
\hat{m} \leftarrow & \\
\frac{m}{1-\beta_1^t} & \text{ [修正一阶矩]} \\
\hat{n} \leftarrow & \\
\frac{n}{1-\beta_2^t} & \text{ [修正二阶矩]} \\
\Delta\theta \leftarrow & \\
-\eta \frac{\hat{m}}{\sqrt{\hat{n}+\epsilon}} \odot g & \text{ [参数增量]} \\
\theta \leftarrow & \\
\theta + \Delta\theta & \text{ [参数更新]}
\end{aligned}
$$

相当于 RMSprop + Momentum。

特点

- 结合了 Adagrad 善于处理稀疏梯度（不同的参数计算不同的自适应学习率）和 RMSprop 善于处理非平稳目标的优点。
- 经过矩修正后，每一次迭代的学习率都有确定范围，使得参数更新比较平稳。

6.8.8 参考资料

1. An overview of gradient descent optimization algorithms

   http://ruder.io/optimizing-gradient-descent/

2. 深度学习——优化器算法 Optimizer 详解

   https://cloud.tencent.com/developer/article/1118673

3. 深度学习——优化器算法 Optimizer 详解

   https://www.cnblogs.com/guoyaohua/p/8542554.html

4. An overview of gradient descent optimization algorithms

   https://arxiv.org/pdf/1609.04747.pdf

6.9 pytorch：add_module

6.9.1 add_module

如果有一个元素是 Module 的列表，直接赋值给一个模型的属性（attribute），并不会让给列表内的 Modules 立即成为模型的模块。

```
import torch
import torch.nn as nn

class A(nn.Module):
    def __init__(self):
        super(A, self).__init__()

        self.layerList = [nn.Linear(5, 1, bias=False), nn.Linear(2,1, bias=False)]
```

```
>>> a = A()
>>> print a
A(
)
```

简单地，可以使用 nn.Sequential(*module_list) 将列表转换成 **串联**的模块。一方面，这样会使得模块的名字被默认地用数字命名；另一方面，如果需要的不是串联结构，这样的做法行不通。

使用 add_module 可以自由地将列表的元素变成模型的模块。add_module 建立了对 Module 的引用，并不是添加了新的对象。因此，对引用的修改会直接修改列表内的 Module 。

加入之后，可以通过模型的名字来进行访问：_modules[name] 。_modules 是一个顺序字典（OrderedDict）。

```
import torch
import torch.nn as nn

class A(nn.Module):
    def __init__(self):
        super(A, self).__init__()
        self.layerList = [nn.Linear(5, 1, bias=False), nn.Linear(2,1, bias=False)]
        self.add_module("layer_0", self.layerList[0])
        self.add_module("layer_1", self.layerList[1])

        print self.layerList[0].weight
```

(continues on next page)

(continued from previous page)

```
        print self._modules['layer_0'].weight
        self._modules['layer_0'].weight.data = self._modules['layer_0'].weight.data + 2 * torch.
↪ones_like(self._modules['layer_0'].weight.data)

    def forward(self):
        print self.layerList[0].weight
        print self._modules['layer_0'].weight
```

```
>>> a = A() ## init
Parameter containing:
 0.0244 -0.0521 -0.4013 -0.1229  0.0343
[torch.FloatTensor of size 1x5]

Parameter containing:
 0.0244 -0.0521 -0.4013 -0.1229  0.0343
[torch.FloatTensor of size 1x5]

>>> print a
A(
  (layer_0): Linear(in_features=5, out_features=1, bias=False)
  (layer_1): Linear(in_features=2, out_features=1, bias=False)
)

>>> a() ## forward
Parameter containing:
 2.0244  1.9479  1.5987  1.8771  2.0343
[torch.FloatTensor of size 1x5]

Parameter containing:
 2.0244  1.9479  1.5987  1.8771  2.0343
[torch.FloatTensor of size 1x5]
## 可以看到，上面的参数是同步更新的
```

6.9.2 attribute 索引

除了使用 `_modules[name]` 访问模块，还可以将 name 转换成属性（attribute）的索引，通过下标的形式访问。

```
import torch
import torch.nn as nn

class AttrProxy(object):
```

(continues on next page)

(continued from previous page)

```
    """Translates index lookups into attribute lookups."""
    def __init__(self, module, prefix):
        self.module = module
        self.prefix = prefix
    def __getitem__(self, index):
        return getattr(self.module, self.prefix + str(index))

class A(nn.Module):
    def __init__(self):
        super(A, self).__init__()
        self.layerList = [nn.Linear(5, 1, bias=False), nn.Linear(2,1, bias=False)]
        self.add_module("layer_0", self.layerList[0])
        self.add_module("layer_1", self.layerList[1])

        self.layer = AttrProxy(self, "layer_")

        print self.layerList[0].weight
        print self.layer[0].weight
        self.layer[0].weight.data = self.layer[0].weight.data + 2 * torch.ones_like(self.layer[0].
↪weight.data)

    def forward(self):
        print self.layerList[0].weight
        print self.layer[0].weight
        print self.layer[1].weight
```

```
>>> a = A() ## init
Parameter containing:
-0.2655  0.1539 -0.2107  0.0740  0.1922
[torch.FloatTensor of size 1x5]

Parameter containing:
-0.2655  0.1539 -0.2107  0.0740  0.1922
[torch.FloatTensor of size 1x5]

>>> a() ## forward
Parameter containing:
 1.7345  2.1539  1.7893  2.0740  2.1922
[torch.FloatTensor of size 1x5]

Parameter containing:
 1.7345  2.1539  1.7893  2.0740  2.1922
[torch.FloatTensor of size 1x5]
```

(continues on next page)

(continued from previous page)

```
18
19 Parameter containing:
20  0.0068 -0.1787
21 [torch.FloatTensor of size 1x2]
```

6.9.3 参考资料

1. pytorch documentation

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

2. List of nn.Module in a nn.Module

   https://discuss.pytorch.org/t/list-of-nn-module-in-a-nn-module/219

CHAPTER

SEVEN

数理与算法

7.1 算法复杂度与主定理

递归算法的复杂性分析方法：代入法、递归树、主定理。这里只讨论主定理方法。

主定理方法应用于如下的递归形式：

$$T(n) = aT(\frac{n}{b}) + f(n).$$

其中，$a \geqslant 1,\ b \geqslant 1$ ，f 是渐进正的。

7.1.1 渐进分析

假设对于所有 n ，满足 $f(n) \geqslant 0,\ g(n) \geqslant 0$ 。

- 渐进上界 $\mathcal{O}$

$$\mathcal{O}(g(n)) = \{f(n) | \text{存在正常数} c \text{和} n_0 \text{使得对所有} n \geqslant n_0 \text{有：}\ 0 \leqslant f(n) \leqslant cg(n)\}$$

- 渐进下界 Ω

$$\Omega(g(n)) = \{f(n) | \text{存在正常数} c \text{和} n_0 \text{使得对所有} n \geqslant n_0 \text{有：}\ 0 \leqslant cg(n) \leqslant f(n)\}$$

- 渐进近界 Θ

$$\Theta(g(n)) = \{f(n) | \text{存在正常数} c_1,\ c_2 \text{和} n_0 \text{使得对所有} n \geqslant n_0 \text{有：}\ c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\}$$

定理：

$$\Theta(g(n)) = \mathcal{O}(g(n)) \cap \Omega(g(n))$$

7.1.2 主定理

需要比较 $f(n)$ 和 $n^{\log_b a}$ 。

- $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ ： $f(n)$ 的增长速度比 $n^{\log_b a}$ 慢一个 n^{ϵ} 因子。

$$T(n) = \Theta(n^{\log_b a})$$

- $f(n) = \Theta(n^{\log_b a} \log^k n)$ ： $f(n)$ 与 $n^{\log_b a} \log^k n$ 具有相似的增长速度。

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

- $f(n) = \Omega(n^{\log_b a + \epsilon})$ ： $f(n)$ 的增长速度比 $n^{\log_b a}$ 快一个 n^{ϵ} 因子，且满足 $af(\frac{n}{b}) \leqslant cf(n),\ 0 < c < 1$ 。

$$T(n) = \Theta(f(n))$$

例子：

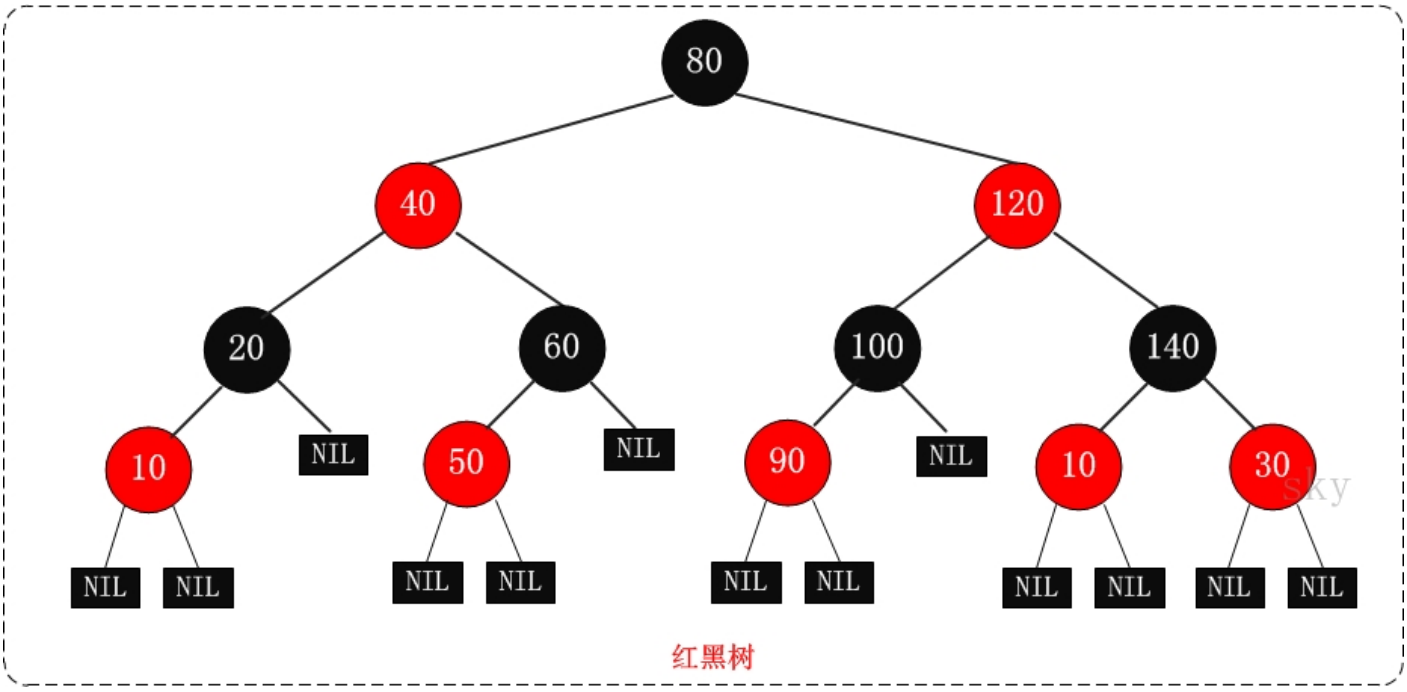
$$T(n) = 4T(\frac{n}{2}) + n \rightarrow$$
$$\epsilon = 1,\ T(n) = \Theta(n^2)$$
$$T(n) = 4T(\frac{n}{2}) + n^2 \rightarrow$$
$$k = 0,\ T(n) = \Theta(n^2 \log n)$$
$$T(n) = 4T(\frac{n}{2}) + n^3 \rightarrow$$
$$\epsilon = 1,\ c = \frac{1}{2},\ T(n) = \Theta(n^3)$$

7.2 红黑树

R-B Tree，全称是 Red-Black Tree，又称为 “红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红（Red）或黑（Black）。

红黑树的特性

- 每个节点或者是黑色，或者是红色。
- 根节点是黑色。
- 每个叶子节点（NIL）是黑色。[注意：这里的叶子节点是指为空 (NIL 或 NULL) 的叶子节点！]
- 如果一个节点是红色的，则它的子节点必须是黑色的。
- 从一个节点到该节点的子孙叶子节点的所有路径上包含相同数目的黑节点（black-height）。



红黑树的应用比较广泛，主要是用它来存储有序的数据，查找、插入、删除操作的时间复杂度是 $\mathcal{O}(\log n)$ ，效率非常之高。C++ STL 中的 set、map，以及 Linux 虚拟内存的管理，都是通过红黑树去实现的。

定理：一颗具有 n 个键值的红黑树，其高度 h 满足：

$$h \leqslant 2\log(n+1).$$

另外，AVL 树是最早被发明的自平衡二叉查找树。在 AVL 树中，任一节点对应的两棵子树的最大高度差为 1，因此它也被称为高度平衡树。查找、插入和删除在平均和最坏情况下的时间复杂度都是 $\mathcal{O}(\log n)$ 。

7.2.1 参考资料

1. 红黑树 (一) 之原理和算法详细介绍

   https://www.cnblogs.com/skywang12345/p/3245399.html

2. 30 张图彻底理解红黑树

   http://developer.51cto.com/art/201901/590926.htm

7.3 最短路径

7.3.1 Bellman-Ford 算法

时间复杂度 $\mathcal{O}(VE)$ 其中顶点数 V ，边数 E 。如果不存在负圈（一条回路的代价和为负），那么每一条最短路径都不会经过同一个顶点两次，因此 while 循环最多执行 $V-1$ 次。

```
struct edge {int from, to, cost;};

edge es[MAX_E];

int d[MAX_V]; // 最短距离
int V, E; // 顶点数，边数

// 从顶点 s 出发的最短距离（假设不存在负圈）
void shortest_path(int s)
{
  fill(d, d+V, INF);
  d[s] = 0;
  while(true)
  {
    bool update = false;
    for(int i = 0; i < E; ++i)
    {
      edge e = es[i];
      if(d[e.from] != INF && d[e.to] > d[e.from] + e.cost)
      {
        d[e.to] = d[e.from] + e.cost;
        update = true;
      }
    }
    if(!update) break;
  }
}
```

检查负圈：如果第 V 次循环还有更新，则表明存在负圈，返回 true。

```
bool find_negative_loop()
{
  fill(d, d+V, 0); // 初始化为 0，防止因为是 d[e.from] == INF 而停止更新
  for(int i = 0; i < V; ++i)
  {
    for(int j = 0; j < E; ++j)
    {
```

(continues on next page)

(continued from previous page)

```
      edge e = es[j];
      if(d[e.to] > d[e.from] + e.cost)
      {
        d[e.to] = d[e.from] + e.cost;
        if(i == V-1) return true;
      }
    }
  }
  return false;
}
```

7.3.2 Dijkstra 算法

适合处理没有负边的情形。每一次循环，在尚未确定最短距离的顶点中，d[i] 最小的顶点就是下一个确定的顶点。但是如果存在负边，d[i] 在之后的更新中还会变小，因此算法失效。

- **方法一** 直接使用邻接矩阵，时间复杂度 $\mathcal{O}(V^2)$ 。

```
int cost[MAX_V][MAX_V];
int d[MAX_V];
bool used[MAX_V];
int V;

void dijkstra(int s)
{
  fill(d, d+V, INF);
  d[s] = 0;
  fill(used, used+V, false);

  while(true)
  {
    int v = -1;
    for(int u = 0; u < V; ++u)
    {
      if(!used[u] && (v==-1 || d[u] < d[v])) v = u;
    }

    if(v == -1 || d[v] == INF) break;
    // v == -1 表示所有顶点都找到了最短距离
    // d[v] == INF 表示后面所有的顶点都已经不可达，直接结束循环

    used[v] = true;
    for(int u = 0; u < V; ++u)
```

(continues on next page)

(continued from previous page)

```
    {
      d[u] = min(d[u], d[v] + cost[v][u]);
    }
  }
}
```

- **方法二** 使用最小堆（优先队列），堆中元素个数为 $\mathcal{O}(V)$，出队（弹出最小值）的次数为 $\mathcal{O}(E)$，时间复杂度 $\mathcal{O}(E \log V)$。

```
struct edge {int to, cost;};
typedef pair<int, int> P; // first: 最短距离, second: 顶点

int V;
vector<edge> G[MAX_V]; // 边
int d[MAX_V];

void dijkstra(int s)
{
  priority_queue<P, vector<P>, greater<P>> que;

  fill(d, d+V, INF);
  d[s] = 0;

  que.push(P(0, s));
  while(!que.empty())
  {
    P p = que.top();
    que.pop();

    int v = p.second;
    if(d[v] < p.first) continue;

    for(int i = 0; i < G[v].size(); ++ i)
    {
      edge e = G[v][i];
      if(d[e.to] > d[v] + e.cost)
      {
        d[e.to] = d[v] + e.cost;
        que.push(P(d[e.to], e.to));
      }
    }
  }
}
```

7.3.3 实例

- 耗时最短的路径：某些顶点有自行车，骑上自行车之后耗时减半。Hint：广度优先遍历，使用优先队列/堆，最早到达终点的一定是耗时最短路径。这里需要设置两个全局数组，一个记录当前顶点有自行车的最短耗时，另一个记录当前顶点没有自行车的最短耗时。

  https://www.nowcoder.com/practice/7689b595f3eb419b9e7816c4f45a400d?tpId=90&tqId=30852&tPage=4&rp=4&ru=/ta/2018test&qru=/ta/2018test/question-ranking

  Code

```
import sys
import heapq as hq

n, m = map(int, sys.stdin.readline().strip().split())
edges = [[] for _ in range(n)]
for _ in range(m):
    begin, end, cost = map(int, sys.stdin.readline().strip().split())
    begin -= 1
    end -= 1
    edges[begin].append((end, cost)) ## 无向边
    edges[end].append((begin, cost))
have_bike = [False for _ in range(n)]
k = int(sys.stdin.readline().strip())
for _ in range(k):
    v = int(sys.stdin.readline().strip())
    v -= 1
    have_bike[v] = True

INF = float('inf') ## 无穷大
## 根据当前顶点是否有自行车，需要定义两个全局数组，存储当前最短耗时
global_cost = {False: [INF for _ in range(n)], True: [INF for _ in range(n)]}
global_cost[have_bike[0]][0] = 0
ans = -1
h = []
## 堆元素：(cost, v, have_bike)
hq.heappush(h, (0, 0, have_bike[0]))
while len(h) > 0:
    v_cost, v, v_bike = hq.heappop(h)
    if v == n-1:
        ans = v_cost
        break
    for u, uv_cost in edges[v]:
        if v_bike:
            uv_cost /= 2
```

(continues on next page)

(continued from previous page)

```
        u_cost = v_cost + uv_cost
        u_bike = have_bike[u] or v_bike

        if u_cost >= global_cost[u_bike][u]:
            continue
        global_cost[u_bike][u] = u_cost
        hq.heappush(h, (u_cost, u, u_bike))

print ans
```

7.4 二叉树遍历

7.4.1 定义

```
// Definition for a binary tree node.
struct TreeNode
{
   int val;
   TreeNode *left;
   TreeNode *right;
   TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

7.4.2 先序遍历

- 递归

```
void preOrder_Recur(TreeNode* T)
{
  if(!T) return;
  else
  {
    visite(T -> val);
    preOrder_Recur(T -> left);
    preOrder_Recur(T -> right);
  }
}
```

- 非递归

```
void preOrder_NonRecur(TreeNode* T)
{
  stack<TreeNode*> stk;
  while(T || !stk.empty())
  {
    while(T)
    {
      visite(T -> val);
      stk.push(T);
      T = T -> left;
    }
    if(! stk.empty())
    {
      T = stk.top();
      stk.pop();
      T = T -> right;
    }
  }
}
```

7.4.3 中序遍历

- 递归

```
void inOrder_Recur(TreeNode* T)
{
  if(!T) return;
  else
  {
    inOrder_Recur(T -> left);
    visite(T -> val);
    inOrder_Recur(T -> right);
  }
}
```

- 非递归

```
void inOrder_NonRecur(TreeNode* T)
{
  stack<TreeNode*> stk;
  while(T || !stk.empty())
  {
    while(T)
```

(continues on next page)

(continued from previous page)

```
    {
      stk.push(T);
      T = T -> left;
    }
    if(! stk.empty())
    {
      T = stk.top();
      stk.pop();
      visite(T -> val);
      T = T -> right;
    }
  }
}
```

7.4.4 后序遍历

- 递归

```
void postOrder_Recur(TreeNode* T)
{
  if(!T) return;
  else
  {
    postOrder_Recur(T -> left);
    postOrder_Recur(T -> right);
    visite(T -> val);
  }
}
```

- 非递归
  - 方法一：后序遍历顺序是：left - right - root；先序遍历顺序是：root - left - right。采用先序遍历的方式，用栈来存储节点（FILO），得到的是按 root - right - left 顺序遍历的临时结果；把临时结果逆序输出，就是后序遍历的结果。

```
vector<int> postOrder_NonRecur(TreeNode* T)
{
  vector<int> res;
  stack<TreeNode*> nodePtr;
  if(T) nodePtr.push(T);
  while(! nodePtr.empty())
  {
    T = nodePtr.top();
```

(continues on next page)

(continued from previous page)

```
    nodePtr.pop();

    res.push_back(T -> val);
    if(T -> left) nodePtr.push(T -> left);
    if(T -> right) nodePtr.push(T -> right);
  }
  reverse(res.begin(), res.end());
  return res;
}
```

- 方法二：一个节点如果不存在右子树，则遍历完左子树之后可以直接访问该节点的值；如果存在右子树，用一个额外的栈（inNode）来临时保存该节点。访问完该节点的右子树之后，就从栈弹出该节点进行访问。

```
vector<int> postOrder_NonRecur(TreeNode* T)
{
  vector<int> res;
  stack<TreeNode*> nodePtr;
  stack<TreeNode*> inNode;
  while(T || ! nodePtr.empty())
  {
      while(T)
      {
          nodePtr.push(T);
          T = T -> left;
      }
      T = nodePtr.top();
      nodePtr.pop();

      if(T -> right)
      {
          inNode.push(T);
          T = T -> right;
      }
      else
      {
          res.push_back(T -> val);
          while(!inNode.empty() && T == inNode.top() -> right)
          // 访问完节点的右子树之后，就从栈弹出该节点进行访问
          {
              res.push_back(inNode.top() -> val);
              T = inNode.top();
              inNode.pop();
```

(continues on next page)

(continued from previous page)

```
            }
            T = NULL;
        }
    }
    return res;
}
```

7.4.5 层次遍历

```
void layerTraversal(TreeNode* T)
{
    queue<TreeNode*> Q;
    if(T) Q.push(T);
    while(!Q.empty())
    {
        T = Q.front();
        Q.pop();
        visite(T -> val);
        if(T -> left) Q.push(T -> left);
        if(T -> right) Q.push(T -> right);
    }
}
```

7.4.6 参考资料

1. 二叉树后序遍历非递归的三种写法 (数据结构)

   https://www.cnblogs.com/demian/p/8117888.html

7.5 游戏与必胜策略

7.5.1 硬币游戏

描述：有 x 枚硬币，A 和 B 两个人轮流取，每次所取的硬币数量要在 $a_1, a_2, ..., a_k$ 当中（其中包含 1 ）。A 先取，取走最后一枚硬币的一方获胜。当双方都采取最优策略，谁会获胜?

策略：动态规划。考虑轮到 A 时，还剩下 j 枚硬币。当 $j = 0$ ，A 必败；如果存在 a_i ，使得 $j - a_i$ 是必败态，则 j 就是必胜态；如果对于所有的 a_i ，$1 \leqslant i \leqslant k$ ，使得 $j - a_i$ 都是必胜态，则 j 是必败态。

```
int X, K, A[MAX_K];

bool win[MAX_X + 1];

void solve()
{
  win[0] = false;
  for(int j = 1; j <= X; ++j)
  {
    win[j] = false;
    for(int i = 0; i < K; ++i)
    {
      win[j] = win[j] | (A[i]<=j && !win[j-A[i]]);
    }
  }
}
```

7.5.2 Nim 游戏

描述：有 n 堆石子，每堆 a_i 颗石子。A 和 B 两个人轮流取，每次从石子堆中至少取走一颗。A 先取，最后取光所有石子的一方获胜。当双方都采取最优策略，谁会获胜？

策略：$a_1 \oplus a_2 \oplus ... \oplus a_n \neq 0$ （异或运算），则 A 必胜；$a_1 \oplus a_2 \oplus ... \oplus a_n = 0$ ，则 A 必败。

7.5.3 Grundy 数

描述：有 n 堆硬币，每堆 x_i 枚硬币。A 和 B 两个人轮流取，每次所取的硬币数量要在 $a_1, a_2, ..., a_k$ 当中（其中包含 1 ）。A 先取，取走最后一枚硬币的一方获胜。当双方都采取最优策略，谁会获胜？

策略：转换成 Nim，$grundy(x_1) \oplus grundy(x_2) \oplus ... \oplus grundy(x_n) \neq 0$ 则 A 必胜，否则必败。当前状态的 grundy 值表示：从该状态出发，一步可达状态的 grundy 值的集合之外的最小非负整数。

```
int N, K, X[MAX_N], A[MAX_K];

int grundy[MAX_X + 1]; // 全局数组，初始化为 0

void solve()
{
  grundy[0] = 0;

  int max_x = *max_element(X, X+N);
  for(int j = 0; j <= max_x; ++j)
  {
```

(continues on next page)

(continued from previous page)

```
    set<int> s;
    for(int i = 0; i < K; ++i)
    {
      if(A[i] < j) s.insert(grundy[j - A[i]]); // 一步可达状态的 grundy 值
    }
    int g = 0; // 集合之外的最小非负整数
    while(s.count(g) != 0) g++;
    grundy[j] = g;
  }

  int res = 0;
  for(int n = 0; n < N; ++n) res ^= grundy[X[n]];
  if(res != 0) cout << "A wins." << endl;
  else cout << "B wins." << endl;
}
```

7.6 蓄水池抽样

问题描述：随机从一个数据流中选取 1 个或 k 个数，保证每个数被选中的概率是相同的。数据流的长度 n 未知或者是非常大。

7.6.1 随机选择 1 个数

在数据流中，依次以概率 1 选择第一个数，以概率 $\frac{1}{2}$ 选择第二个数（替换已选中的数），…，依此类推，以概率 $\frac{1}{m}$ 选择第 m 个数（替换已选中的数）。结束时（遍历完了整个数据流），每个数被选中的概率都是 $\frac{1}{n}$ 。证明:

```
第 m 个对象最终被选中的概率 = 选择第 m 个数的概率 x 后续所有数都不被选择的概率
```

即

$$P = \frac{1}{m} \times \left(\frac{m}{m+1} \times \frac{m+1}{m+2} \times \cdots \times \frac{n-1}{n} \right) = \frac{1}{n}.$$

```
#include <iostream>
#include <vector>
#include <utility> // swap
#include <ctime>
#include <cstdlib> // rand, srand
using namespace std;

```

(continues on next page)

(continued from previous page)

```cpp
typedef vector<int> VecInt;
typedef VecInt::iterator Itr;
typedef VecInt::const_iterator CItr;

// 等概率产生区间 [a, b] 之间的随机数
int RandInt(int a, int b)
{
  if (a > b) swap(a, b);
  return a + rand() % (b - a + 1);
}

bool Sample(const VecInt data, int &result)
{
  if (data.size() <= 0) return false;

  //srand(time(nullptr)); // 设置随机 seed

  CItr it = data.begin();
  result = *it;
  int m;
  for (m = 1, it = data.begin() + 1; it != data.end(); ++m, ++it)
  {
    int ri = RandInt(0, m); // ri < 1 的概率为 1/(m+1)
    if (ri < 1) result = *it;
  }
  return true;
}
```

7.6.2 随机选择 k 个数

在数据流中，先把读到的前 k 个数放入“池”中，然后依次以概率 $\frac{k}{k+1}$ 选择第 k+1 个数，以概率 $\frac{k}{k+2}$ 选择第 k+2 个数，…；以概率 $\frac{k}{m}$ 选择第 m 个数（m > k）。如果某个数被选中，则 **随机替换** “池”中的一个数。最终每个数被选中的概率都为 $\frac{k}{n}$ 。证明:

第 m 个对象最终被选中的概率 = 选择第 m 个数的概率 x (其后元素不被选择的概率 + 其后元素被选择的概率 x 不替换第 m 个数的概率)

即

$$P = \frac{k}{m} \times \left[\left((1 - \frac{k}{m+1}) + \frac{k}{m+1} \times \frac{k-1}{k} \right) \times \left((1 - \frac{k}{m+2}) + \frac{k}{m+2} \times \frac{k-1}{k} \right) \times \cdots \times \left((1 - \frac{k}{n}) + \frac{k}{n} \times \frac{k-1}{k} \right) \right] = \frac{k}{m} \times \frac{m}{m+1} \times \frac{m+1}{m+2} \times \cdots \times \frac{n-1}{n} = \frac{k}{n}.$$

```cpp
#include <iostream>
#include <vector>
#include <utility> // swap
#include <ctime>
#include <cstdlib> // rand, srand
using namespace std;

typedef vector<int> VecInt;
typedef VecInt::iterator Itr;
typedef VecInt::const_iterator CItr;

const int k = 10;
int result[k];

// 等概率产生区间 [a, b] 之间的随机数
int RandInt(int a, int b)
{
  if (a > b) swap(a, b);
  return a + rand() % (b - a + 1);
}

bool Sample(const VecInt data)
{
  if (data.size() < k) return false;

  //srand(time(nullptr)); // 设置随机 seed

  CItr it = data.begin();
  for(int m = 0; m < k; ++m) result[m] = *it++;

```

(continues on next page)

(continued from previous page)

```
  for (int m = k; it != data.end(); ++m, ++it)
  {
    int ri = RandInt(0, m);
    if (ri < k) result[ri] = *it; // ri < k 的概率为 k/(m+1)
  }
  return true;
}
```

7.6.3 参考资料

1. 蓄水池抽样——《编程珠玑》读书笔记

   https://blog.csdn.net/huagong_adu/article/details/7619665

7.7 排序算法

比较排序

- 插入排序
- 选择排序
- 冒泡排序
- 快速排序
- 堆排序
- 归并排序
- 希尔排序

非比较排序

- 计数排序
- 桶排序
- 基数排序

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	排
插入排序	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	In-p
选择排序	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	In-p
冒泡排序	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	In-p
快速排序	$\mathcal{O}(n\log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n\log n)$	$\mathcal{O}(\log n)$	In-p
堆排序	$\mathcal{O}(n\log n)$	$\mathcal{O}(n\log n)$	$\mathcal{O}(n\log n)$	$\mathcal{O}(1)$	In-p
归并排序	$\mathcal{O}(n\log n)$	$\mathcal{O}(n\log n)$	$\mathcal{O}(n\log n)$	$\mathcal{O}(n)$	Out
希尔排序	受实现方式影响，在$\mathcal{O}(n^{1.3})$~$\mathcal{O}(n^2)$之间			$\mathcal{O}(1)$	In-p
计数排序	$\mathcal{O}(n+k)$	$\mathcal{O}(n+k)$	$\mathcal{O}(n+k)$	$\mathcal{O}(k)$	Out
桶排序	$\mathcal{O}(n+n(\log n-\log k))$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n+k)$	Out
基数排序	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(n+k)$	Out

- **稳定性**：键值相同的元素在排序之后仍能保持原来的相对顺序。
- **空间复杂度**：算法的额外内存开销，不包括输入所占空间。
- **in-place**：原位运算，直接在输入数组/链表的基础上修改。
- **k**：计数排序/桶排序，桶的个数；基数排序，关键字位数。

7.7.1 插入排序

Code

```
template<class T>
void insertionSort(T* arr, int len)
{
  if(!arr) return;
  for(int i = 1; i < len; ++i)
```

(continues on next page)

(continued from previous page)

```
  {
    int j = i;
    while(arr[j] < arr[j-1])
    {
      swap(arr[j], arr[j-1]);
      -- j;
    }
  }
}
```

7.7.2 选择排序

Code

```
template<class T>
void selectionSort(T* arr, int len)
{
  if(!arr) return;
  for(int i = 0; i < len - 1; ++i)
  {
    int k = i;
    for(int j = i+1; j < len; ++j)
    {
      if(arr[j] < arr[k]) k = j;
    }
    swap(arr[i], arr[k]);
  }
}
```

7.7.3 冒泡排序

Code

```cpp
// 下起泡: 大的数下沉
template<class T>
void bubbleSort(T* arr, int len)
{
  if(!arr) return;
  for(int i = 1; i < len; ++i)
  {
    for(int j = 0; j < len - i; ++j)
    {
      if(arr[j] > arr[j+1]) swap(arr[j], arr[j+1]);
    }
  }
}

// 上起泡: 小的数上浮
template<class T>
void bubbleSort(T* arr, int len)
{
  if(!arr) return;
  for(int i = 0; i < len - 1; ++i)
  {
    for(int j = len - 1; j > i; --j)
    {
      if(arr[j] < arr[j-1]) swap(arr[j], arr[j-1]);
    }
  }
}
```

7.7.4 快速排序

Code

```cpp
// 全闭区间 [start, end]
template<class T>
int partion(T* arr, int start, int end)
{
  T p = arr[start]; // pivot
  int left = start;
```

(continues on next page)

(continued from previous page)

```
  int right = end + 1;
  while(true)
  {
    while(arr[++left] < p && left < end);
    while(arr[--right] > p);
    if(left >= right) break;
    swap(arr[left], arr[right]);
  }
  swap(arr[start], arr[right]);
  return right;
}

template<class T>
void quickSort(T* arr, int start, int end)
{
  if(!arr || start >= end) return;
  int p = partion(arr, start, end);
  if(p > start + 1) quickSort(arr, start, p-1);
  if(p < end - 1) quickSort(arr, p+1, end);
}
```

快速排序的空间复杂度是 $\mathcal{O}(\log n)$，用于保存递归的函数栈，最差情况下为 $\mathcal{O}(n)$。

7.7.5 堆排序

建堆，从 **最后一个非叶子节点**开始调整，使其成为大顶堆；将堆顶元素放到数组末尾；最后一个叶子节点放到堆顶，重新调整堆；…。

Code

```
// 调整堆。区间 [start, end]，除了 start 不满足大顶堆的性质之外，其他节点都满足。
template<class T>
void heapAdjust(T* arr, int start, int end)
{
  T tmp = arr[start];
  for(int i = 2*start+1; i <= end; i = 2*i + 1)
  {
```

(continues on next page)

(continued from previous page)

```
    if(i < end) i = arr[i] > arr[i+1] ? i: i+1;
    if(arr[i] < tmp) break;
    arr[start] = arr[i];
    start = i;
  }
  arr[start] = tmp;
}

// 某节点下标为 i，则其左右子节点的下标分别为：2*i+1, 2*i+2 。
template<class T>
void heapSort(T* arr, int len)
{
  if(!arr) return;
  for(int k = (len-1-1)/2; k>=0; --k) heapAdjust(arr, k, len-1);
  for(int i = 1; i <= len; ++i)
  {
    swap(arr[0], arr[len-i]);
    heapAdjust(arr, 0, len-1-i);
  }
}
```

初始建立大顶堆的时间复杂度为 $\mathcal{O}(n \log n)$ ； 每次取出堆的最大元素并重新调整堆也要用 $\mathcal{O}(\log n)$ 时间。

7.7.6 归并排序

Code

```
// 把有序表 from: [start, mid] 和 from: [mid+1, end] 合并到临时数组 to: [start, end]。
template<class T>
void merge(T* from, T* to, int start, int mid, int end)
{
  int i, j, k;
  for(i = start, j = mid+1, k = start; i <= mid && j <= end; ++k)
  {
    if(from[i] < from[j]) to[k] = from[i++];
    else to[k] = from[j++];
  }
```

(continues on next page)

(continued from previous page)

```
  for(;i <= mid; ) to[k++] = from[i++];
  for(;j <= end; ) to[k++] = from[j++];
}

template<class T>
void mergeSort(T* arr, T* atmp, int start, int end)
{
  if(start == end) return;
  int mid = start + (end - start) / 2;
  mergeSort(arr, atmp, start, mid);
  mergeSort(arr, atmp, mid+1, end);
  merge(arr, atmp, start, mid, end);
  for(int i = start; i <= end; ++i) arr[i] = atmp[i];
}

template<class T>
void mergeSort(T* arr, int start, int end)
{
  if(!arr) return;
  T* atmp = new T[MAX_LEN]; // 申请临时空间
  fill(atmp, atmp + MAX_LEN, -1);
  mergeSort(arr, atmp, start, end);
  delete[] atmp;
}
```

7.7.7 希尔排序

缩小增量排序法：对于每一个增量（步长），利用插入排序方法进行排序。如果序列是基本有序的，使用直接插入排序效率非常高。

Code

```
template<class T>
void insertSort(T* arr, int start, int gap, int len)
{
  for(int i = start + gap; i < len; i += gap)
  {
    int j = i;
```

(continues on next page)

(continued from previous page)

```
    while(arr[j] > arr[j - gap])
    {
      swap(arr[j], arr[j - gap]);
      j -= gap;
    }
  }
}

template<class T>
void shellSort(T* arr, int len)
{
  if(!arr) return;
  for(int gap = len/2; gap >= 1; gap /= 2)
  {
    for(int start = 0; start < gap; ++ start) insertSort(arr, start, gap, len);
  }
}
```

7.7.8 计数排序

Code

```
// 空间复杂度 O(n+k)
void Sort(vector<int> &arr, int maxVal)
{
  int len = arr.size();
  if (len < 1) return;

  vector<int> count(maxVal + 1, 0);
  vector<int> tmp(arr);

  for (auto x : arr) count[x]++;

  partial_sum(count.begin(), count.end(), count.begin());

  for (int i = len - 1; i >= 0; --i)
  {
```

(continues on next page)

(continued from previous page)

```
    int val = tmp[i];
    arr[count[val] - 1] = val;
    count[val]--;
  }
}

// 空间复杂度 O(k)
void Sort(vector<int> &arr, int maxVal)
{
  int len = arr.size();
  if (len < 1) return;

  vector<int> count(maxVal + 1, 0);

  for (auto x : arr) count[x]++;

  int i = 0;
  for (int x = 0; x <= maxVal; ++x)
  {
    while (count[x]-- > 0) arr[i++] = x;
  }
}
```

7.7.9 桶排序

参考：https://blog.csdn.net/developer1024/article/details/79770240

Code

```
#include<iterator>
#include<iostream>
#include<vector>
using namespace std;
const int BUCKET_NUM = 10;

struct ListNode
{
    explicit ListNode(int i=0):mData(i),mNext(NULL){}
```

(continues on next page)

(continued from previous page)

```
    ListNode* mNext;
    int mData;
};

ListNode* insert(ListNode* head,int val)
{
    ListNode dummyNode;
    ListNode *newNode = new ListNode(val);
    ListNode *pre,*curr;
    dummyNode.mNext = head;
    pre = &dummyNode;
    curr = head;
    while(NULL!=curr && curr->mData<=val)
    {
        pre = curr;
        curr = curr->mNext;
    }
    newNode->mNext = curr;
    pre->mNext = newNode;
    return dummyNode.mNext;
}
ListNode* Merge(ListNode *head1,ListNode *head2)
{
    ListNode dummyNode;
    ListNode *dummy = &dummyNode;
    while(NULL!=head1 && NULL!=head2)
    {
        if(head1->mData <= head2->mData)
        {
            dummy->mNext = head1;
            head1 = head1->mNext;
        }
        else
        {
            dummy->mNext = head2;
            head2 = head2->mNext;
        }
        dummy = dummy->mNext;
    }
    if(NULL!=head1) dummy->mNext = head1;
    if(NULL!=head2) dummy->mNext = head2;

    return dummyNode.mNext;
```

(continues on next page)

(continued from previous page)

```
}
void BucketSort(int n,int arr[])
{
    vector<ListNode*> buckets(BUCKET_NUM,(ListNode*)(0));

    // 插入桶中
    for(int i=0;i<n;++i)
    {
        int index = arr[i]/BUCKET_NUM;
        ListNode *head = buckets.at(index);
        buckets.at(index) = insert(head,arr[i]);
    }

    // 合并各个桶中的排序结果
    ListNode *head = buckets.at(0);
    for(int i=1;i<BUCKET_NUM;++i)
    {
        head = Merge(head,buckets.at(i));
    }

    // 结果输出到 arr
    for(int i=0;i<n;++i)
    {
        arr[i] = head->mData;
        head = head->mNext;
    }
}
```

时间复杂度 对于 n 个待排数据，k 个桶，平均每个桶 $\frac{n}{k}$ 个数据，桶内排序复杂度为 $\mathcal{O}(\frac{n}{k}\log\frac{n}{k})$ ，总体平均时间复杂度为：

$$\mathcal{O}(n)+\mathcal{O}(k*\frac{n}{k}\log\frac{n}{k})=\mathcal{O}(n+n(\log n-\log k)).$$

当 $n=k$ ，每个桶只有一个数据，时间复杂度为 $\mathcal{O}(n)$ 。

7.7.10 基数排序

Code

```cpp
// digit 表示关键字位数
void radixSort(int* arr, int len, int digit)
{
  if(!arr) return;

  vector<vector<int>> radix(10, vector<int>{});
  int order = 1;
  while(digit--)
  {
    for(int i = 0; i < len; ++i)
    {
      int idx = (arr[i] / order) % 10;
      radix[idx].emplace_back(arr[i]);
    }

    int k = 0;
    for(int i = 0; i < 10; ++i)
    {
      int j = 0;
      while (j < radix[i].size()) arr[k++] = radix[i][j++];
      while (j--) radix[i].pop_back();
    }

    order *= 10;
  }
}
```

7.7.11 总结

- 从平均时间来看，**快速排序**是效率最高的，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。
- 在 n 较大时 **归并排序**使用时间较少，但使用辅助空间较多。
- 当序列基本有序或 n 较小时，直接 **插入排序**是好的方法，因此常将它和其他的排序方法（如快速排序、归并排序等）结合在一起使用。
- **选择排序**、**堆排序**、**快速排序**、**希尔排序**是不稳定的排序方法。
- **基数排序**适合于 n 较大而关键字位数较少的情况。
- 如果我们只希望找到数组中前 k 大的元素，且 k 很小，则 **堆排序**速度较快。

7.7.12 参考资料

1. 十大经典排序算法（动图演示）

   https://www.cnblogs.com/onepixel/p/7674659.html

2. 十大经典排序算法

   https://zhuanlan.zhihu.com/p/41923298

3. 10 大经典排序算法动图演示

   https://www.cnblogs.com/zhuqi7758258/articles/10643262.html

7.8 动态规划

7.8.1 矩阵连乘

矩阵连乘，通过调整加括号的方式，使得乘法元素次数最少。设矩阵链 $A[0:n-1]$ ，$A[i]$ 的维度为 $p_i \times p_{i+1}$ 。$m[i][j]$ 是计算 $A[i:j]$, $1 \leqslant i \leqslant j \leqslant n$ 所需的最少乘法次数。

递归关系：

$$m[i][j] = \begin{cases} min\{m[i][k] + m[k+1][j] + p_i \times p_{k+1} \times p_{j+1}\},\ i \leqslant k < j \\ i \neq j \\ 0 \\ i = j \end{cases}$$

Code

```
// s[i][j] 记录 A[i:j] 的划分点 k
void matrixChain(int* p, int n, int** m, int** s)
{
  for(int i = 0; i < n; ++i) m[i][i] = 0;
  for(int gap = 1; gap < n - 1; ++gap)
  {
    for(int i = 0; i + gap < n; ++i)
    {
      int j = i + gap;
      m[i][j] = m[i+1][j] + p[i] * p[i+1] * p[j+1]; // k = i
      s[i][j] = i;
      for(int k = i+1; k < j; ++k)
      {
        int cost = m[i][k] + m[k+1][j] + p[i] * p[k+1] * p[j+1];
```

(continues on next page)

(continued from previous page)

```
        if(cost < m[i][j])
        {
          m[i][j] = cost;
          s[i][j] = k;
        }
      }
    }
  }
}
```

7.8.2 最长公共子序列

用 $c[i][j]$ 记录序列 $X[0:i-1]$ （前 i 个字符）和 $Y[0:j-1]$ （前 j 个字符）的最长公共子序列的长度。

递归关系：

$$c[0][j] = 0,\ 0 \leqslant j \leqslant nc[i][0] = 0,\ 0 \leqslant i \leqslant m$$

$$c[i][j] = \begin{cases} c[i-1][j-1] + 1 \\ i,j > 0;\ X[i-1] = Y[j-1] \\ max\{c[i-1][j], c[i][j-1]\} \\ i,j > 0;\ X[i-1] \neq Y[j-1] \end{cases}$$

Code

```
void lcsLength(char* x, int m, char* y, int n, int** c) // c 对应的实参为 int *c[]
{
  for(int i = 0; i <= m; ++i) c[i][0] = 0;
  for(int j = 0; j <= n; ++j) c[0][j] = 0;
  for(int i = 1; i <= m; ++i)
  {
    for(int j = 1; j <=n; ++j)
    {
      if(x[i-1] == y[j-1]) c[i][j] = c[i-1][j-1] + 1;
      else c[i][j] = max(c[i-1][j], c[i][j-1]);
    }
  }
}
```

```
/* 记录并构造公共子序列 */

void lcsLength(char* x, int m, char* y, int n, int** c, int** b)
{
  for(int i = 0; i <= m; ++i) c[i][0] = 0;
  for(int j = 0; j <= n; ++j) c[0][j] = 0;
  for(int i = 1; i <= m; ++i)
  {
    for(int j = 1; j <=n; ++j)
    {
      if(x[i-1] == y[j-1])
      {
        c[i][j] = c[i-1][j-1] + 1;
        b[i][j] = 0;
      }
      else
      {
        if(c[i-1][j] > c[i][j-1])
        {
          c[i][j] = c[i-1][j];
          b[i][j] = 1;
        }
        else
        {
          c[i][j] = c[i][j-1];
          b[i][j] = 2;
        }
      }
    }
  }
}

void lcs(char* x, int m, int n, int** b)
{
  if(m == 0 || n == 0) return;
  if(b[m][n] == 0)
  {
    lcs(x, m-1, n-1, b);
    cout << x[m-1];
  }
  else if(b[m][n] == 1) lcs(x, m-1, n, b);
  else lcs(x, m, n-1, b);
}
```

7.8.3 最长上升子序列

- 方法一

  设 $dp[i]$ 是以 $a[i]$ 结尾的最长上升子序列的长度。

  递归关系：

$$dp[i] = max\{1, dp[j] + 1 \mid j < i \text{ 且 } a[j] < a[i]\}.$$

Code

```
/* O(n^2) in time.*/
int n;
int a[MAX_N];

int dp[MAX_N];

int solve()
{
  int res = 0;
  for(int i = 0; i < n; ++i)
  {
    dp[i] = 1;
    for(int j = 0; j < i; ++ j)
    {
      if(a[j] < a[i]) dp[i] = max(dp[i], dp[j] + 1);
    }
    res = max(res, dp[i]);
  }
  return res;
}
```

- 方法二

  设 $dp[i]$ 是长度为 $i+1$ 的上升子序列中末尾元素的最小值。

Code

```
/* https://leetcode.com/problems/longest-increasing-subsequence/ */
/* O(nlogn) in time.*/
class Solution
{
public:
```

(continues on next page)

(continued from previous page)

```
  int lengthOfLIS(vector<int>& nums)
  {
    if(nums.size()<=1) return nums.size();
    int inf = INT_MAX;
    int len = nums.size();
    int* dp = new int[len];
    fill(dp, dp+len, inf);
    for(int k = 0; k < len; ++k) *lower_bound(dp, dp+len, nums[k]) = nums[k];
    int length = lower_bound(dp, dp+len, inf) - dp;
    delete[] dp;
    return length;
  }
};
```

7.8.4 最大子段和

设 $dp[i]$ 是以 $a[i]$ 结尾的最大子段和。

递归关系：

$$dp[i] = max\{dp[i-1] + a[i], a[i]\},\ 1 \leqslant i < n.$$

Code

```
int maxSum(int* a, int n)
{
  int dp = 0;
  int res = 0;
  for(int i = 0; i < n; ++i)
  {
    if(dp > 0) dp += a[i];
    else dp = a[i];
    res = max(res, dp);
  }
  return res;
}
```

7.8.5 0-1 背包问题

设 $dp[i][j]$ 表示从 0 到 $i-1$ 这前 i 个物品中选出总重量不超过 j 的物品时总价值的最大值。

递归关系：

$$dp[0][j] = 0,\ 0 \leqslant j \leqslant W$$

$$dp[i+1][j] = \begin{cases} dp[i][j] \\ j < w[i] \\ max\{dp[i][j], dp[i][j-w[i]] + v[i]\} \\ j \geqslant w[i] \end{cases}$$

Code

```
int n, W;
int w[MAX_N], v[MAX_N];
int dp[MAX_N+1][MAX_W+1];
int solve()
{
  for(int i = 1; i <= n; ++i)
  {
    for(int j = 0; j <= W; ++j)
    {
      if(j < w[i]) dp[i][j] = dp[i-1][j];
      else dp[i][j] = max(dp[i][j], dp[i-1][j-w[i]] + v[i]);
    }
  }
  return dp[n][W];
}
```

7.8.6 实例

- 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

  Code

```
import numpy as np
money = np.array([1, 5, 10, 20, 50, 100])
```

(continues on next page)

(continued from previous page)

```
dp = np.array([[0 for i in range(10000+1)] for j in range(6+1)], dtype=np.int64)
## dp[m,n]: first m currency values, make money n
dp[0,:] = 0
dp[:,0] = 1
for m in range(1,6+1):
    for n in range(1, 10000+1):
        if n >= money[m-1]:
            dp[m,n] = dp[m,n-money[m-1]] + dp[m-1,n]
        else:
            dp[m,n] = dp[m-1,n]
print dp[6, 10000]
```

```
// 作者: 李泽政
// 链接: https://www.zhihu.com/question/315108379/answer/620254961

#include<cstdio>
#define maxn 10001
long long dp[maxn];
int main(void)
{
    int i,j,num[] = {5, 10, 20, 50, 100};
    for(i = 0; i < maxn; ++i)
        dp[i] = 1; // 作者把 1 从 num[] 中去掉了，转化到初始化中。全用 1 元只能得到一种组成方案
    for(i = 0; i < 5; ++i)
        for(j = num[i]; j < maxn; ++j)
            dp[j] += dp[j - num[i]];
    printf("%lld", dp[maxn - 1]);
    return 0;
}
```

- 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

  Code

```
## 作者: 知乎用户
## 链接: https://www.zhihu.com/question/19690210/answer/18079633
## f(n,m): n 层楼，m 个鸡蛋所需最少次数
## f(0, m) = 0
## f(n, 1) = n
## f(n, m) = min{max{f(k-1, m-1), f(n-k, m)}} + 1, 1 <= k <= n。 k 表示尝试在第 k 层扔下鸡蛋。

import functools
```

(continues on next page)

(continued from previous page)

```python
@functools.lru_cache(maxsize=None)
def f(n, m):
    if n == 0:
        return 0
    if m == 1:
        return n

    ans = min([max([f(i - 1, m - 1), f(n - i, m)]) for i in range(1, n + 1)]) + 1
    return ans

print(f(100, 2))  # 14
print(f(200, 2))  # 20
```

7.9 回溯

总体思路是深度优先遍历（DFS）。

7.9.1 子集树

子集树大小为 $\mathcal{O}(m^n)$ ，m 是树的分支个数（m 叉树），n 是树的深度。

算法描述：

```
void Backtrack(int t)
{
  if(t >= n) Output(x);
  else
  {
    for(int i = 0; i < m; ++i)
    {
      x[t] = i;
      if(Constrain(t) and Bound(t)) Backtrack(t+1);
    }
  }
}
```

7.9.2 排列树

排列树大小为 $\mathcal{O}(n!)$ 。

算法描述：

```
void Backtrack(int t)
{
  if(t >= n) Output(x);
  else
  {
    for(int k = t; k < n; ++k)
    {
      Swap(x[t], x[k]);
      if(Constrain(t) and Bound(t)) Backtrack(t+1);
      Swap(x[t], x[k]);
    }
  }
}
```

7.9.3 0-1 背包问题

算法描述：

```
void Backtrack(int t)
{
  if(t >= n)
  {
    best_value = curr_value;
    bext_x = x;
    return;
  }
  else
  {
    if(curr_weight + w[t] <= W)
    {
      x[t] = 1;
      curr_weight += w[t]; // 进入左子树
      curr_value += v[t];
      Backtrack(t+1);

      curr_weight -= w[t]; // 状态恢复
      curr_value -= v[t];
    }
    x[t] = 0;
    Backtrack(t+1); // 进入右子树
  }
}
```

7.9.4 实例

- 全排列（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

  Code

```
int cnt = 0; // 不同排列的个数

//检查 [from,to) 之间的元素和第 to 号元素是否相同
bool isRepeat(int* A, int from, int to)
{
    for(int i = from; i < to; i++)
    {
        if(A[to] == A[i]) return true;
    }
    return false;
}

void permutation(int* A, int t, int n)
{
    if(t == n)
    {
        cnt++;
        Output(A);
    }
    else
    {
        for(int j = t; j < n; j++)
        {
            if(!isRepeat(A, t, j))
            {
                swap(A[t], A[j]);
                permutation(A, t+1, n);
                swap(A[t], A[j]);
            }
        }
    }
}
```

- Next Permutation 下一个排列。Hint：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 和 y；翻转区间 $[i+1,end]$。

  https://www.cnblogs.com/grandyang/p/4428207.html

Code

```
class Solution
{
public:
    void nextPermutation(vector<int> &num)
    {
        int i, j, n = num.size();
        for (i = n - 2; i >= 0; --i)
        {
            if (num[i + 1] > num[i])
            {
                for (j = n - 1; j > i; --j)
                {
                    if (num[j] > num[i]) break;
                }
                swap(num[i], num[j]);
                reverse(num.begin() + i + 1, num.end());
                return;
            }
        }
        reverse(num.begin(), num.end()); // 当前排列是最大的排列，则翻转为最小的排列
    }
};
```

- Word search 查找字符串路径。

  https://leetcode.com/problems/word-search/

Code

```
class Solution {
public:
    bool find_path(vector<vector<char>>& board, string word, bool** flag, int x, int y, int k)
    {
        if(k == word.size()) return true;
        for(int t = 0; t < 4; ++t)
        {
            int tx = x + mv[t][0];
            int ty = y + mv[t][1];

            if(flag[tx+1][ty+1] && board[tx][ty] == word[k])
            {
                flag[tx+1][ty+1] = false; // 设置 flag
                if(find_path(board, word, flag, tx, ty, k+1)) return true;
                flag[tx+1][ty+1] = true; // flag 还原
```

(continues on next page)

(continued from previous page)

```
            }

        }
        return false;
    }
    bool exist(vector<vector<char>>& board, string word) {
        if(word=="") return true;
        if(board.size()==0) return false;
        int M = board.size();
        int N = board[0].size();
        bool** flag = new bool*[M+2];
        for(int m = 0; m < M+2; ++m)
        {
            flag[m] = new bool[N+2];
            for(int n = 0; n < N+2; ++n)
            {
                if(m==0 || m==M+1 || n==0 || n==N+1) flag[m][n] = false;
                else flag[m][n] = true;
            }
        }
        bool EXIST = false;
        for(int i = 0; i < M; ++i)
        {
            for(int j = 0; j < N; ++j)
            {
                if(board[i][j] == word[0])
                {
                    flag[i+1][j+1] = false; // 注意： flag 的下标与 board 相差 1
                    if(find_path(board, word, flag, i, j, 1))
                    {
                        EXIST = true;
                        break; // 跳出第二重循环
                    }
                    flag[i+1][j+1] = true; // flag 还原
                }
            }
            if(EXIST) break; // 跳出第一重循环
        }

        for(int m = 0; m < M+2; ++m) delete[] flag[m];
        delete[] flag;

        return EXIST;
```

(continues on next page)

(continued from previous page)

```
59      }
60  private:
61      static const int mv[4][2];
62  };
63
64  const int Solution::mv[4][2] = {{-1,0},{0,-1},{0,1},{1,0}};
```

CHAPTER

EIGHT

计算机网络

8.1 计算机网络体系结构

8.1.1 不同的体系结构

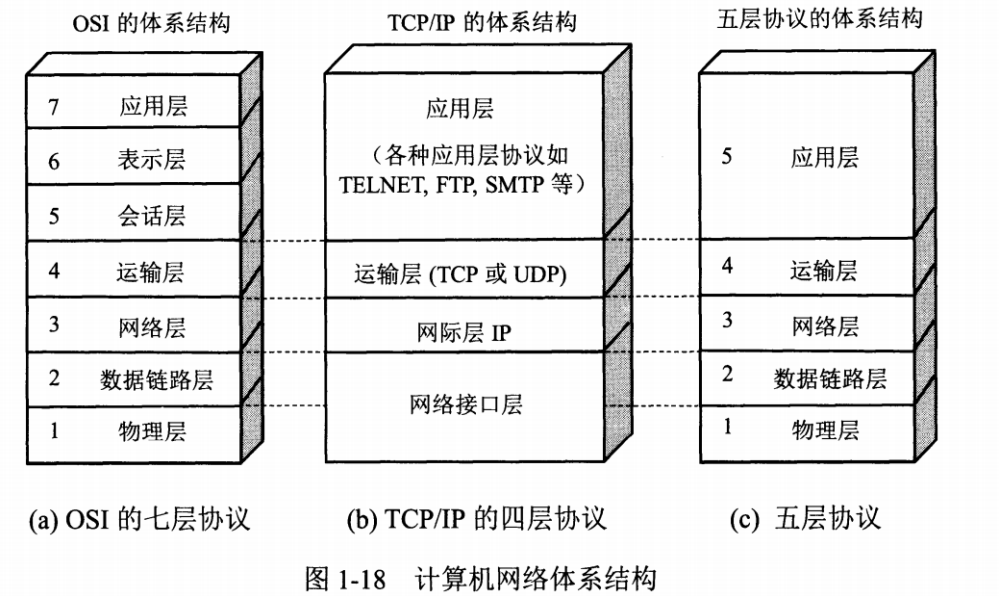


图 1-18　计算机网络体系结构

8.1.2 七层网络体系

OSI（Open System Interconnect Reference Model）：开放式系统互联参考模型。

- **物理层（pysical layer）** 把电脑连接起来的物理手段，如光缆、电缆、双绞线、无线电波。它规定了网路的一些电气属性，作用是负责传输 0 和 1 比特的电信号。
- **数据链路层（data link layer）** 物理寻址，并将比特组装成帧和点到点的传递。
- **网络层（network layer）** 负责数据包从源到宿的传递和网际互连，控制子网的运行，逻辑编址、分组传输、路由选择。

- **传输层（transport layer）** 提供端到端的可靠报文传递和错误恢复。
- **会话层（session layer）** 负责建立、管理和断开通信连接，以及数据的分割等数据传输相关的管理。
- **表示层（presentation layer）** 设备固有的数据格式与网络标准数据格式之间的转换（接受不同格式的信息，如文字流、图像、声音等）。
- **应用层（application layer）** 针对特定应用的协议，如电子邮件协议、SSH、FTP、HTTP。

8.1.3 参考资料

1. 七层网络结构

   https://blog.csdn.net/u010359398/article/details/82142449

8.2 TCP

TCP（Transmission Control Protocol，传输控制协议）是一种 **面向连接的**、**可靠的**、**基于字节流**的传输层通信协议。

8.2.1 三次握手与四次挥手

三次握手

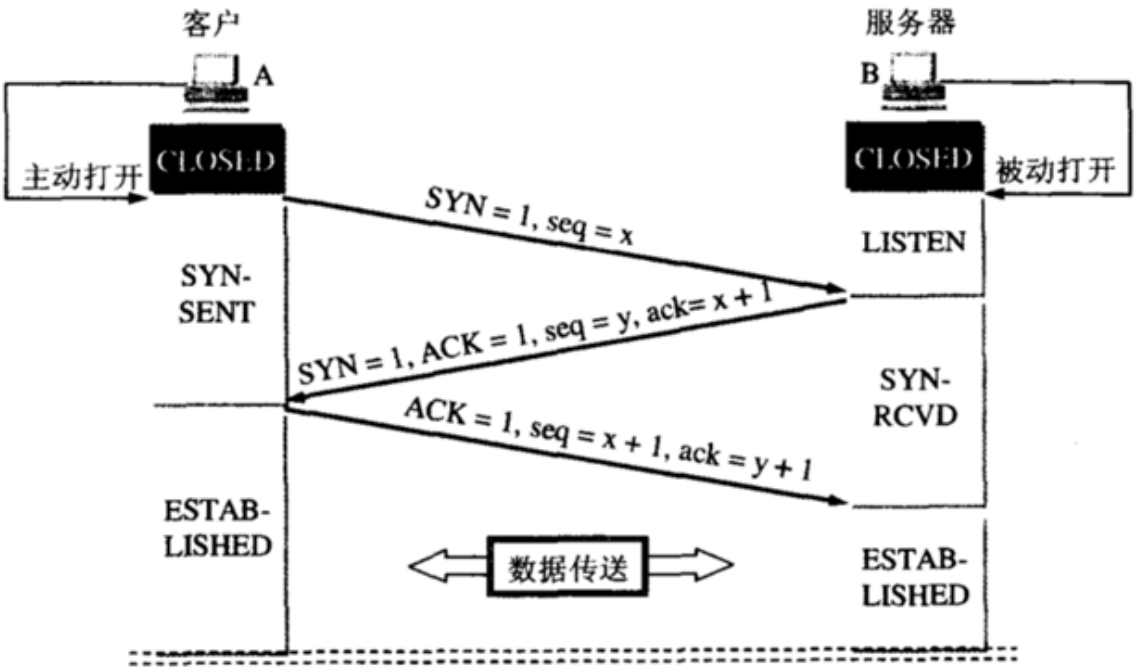


图 5-31　用三次握手建立 TCP 连接

- **第一次握手** 起初两端都处于 CLOSED 关闭状态，Client 将标志位 SYN 置为 1，随机产生一个值 seq=x，并将该数据包发送给 Server ，Client 进入 SYN-SENT 状态，等待 Server 确认。
- **第二次握手** Server 收到数据包后由标志位 SYN=1 得知 Client 请求建立连接，Server 将标志位 SYN 和 ACK 都置为 1，ack=x+1，随机产生一个值 seq=y，并将该数据包发送给 Client 以确认连接请求，Server 进入 SYN-RCVD 状态，此时操作系统为该 TCP 连接分配 TCP 缓存和变量。
- **第三次握手** Client 收到确认后，检查 ack 是否为 x+1，ACK 是否为 1，如果正确则将标志位 ACK 置为 1，ack=y+1，并且此时操作系统为该 TCP 连接分配 TCP 缓存和变量，并将该数据包发送给 Server 。Server 检查 ack 是否为 y+1，ACK 是否为 1，如果正确则连接建立成功，Client 和 Server 进入 ESTABLISHED 状态，完成三次握手，随后 Client 和 Server 就可以开始传输数据。

Note:

为什么 Client 还要发送一次确认呢？可以二次握手吗？ 主要为了防止已失效的连接请求报文段突然又传送到了 Server ，因而产生错误。如 Client 发出连接请求，但因连接请求报文丢失而未收到确认，于是 Client 再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接，Client 发出了两个连接请求报文段，其中第一个丢失，第二个到达了 Server ，但是第一个丢失的报文段只是在某些网络结点长时间滞留了，延误到连接释放以后的某个时间才到达 Server ，此时 Server 误认

为 Client 又发出一次新的连接请求，于是就向 Client 发出确认报文段，同意建立连接。不采用三次握手，只要 Server 发出确认，就建立新的连接了，此时 Client 不理睬 Server 的确认且不发送数据，则 Server 一直等待 Client 发送数据，浪费资源。

四次挥手

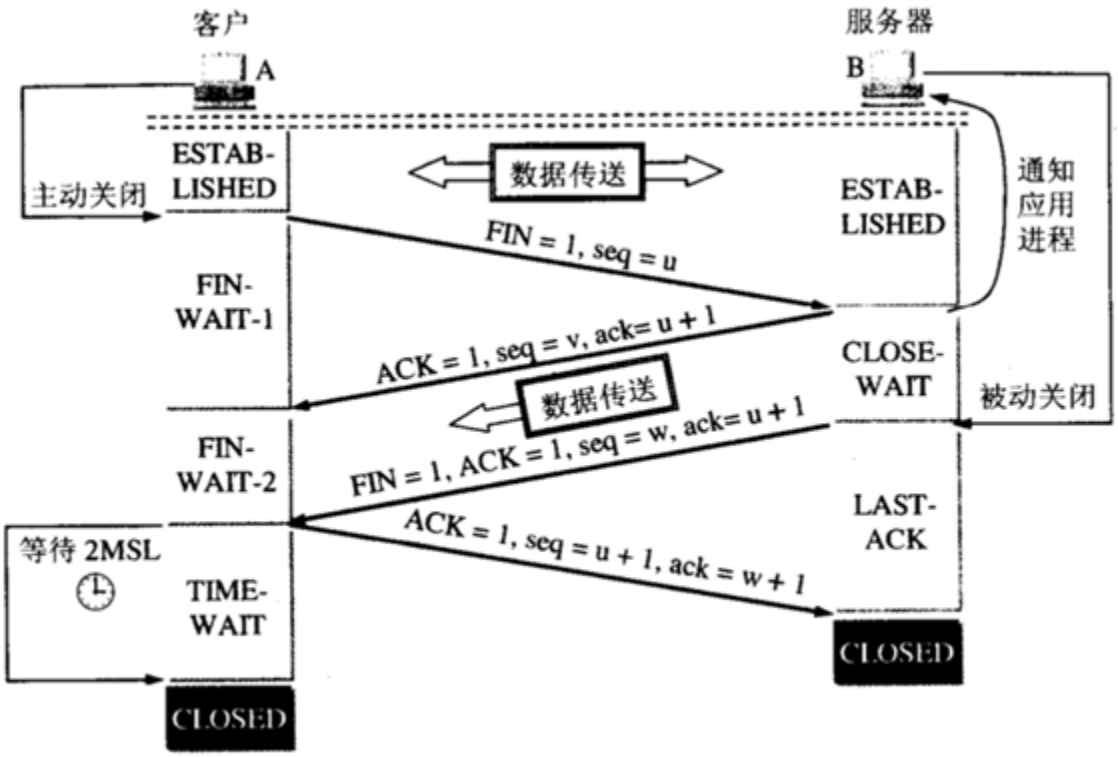


图 5-32　TCP 连接释放的过程

- **第一次挥手** Client 的应用进程先向其 TCP 发出连接释放报文段（FIN=1，序号 seq=u），并停止再发送数据，主动关闭 TCP 连接，进入 FIN-WAIT-1（终止等待 1）状态，等待 Server 的确认。
- **第二次挥手** Server 收到连接释放报文段后即发出确认报文段，（ACK=1，确认号 ack=u+1，序号 seq=v），Server 进入 CLOSE-WAIT（关闭等待）状态，此时的 TCP 处于半关闭状态，Client 到 Server 的连接释放；

  Client 收到 Server 的确认后，进入 FIN-WAIT-2（终止等待 2）状态，等待 Server 发出的连接释放报文段。
- **第三次挥手** Server 没有要向 Client 发出的数据了，Server 发出连接释放报文段（FIN=1，ACK=1，序号 seq=w，确认号 ack=u+1），Server 进入 LAST-ACK（最后确认）状态，等待 Client 的确认。
- **第四次挥手**

`Client` 收到 `Server` 的连接释放报文段后，对此发出确认报文段（ACK=1，seq=u+1，ack=w+1），`Client` 进入 TIME-WAIT（时间等待）状态。

`Server` 收到确认报文段后进入 CLOSED 状态。

经过时间等待计时器设置的时间 2MSL 后，`Client` 才进入 CLOSED 状态。

Note:

- **为什么连接的时候是三次握手，关闭的时候却是四次挥手？** 因为当 `Server` 收到 `Client` 的 SYN 连接请求报文后，可以直接同时发送 SYN+ACK 报文。其中 ACK 报文是用来应答的，SYN 报文是用来同步的。但是关闭连接时，当 `Server` 端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉 `Client` 端："你发的 FIN 报文我收到了，只有等到我 `Server` 端所有的报文都发送完了，我才能发送 FIN 报文"。即 FIN 和 ACK 不能一起发送，故需要四步握手。
- **为什么 `Client` 需要经过 2MSL（最大报文段生存时间）才能从 TIME_WAIT 状态进入 CLOSE 状态？** 最后一个 ACK 有可能丢失，所以 TIME_WAIT 状态就是用来重发可能丢失的 ACK 报文。

8.2.2 TCP 和 UDP 的区别

TCP 是 **面向连接的、可靠的、基于字节流**的数据流传输协议，效率低。

UDP （User Datagram Protocol，用户数据协议）是 **非面向连接的、不可靠**的数据流传输协议。不需要建立连接，只需要知道对方的 IP（Internet Protocol）地址和端口号（port），就可以直接发数据包。因此效率高。

8.2.3 参考资料

1. TCP 三次握手和四次挥手过程

   https://www.cnblogs.com/Andya/p/7272462.html

8.3 网桥、交换机、路由器、网关

8.3.1 网桥（Bridge）

两个或多个以太网通过 **网桥**连接起来后，就成为一个覆盖范围更大的以太网，而原来的每个以太网就可称为一个网段。

网桥工作在数据链路层的 MAC 子层，可以互联不同的物理层、不同的 MAC 子层以及不同速率的以太网，使以太网各网段成为隔离开的碰撞域。

网桥具有过滤帧以及存储转发帧的功能，可以隔离冲突域，但不能隔离广播域。

8.3.2 交换机（Switch）

交换机工作在数据链路层，相当于一个多端口的网桥，是交换式局域网的核心设备。

交换机内部的 CPU 会在每个端口成功连接时，通过 ARP 协议学习它的 MAC 地址，保存成一张 ARP 表。在今后的通讯中，发往该 MAC 地址的数据包将仅送往其对应的端口，而不是所有的端口。

利用 **交换机**可以实现虚拟局域网 (VLAN)，VLAN 可以隔离冲突域，也可以隔离广播域。

8.3.3 路由器（Router）

路由器是网络层设备，可以连接不同的网络（异构网络）并主要完成两个功能：分组转发和路由计算。前者处理通过路由器的数据流，关键操作是转发表查询、转发以及相关的队列管理和任务调度等；后者通过和其他路由器进行基于路由协议的交互，完成路由表的计算。

8.3.4 网关（Gateway）

网关能在不同协议间移动数据，而路由器是在不同网络间移动数据。

网关是连接两个网络的设备。语音网关可以连接 PSTN 网络和以太网，这就相当于 VOIP，把不同电话中的模拟信号通过网关而转换成数字信号，并加入协议再进行传输。

8.3.5 参考资料

1. 牛客网：精华专题

   https://www.nowcoder.com/questionTerminal/497f4570bcfa4399a803f23fb92f7732

CHAPTER

NINE

资源链接

9.1 Github Page

https://fongyq.github.io/

9.2 AnyKnew

https://www.anyknew.com/#/

9.3 arXiv

https://arxiv.org/

9.4 Read the Docs

https://readthedocs.org/

9.5 C++ Reference

http://www.cplusplus.com/reference/

9.6 Numpy

http://cs231n.github.io/python-numpy-tutorial/

9.7 Pytorch

Tutorials: https://pytorch.org/tutorials/

Docs: https://pytorch.org/docs/master/index.html

9.8 Jupyter Notebook

https://jupyter.org/

9.9 Standford University Lectures

CS229: http://cs229.stanford.edu/syllabus.html

CS231: http://cs231n.github.io/

9.10 ShareLatex

https://www.sharelatex.com/login

9.11 PlanetB

http://www.planetb.ca/syntax-highlight-word

9.12 Vision Open Source Library

检索：http://yael.gforge.inria.fr/index.html

特征：http://www.vlfeat.org/index.html

9.13 牛客网

https://www.nowcoder.com/

CHAPTER

TEN

实用软件

10.1 Listary

Note: Windows 下快速查找文件及应用程序

http://www.listary.com/

10.2 FreeCommander

Note: Windows 下的资源管理器

https://freecommander.com/en/summary/

10.3 MobaXterm

Note: Windows 下连接服务器的终端

https://mobaxterm.mobatek.net/

10.4 TeamViewer

Note: 远程连接

https://www.teamviewer.com/zhCN/

10.5 Notepad++

Note: 强大的文本阅读/编辑器

https://notepad-plus-plus.org/

10.6 cmder

Note: Windows 下终端神器

https://cmder.net/

CHAPTER

ELEVEN

TECH

11.1 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 +cmder+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/

- 在 windows 下安装 Jupyter Notebook 的安装和使用

  https://www.cnblogs.com/gengyi/p/9769471.html

- 加速 TexLive 编译

  https://blog.csdn.net/raby_gyl/article/details/73613601

11.2 问题

- Unable to locate package python-pip

  https://askubuntu.com/questions/268539/unable-to-locate-package-python-pip-on-live-system

- pytorch: custom nn modules & define new autograd functions

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-custom-nn-modules

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-defining-new-autograd-functions

CHAPTER

TWELVE

其他

12.1 复习

12.1.1 汇总

1. github

- 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

12.1.2 编程算法

1. 找出数组中 N 个出现 1（或奇数次）次的数字

   https://www.jianshu.com/p/e1331664c8cf

2. 均匀分布生成其他分布的方法

   https://blog.csdn.net/haolexiao/article/details/60511164

3. 海量数据处理。Hint：哈希方法，把大文件划分成小文件，读进内存依次处理；Bitmap，用一个（或几个）比特位来标记某个元素对应的值。

- 面试题集锦

  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

4. 链表

- 反转链表。Hint：方法一，逐个反转；方法二，递归；方法三，使用栈保存节点的值，反向赋给所有节点。

Code

```
struct ListNode
{
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

```
// 方法一，逐个反转
ListNode* reverseList(ListNode* head)
{
    if(head==NULL || head->next==NULL) return head;
    ListNode* newHead = head;
    ListNode* curr = head -> next;
    ListNode* post = curr -> next;
    newHead -> next = NULL;
    while(curr)
    {
        curr -> next = newHead;
        newHead = curr;
        curr = post;
        if(post) post = post -> next;
    }
    return newHead;
}
```

```
// 方法二，递归
ListNode* reverseList(ListNode* head)
{
    if(head==NULL || head->next==NULL) return head;
    else
    {
        ListNode* newHead = reverseList(head -> next);
        head -> next -> next = head; // head 指向的下一个节点是 newHead 的最后一个节点
        head -> next = NULL;
        return newHead;
    }
}
```

```
// 方法三，使用栈保存节点的值，占用 O(n) 额外空间
```

(continues on next page)

(continued from previous page)

```
ListNode* reverseList(ListNode* head)
{
    if(head==NULL || head->next==NULL) return head;
    stack<int> stk;
    ListNode* p = head;
    while(p)
    {
        stk.emplace(p -> val);
        p = p -> next;
    }
    p = head;
    while(p)
    {
        p -> val = stk.top();
        stk.pop();
        p = p -> next;
    }
    return head;
}
```

- 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 $\mathcal{O}(1)$ 时间删除给定节点。Hint：交换当前节点与下一个节点的值，删除下一个节点。

  https://blog.csdn.net/qq_35546040/article/details/80341136

5. 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

6. [LeetCode] Sort Colors（三颜色排序 →K 颜色排序）

   https://blog.csdn.net/princexiexiaofeng/article/details/79645511

7. 找到数组第 k 大的数

   https://leetcode.com/problems/kth-largest-element-in-an-array/

   Code

```
class Solution
{
public:
```

(continues on next page)

(continued from previous page)

```
    int partition(vector<int>& nums, int i, int j)
    {
        int pivot = nums[i];
        int l = i+1;
        int r = j;
        while(true)
        {
            while(l<=j && nums[l]<pivot) l++;
            while(r>i && nums[r]>pivot) r--;
            if(l>=r) break;
            swap(nums[l], nums[r]);
            l++;
            r--;
        }
        swap(nums[i], nums[r]);
        return r;
    }
    // partition 可用如下更简洁的形式
    int partition(vector<int>& nums, int i, int j)
    {
        int pivot = nums[i];
        int l = i;
        int r = j+1;
        while(true)
        {
            while(nums[++l]<pivot && l<j);
            while(nums[--r]>pivot);
            if(l>=r) break;
            swap(nums[l], nums[r]);
        }
        swap(nums[i], nums[r]);
        return r;
    }

    // T(n) = T(n/2) + O(n), 时间复杂度 O(N)
    int quicksort(vector<int>& nums, int a, int b, int k)
    {
        int p = partition(nums, a, b);
        if(b - p + 1 == k) return p;
        if(b - p + 1 < k) return quicksort(nums, a, p-1, k - (b - p + 1));
        else return quicksort(nums, p+1, b, k);
    }
    int findKthLargest(vector<int>& nums, int k)
```

(continues on next page)

(continued from previous page)

```
    {
        int k_id = quicksort(nums, 0, nums.size()-1, k);
        return nums[k_id];
    }
};
```

8. [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

9. [LeetCode] Partition Equal Subset Sum 数组分成两个子集，和相等

   https://leetcode.com/problems/partition-equal-subset-sum/

   Code

```
class Solution(object):
def backtrack(self, nums, sum_nums, sum_current, i): ## self
    if sum_current == sum_nums/2:
        return True
    if i == len(nums):
        return False
    if self.backtrack(nums, sum_nums, sum_current+nums[i],i+1): ## self
        return True
    if self.backtrack(nums, sum_nums, sum_current, i+1): ## self
        return True
    return False

def canPartition(self, nums):
    """
```

(continues on next page)

(continued from previous page)

```
    :type nums: List[int]
    :rtype: bool
    """
    if len(nums) <= 1:
        return False
    sum_nums = sum(nums)
    if sum_nums % 2:
        return False
    return self.backtrack(nums, sum_nums, 0, 0) ## self
```

10. [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/

    Code

```
/* https://leetcode.com/problems/find-all-anagrams-in-a-string/discuss/92027/C%2B%2B-
↪O(n)-sliding-window-concise-solution-with-explanation */

class Solution
{
public:
    vector<int> findAnagrams(string s, string p)
    {
        vector<int> vec;
        if(s.size()<p.size() || (s.empty() && p.empty())) return vec;
        vector<int> p_counter(26, 0), s_counter(26, 0);
        for(int i = 0; i < p.size(); ++i)
        {
            ++ p_counter[p[i]-'a'];
            ++ s_counter[s[i]-'a'];
        }
        if(p_counter == s_counter) vec.push_back(0);
        for(int i = p.size(); i < s.size(); ++i)
        {
            -- s_counter[s[i-p.size()]-'a'];
            ++ s_counter[s[i]-'a'];
            if(s_counter == p_counter) vec.push_back(i-p.size()+1);
        }
        return vec;
    }
};
```

11. [LeetCode] Find the Duplicate Number 寻找重复数。数值范围为 $\{1, 2, 3, ..., n\}$ 。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

Code

```
// 解法一：快慢指针，寻找某个“环”的入口
class Solution
{
public:
    int findDuplicate(vector<int>& nums)
    {
        int slow = 0, fast = 0, t = 0;
        while (true)
        {
            slow = nums[slow];
            fast = nums[nums[fast]];
            if (slow == fast) break;
        }
        while (true)
        {
            slow = nums[slow];
            t = nums[t];
            if (slow == t) break;
        }
        return slow;
    }
};
```

```
// 解法二：不断交换位置，找到第一个重复访问的元素
class Solution
{
public:
    int findDuplicate(vector<int>& nums)
    {
        int duplicate = -1;
        for(int k = 0; k < nums.size(); ++k)
        {
            while(nums[k]-1 != k)
            {
                if(nums[k] == nums[nums[k]-1])
                {
                    duplicate = nums[k];
```

(continues on next page)

(continued from previous page)

```
                    break;
                }
                swap(nums[k], nums[nums[k]-1]);
                // 一次交换之后，下标为 nums[k]-1 的元素就等于 nums[k] 了。
            }
            if(duplicate != -1) break;
        }
        return duplicate;
    }
};
```

- 找到所有重复数字。

  http://www.cnblogs.com/grandyang/p/6209746.html

Code

```
// 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
class Solution
{
public:
    vector<int> findDuplicates(vector<int>& nums)
    {
        vector<int> res;
        for (int i = 0; i < nums.size(); ++i)
        {
            int idx = abs(nums[i]) - 1;
            if (nums[idx] < 0) res.push_back(idx + 1);
            else nums[idx] = -nums[idx];
        }
        return res;
        // 这种方法得到的 res 可能多次包含同一个元素，可以使用 set
    }
};
```

```
// 解法二：不断交换位置使得 i == nums[i]-1
class Solution
{
public:
    vector<int> findDisappearedNumbers(vector<int>& nums)
    {
        vector<int> disappear;
        if(nums.size()<=1) return disappear;
        for(int k = 0; k < nums.size(); ++k)
```

(continues on next page)

(continued from previous page)

```
        {
            while(nums[k] != nums[nums[k]-1]) swap(nums[k], nums[nums[k]-1]);
        }
        for(int k = 0; k < nums.size(); ++k)
        {
            if(nums[k]-1 != k) disappear.push_back(nums[k]);
        }
        return disappear;
    }
};
```

12. [LeetCode] Spiral Matrix 环形打印矩阵

    https://leetcode.com/problems/spiral-matrix/

    Code

```
class Solution
{
public:
    void tranverseMatrixAccorindTo4Directions(vector<vector<int>> &matrix, const unsigned int row, const unsigned int col, int start, vector<int>& vec)
    {
        // 特别注意
        // 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。

        int endX = row-1 - start;
        int endY = col-1 - start;

        // 1 向右
        for(int k = start; k <= endY; ++k) vec.push_back(matrix[start][k]);

        // 2 向下
        for(int k = start+1; k <= endX; ++k) vec.push_back(matrix[k][endY]);

        // 3 向左：要求至少存在两行（不加判断会重复扫描同一行）
        if(endX > start) for(int k = endY-1; k >= start; --k) vec.push_back(matrix[endX][k]);

        // 4 向上：要求至少存在两列（不加判断会重复扫描同一列）
        if(endY > start) for(int k = endX-1; k > start; --k) vec.push_back(matrix[k][start]);

    }
```

(continues on next page)

(continued from previous page)

```
    vector<int> spiralOrder(vector<vector<int>>& matrix)
    {
        vector<int> vec;
        unsigned int row = matrix.size();
        if(row == 0) return vec;
        unsigned int col = matrix[0].size();
        if(col == 0) return vec;
        int start = 0;
        // 循环中止条件：圈数判断（(start,start)是每一圈的入口坐标）
        while(start*2 < row && start*2 < col)
        {
            tranverseMatrixAccorindTo4Directions(matrix, row, col, start, vec);
            ++ start;
        }
        return vec;
    }
};
```

13. [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/

    Code

```
class Solution(object):
def longestConsecutive(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """

    longest = 0
    num_set = set(nums)

    for num in nums:
        if num-1 not in num_set:
            current_long = 1
            while num + 1 in num_set:
                current_long += 1
                num += 1
            longest = max(longest, current_long)

    num_set.clear()
```

(continues on next page)

(continued from previous page)

```
20
21      return longest
```

14. 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

15. 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

$$(1+2+...+n)-x=\frac{n(n+1)}{2}-x$$

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

    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;
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 }
```

16. 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 …。

    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          {
```

(continues on next page)

(continued from previous page)

```
            res += n / 5;
            n /= 5;
        }
        return res;
    }
};
```

17. 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），或者对负数进行左移操作，结果都是未定义的。**例如：`n >> 32` 是未定义的，但是允许 `n >>= 1` 执行无限次，这是安全的。

Code

```
// 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␣
↪0xFFFFFFFF 而陷入死循环。
int Numberof1(int n)
{
  int cnt = 0;
  while(n)
  {
    if(n & 1) cnt ++;
    n >>= 1;
  }
  return cnt;
}
```

```
// 方法二：n 不动，左移一个比较子。
int Numberof1(int n)
{
  int cnt = 0;
  unsigned int flag = 1;
  while(flag) // 连续左移 32 次之后为 0
  {
    if(n & flag) cnt ++;
    flag <<= 1;
  }
  return cnt;
}
```

```
// 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。
int Numberof1(int n)
{
  int cnt = 0;
```

(continues on next page)

(continued from previous page)

```
  while(n)
  {
    cnt ++;
    n = (n - 1) & n;
  }
  return cnt;
}
```

18. [LeetCode] Subarray Sum Equals K 子数组和为 K 。Hint：依次求数组的前 n 项和 $sum[n]$ ，$n \in [0, arr\_size]$ （注意：0 也在内），将和作为哈希表的 key，和的值出现次数作为 value；如果存在 $sum[i]-sum[j] = K$ $(i \geq j)$ ，则 $sum[i]$ 和 $sum[j]$ 都应该在哈希表中。

https://leetcode.com/problems/subarray-sum-equals-k/

Code

```
// https://leetcode.com/problems/subarray-sum-equals-k/solution/ : Approach #4 Using␣
↪hashmap

from collections import defaultdict
class Solution(object):
def subarraySum(self, nums, k):
    """
    :type nums: List[int]
    :type k: int
    :rtype: int
    """

    if len(nums) == 0:
        return 0

    N = len(nums)

    sum_to_num = defaultdict(int)
    sum_to_num[0] = 1 // 前 0 项和

    cnt = 0
    tmp_sum = 0
    for n in nums:
        tmp_sum += n
        diff = tmp_sum - k
        cnt += sum_to_num[diff]
        sum_to_num[tmp_sum] += 1

    return cnt
```

19. 使用位运算进行加法运算。Hint：原位加法运算等效为 ^ 运算，进位等效为 & 和 移位的复合。**注：C++ 不允许对负数进行左移运算**。

    https://leetcode.com/problems/sum-of-two-integers/

    Code

```cpp
class Solution
{
public:
    int getSum(int a, int b)
    {
        int sum, carry;
        do
        {
            sum = (a ^ b);
            carry = (a & b & INT_MAX) << 1; // & INT_MAX 操作保证移位前的数是正数，否则结果是未定义的。
            a = sum;
            b = carry;
        }while(b != 0);
        return a;
    }
};
```

```python
from numpy import int32

class Solution(object):
    def getSum(self, a, b):
        """
        :type a: int
        :type b: int
        :rtype: int
        """
        a, b = int32(a), int32(b)

        while True:
            a, b = a ^ b, (a & b) << 1
            print a, b
            if b == 0:
                break

        return int(a)

## 注意，这里并没有与 0x7fffffff 做 & 运算
```

(continues on next page)

(continued from previous page)

```
## 假设 a & b = -16, -16 & 0x7fffffff = 2147483632
## C++ 中，对 2147483632 左移 1 位使得最高位符号位为 1，得到 -32
## python 中，2147483632 的符号位为 0，继续左移 1 位，会直接做大整数运算，得到 4294967264L，导致不能得到正确结果
## python 中，使用 type() 查看数据类型时发现，有时候系统会把 int32 转化为 int64，或者 int64转为 int32，疑惑中。。。
```

20. [LeetCode] Longest Substring with At Least K Repeating Characters 包含重复字符的最长子串。Hint：由于该字符串只包含小写字母，因此直接使用长度为 26 的静态数组来统计字符频率更为简洁高效，不需要使用 map。

https://leetcode.com/problems/longest-substring-with-at-least-k-repeating-characters/

Code

```
// https://www.cnblogs.com/grandyang/p/5852352.html
// 使用一个 int 型 (32 位) 的 mask，指示各字符频率是否到达 k
// 以每一个字符作为起点，往后统计。时间复杂度 O(N^2)
// mask 第 idx 位从 0 -> 1，表示对应字符出现了，但是未达到 k 次
// mask 第 idx 位从 1 -> 0，表示对应字符已经出现了 k 次
// mask 变成 0，表示这段子串满足要求

class Solution
{
public:
    int longestSubstring(string s, int k)
    {
        int ans = 0;
        int start = 0;
        while(start + k <= s.size())
        {
            int hash[26] = {0};
            int mask = 0;
            int next_start = start + 1;
            for(int end = start; end < s.size(); ++ end)
            {
                int idx = s[end] - 'a';
                ++ hash[idx];
                if(hash[idx] < k) mask |= (1 << idx); // 0 -> 1
                else mask &= ~(1 << idx);             // 1 -> 0
                if(mask == 0)
                {
                    ans = max(ans, end - start + 1);
                    next_start = end + 1;
```

(continues on next page)

(continued from previous page)

```
                }
            }
            start = next_start;
        }
        return ans;
    }
};
```

21. [LeetCode] 4Sum II 4 个数和为 0 的组合数。Hint：两两之和存入哈希表，时间复杂度和空间复杂度 $\mathcal{O}(N^2)$ 。

    https://leetcode.com/problems/4sum-ii/

    Code

```
def fourSumCount(self, A, B, C, D):
    AB = collections.Counter(a+b for a in A for b in B)
    return sum(AB[-c-d] for c in C for d in D)
```

22. [LeetCode] Maximum Product Subarray 求连续子数组的最大乘积。Hint：数组中存在负数，负负得正，因此相比于连续子数组最大和问题，不仅需要记录以每个元素结尾的连续乘积的最大值，还需要记录最小值。

    https://blog.csdn.net/xblog_/article/details/72872263

23. 给定一个十进制整数 N ，统计从 1 到 N 所有的整数各位出现的 1 的数目。Hint：1 的数目 = 个位出现 1 的数目 + 十位出现 1 的数目 + 百位出现 1 的数目 + ……。以百位为例：如果百位数字为 0，则百位出现 1 的次数只由更高位决定，如 12013，次数为 12 * 100；如果百位数字为 1，则百位出现 1 的次数由更高位和更低位同时决定，如 12113，次数为 12 * 100 + (113 + 1)；如果百位数字大于 1，则百位出现 1 的次数只由更高位决定，如 12213，次数为 (12 + 1) * 100。时间复杂度 $\mathcal{O}(\log_{10}(N))$ 。

    http://www.cnblogs.com/jy02414216/archive/2011/03/09/1977724.html

    Code

```
typedef unsigned long long ULL;
ULL number_of_1(ULL N)
{
  ULL cnt = 0;
  ULL factor = 1;
  ULL lowerNum = 0;
  ULL currNum = 0;
  ULL highNum = 0;
  while(N / factor)
  {
    lowerNum = N - (N / factor) * factor;
```

(continues on next page)

(continued from previous page)

```
    currNum = (N / factor) % 10;
    highNum = N / (factor * 10);
    switch(currNum)
    {
      case 0:
        cnt += highNum * factor;
        break;
      case 1:
        cnt += highNum * factor + (lowerNum + 1);
        break;
      default:
        cnt += (highNum + 1) * factor;
        break;
    }
    factor *= 10;
  }
  return cnt;
}
```

24. 数组循环移位：循环右移 K 位，时间复杂度 $\mathcal{O}(N)$ 。Hint：三次翻转。

    Code

```
void reverse(int *arr, int begin, int end)
{
  for(; begin < end; begin++, end--) swap(arr[begin], arr[end]);
}

void right_shift(int *arr, int N, int K)
{
  K %= N;
  reverse(arr, 0, N-K-1);
  reverse(arr, N-K, N-1);
  reverse(arr, 0, N-1);
}
```

25. [LeetCode] Divide Two Integers 整数除法。Hint：先取绝对值，做正整数之间的除法；防止溢出。

    https://leetcode.com/problems/divide-two-integers/

    Code

```
class Solution
{
public:
```

(continues on next page)

(continued from previous page)

```
    int divide(int dividend, int divisor)
    {
        if(dividend == INT_MIN && divisor == -1) return INT_MAX; // 越界则输出最大值
        if(dividend == INT_MIN && divisor == 1) return INT_MIN;
        if(divisor == INT_MIN && dividend == INT_MIN) return 1; // 枚举分子为最小整数时的情形
        if(divisor == INT_MIN) return 0;

        bool sign = (dividend>0) ^ (divisor>0) ? false : true;

        int res = 0;

        bool max_flow = false;
        if(dividend == INT_MIN)
        {
            dividend = abs(1 + INT_MIN); // 防止取绝对值之后溢出
            max_flow = true;
        }
        else dividend = abs(dividend);
        divisor = abs(divisor);

        while(dividend >= divisor)
        {
            int diff = divisor;
            int n = 1;
            while(diff <= (dividend >> 1))
            {
                diff <<= 1;
                n <<= 1;
            }
            dividend -= diff;
            res += n;
        }
        if(max_flow && dividend == divisor-1) res += 1;

        return sign? res : -res;
    }
};
```

26. [LeetCode] Fraction to Recurring Decimal 循环小数。Hint：小数除法：余数乘以 10 再求余；如果余数出现重复，则说明是循环小数。

    https://leetcode.com/problems/fraction-to-recurring-decimal/

    Code

```cpp
class Solution
{
public:
    string fractionToDecimal(int numerator, int denominator)
    {
        if(numerator == 0 || denominator == 0) return "0";
        int sign_num = numerator > 0? 1:-1;
        int sign_den = denominator > 0? 1:-1;

        long long num = abs((long long)numerator);
        long long den = abs((long long)denominator);

        long long integer = num / den;
        long long rem = num % den;

        string int_part = to_string(integer);
        if(rem) int_part += ".";

        string frac_part = "";
        unordered_map<long long, int> mp;
        int idx = 0;

        while(rem)
        {
            if(mp.find(rem) != mp.end()) // 余数重复
            {
                frac_part.insert(mp[rem], "(");
                frac_part += ")";
                break;
            }
            mp[rem] = idx ++;
            frac_part += to_string((10*rem) / den);
            rem = (10*rem) % den;
        }

        string res = "";
        if(sign_num * sign_den < 0) res += "-";
        res += int_part + frac_part;
        return res;
    }
};
```

27. 正整数质因数分解。

Code

```
## 不断除以 2 之后，2 的倍数都不可能再整除 n；3 同理。
## 思想类似于：找到 n 以内的素数，即把素数的倍数都排除。
def decomp(n):
    prime = 2
    while n >= prime:
        if n % prime == 0:
            print prime
            n /= prime
        else:
            prime += 1
```

28. 旋转数组查找。Hint：采用二分查找的思路。

- 二分查找

Code

```
// preliminary: binary search, 时间复杂度 O(logN)
template<class T>
int binarySearch(T *arr, int n, const T& target)
{
  if (arr == nullptr || n <= 0) return -1;
  int low = 0;
  int high = n - 1;
  while (low <= high)
  {
    int mid = low + (high - low) / 2; // mid = (low + high)/2 可能导致溢出
    if (arr[mid] == target) return mid;
    if (arr[mid] < target) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
```

- 查找旋转数组最小值（含重复元素）

  https://leetcode.com/problems/find-minimum-in-rotated-sorted-array-ii/

Code

```
// 方法一
// 第一个指针总指向前面递增数组的元素
// 第二个指针总指向后面递增数组的元素
// 最终两个指针指向相邻元素：第一个指针指向前面递增数组的最后一个元素，第二个指针指向后面递增数组的第一个元素（也就是最小元素）
template<class T>
```

(continues on next page)

(continued from previous page)

```
int findRotateMin(T* arr, int n)
{
  if (arr == nullptr || n <= 0) return -1;
  int low = 0;
  int high = n - 1;
  while (arr[low] >= arr[high])
  {
    if (high - 1 == low) return high;

    int mid = low + (high - low) / 2;

    // 如果这三个元素相等，则在区间 [low, high] 内顺序查找
    if (arr[low] == arr[mid] && arr[mid] == arr[high]) return (min_element(arr + low,␣
↪arr + high + 1) - arr);

    if (arr[mid] <= arr[high]) high = mid;
    else low = mid;
  }
  // 如果数组本身是有序的，即 arr[0] < arr[n-1]，则第一个元素就是最小值
  return 0;
}
```

```
// 方法二
// 如果 arr[mid] < arr[mid-1]，则 arr[mid] 是最小值
// 每次比较 nums[mid] 与 nums[high]，如果两者相等，则 --high
template<class T>
int findRotateMin(T* arr, int n)
{
  if (arr == nullptr || n <= 0) return -1;
  int low = 0;
  int high = n - 1;
  while (low <= high)
  {
    int mid = low + (high - low) / 2;
    if (mid > 0 && arr[mid] < arr[mid-1]) return mid;

    if (arr[mid] == arr[high]) --high;

    else if (arr[mid] < arr[high]) high = mid - 1;

    else low = mid + 1;
  }
  return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

- 在旋转数组查找目标值（无重复元素）

  https://leetcode.com/problems/search-in-rotated-sorted-array/

Code

```
// 每次比较 nums[mid] 与 nums[high]
class Solution
{
public:
    int search(vector<int>& nums, int target)
    {
        int n = nums.size();
        if(n == 0) return -1;
        int low = 0;
        int high = n - 1;
        while(low <= high)
        {
            int mid = low + (high - low) / 2;
            if(nums[mid] == target) return mid;

            if(nums[mid] < nums[high]) // 注：只有当 low == high，才会出现： mid == high，nums[mid] == nums[high]
            {
                if(nums[mid] < target && target <= nums[high]) low = mid + 1;
                else high = mid - 1;
            }
            else
            {
                if(nums[mid] > target && target >= nums[low]) high = mid - 1;
                else low = mid + 1;
            }
        }
        return -1;
    }
};
```

- 在旋转数组查找目标值（含重复元素）

  https://leetcode.com/problems/search-in-rotated-sorted-array-ii/

Code

```cpp
// https://www.cnblogs.com/grandyang/p/4325840.html
// 相对于上例，需要增加一个判断：如果 nums[mid] 与 nums[high] 相等，则 --high
class Solution
{
public:
    bool search(vector<int>& nums, int target)
    {
        int n = nums.size();
        if(n == 0) return false;
        int low = 0;
        int high = n - 1;
        while(low <= high)
        {
            int mid = low + (high - low) / 2;
            if(nums[mid] == target) return true;

            if(nums[mid] == nums[high]) -- high; // 增加这个判断。注：只有当 low == high，才会出现： mid == high 。

            else if(nums[mid] < nums[high])
            {
                if(nums[mid] < target && target <= nums[high]) low = mid + 1;
                else high = mid - 1;
            }
            else
            {
                if(nums[mid] > target && target >= nums[low]) high = mid - 1;
                else low = mid + 1;
            }
        }
        return false;
    }
};
```

29. [LeetCode] Maximum Gap 最大间隔。Hint：方法一，普通排序，逐个比较；方法二，桶排序。将 n 个数放到 $n+1$ 个桶中，最小值放第一个桶，最大值放最后一个桶，每个桶的大小为 $\frac{max-min}{n}$ 。根据鸽巢原理，至少存在一个桶为空。最大间隔必然出现在空桶两侧，且只与左侧桶的最大值、右侧桶的最小值有关。（事实上，可以将 n 个数放到 n 个桶中，如果没有空桶，则刚好每个桶有且仅有一个数，最大间隔出现在相邻桶中；如果某个桶有 2 个数以上，说明存在有空桶，最大间隔出现在非空的相邻桶中。总之，最大间隔不会出现在一个桶中。）

https://leetcode.com/problems/maximum-gap/

Code

```cpp
// 建立 n 个桶
class Solution
{
public:
    int maximumGap(vector<int>& nums)
    {
        size_t n = nums.size();
        if(n < 2) return 0;

        int MIN = *min_element(nums.begin(), nums.end());
        int MAX = *max_element(nums.begin(), nums.end());
        if(MIN == MAX) return 0;

        vector<vector<int>> bucket(n, vector<int>{});

        double delta = (MAX - MIN) / double(n - 1);
        for(size_t k = 0; k < n; ++k)
        {
            int idx = (nums[k] - MIN) / delta;
            bucket[idx].push_back(nums[k]);
        }

        int gap = 0;
        size_t pre = 0;
        size_t curr = 1;
        while(curr < bucket.size())
        {
            if(bucket[curr].size() == 0) curr ++;
            else
            {
                if(curr - pre >= 1)
                {
                    int pre_max = *max_element(bucket[pre].begin(), bucket[pre].end());
                    int curr_min = *min_element(bucket[curr].begin(), bucket[curr].
↪end());
                    gap = max(gap, curr_min - pre_max);
                }
                pre = curr;
                curr ++;
            }
        }
        return gap;
    }
};
```

30. 数组操作模拟大数乘法。Hint：从低位到高位，采用竖式计算，记录所有位的乘积，再将对应位的结果相加，最后进位。假设数组 a 和 b 从低位到高位存储了两个大数（可能存在小数点），则乘积为 $ans[i+j] = ans[i+j] + a[i] + b[j]$ 。

Code

```
def preProcess(a):
    ## input: str
    ## output: list, l
    pf = a.find('.')
    lf = 0
    if pf != -1:
        lf = len(a) - 1 - pf ## 小数位数
        a = a[:pf] + a[pf+1:] ## 去掉小数点
    a = list(a)
    a = a[::-1] ## 翻转数组，a[0] 表示最低位
    return a, lf

def strMul(a, b):
    a, la = preProcess(a)
    b, lb = preProcess(b)
    lf = la + lb

    ans = [0 for _ in range(len(a) + len(b))]
    for ia in range(len(a)):
        for ib in range(len(b)):
            ans[ia+ib] += int(a[ia]) * int(b[ib])
    carry = 0
    for i in range(len(ans)):
        tmp = ans[i] + carry
        ans[i] = tmp % 10
        carry = tmp / 10
    ans = ans[::-1] ## 翻转数组

    if lf > 0:
        ans.insert(len(ans) - lf, '.') ## 插入小数点
    if ans[0] == 0:
        ans = ans[1:] ## 最高位是 0 则去掉
    iz = len(ans)-1
    while lf > 0 and ans[iz] == 0: ## 去掉小数点末尾的 0
        iz -= 1

    s = ''
    for e in ans[:iz+1]:
        s += str(e)
```

(continues on next page)

(continued from previous page)

```
40
41      return s
```

31. [LeetCode] Number of Islands 孤岛个数。Hint：使用队列，广度优先遍历（BFS）。

    https://leetcode.com/problems/number-of-islands/

    Code

```
1  class Solution
2  {
3  public:
4     void traverseIsland(vector<vector<char>>& grid, int m, int n, const int M, const int␣
   ↪N)
5      {
6          queue<pair<int, int>> que;
7
8          que.push(make_pair(m, n));
9          grid[m][n] = '0';
10
11         while (!que.empty())
12         {
13             pair<int, int> p = que.front();
14             que.pop();
15
16             if (p.first - 1 >= 0 && grid[p.first - 1][p.second] == '1')
17             {
18                 grid[p.first - 1][p.second] = '0';
19                 que.push(make_pair(p.first - 1, p.second));
20             }
21             if (p.first + 1 < M && grid[p.first + 1][p.second] == '1')
22             {
23                 grid[p.first + 1][p.second] = '0';
24                 que.push(make_pair(p.first + 1, p.second));
25             }
26             if (p.second - 1 >= 0 && grid[p.first][p.second - 1] == '1')
27             {
28                 grid[p.first][p.second - 1] = '0';
29                 que.push(make_pair(p.first, p.second - 1));
30             }
31             if (p.second + 1 < N && grid[p.first][p.second + 1] == '1')
32             {
33                 grid[p.first][p.second + 1] = '0';
34                 que.push(make_pair(p.first, p.second + 1));
35             }
```

(continues on next page)

(continued from previous page)

```
        }
    }

    int numIslands(vector<vector<char>>& grid)
    {
        if(grid.size()==0) return 0;
        int M = grid.size();
        int N = grid[0].size();
        int island = 0;
        for(int m = 0; m < M; ++m)
        {
            for(int n = 0; n < N; ++n)
            {
                if(grid[m][n]=='1')
                {
                    island += 1;
                    traverseIsland(grid, m, n, M, N);
                }
            }
        }
        return island;
    }
};
```

32. 回文。

- [LeetCode] Longest Palindromic Substring 最长回文子串（子串连续）。Hint：中心扩展法，回文中心的两侧互为镜像，将每一个位置作为中心进行扩展，回文分偶数和奇数；动态规划，类似于矩阵连乘问题，逐渐增大步长。

  https://leetcode.com/problems/longest-palindromic-substring/

$$dp[i][i] = true$$

$$dp[i][j] = \begin{cases} true \\ s[i] = s[j] \ \&\& \ (i \leqslant j \leqslant i+1 \ || \ dp[i+1][j-1]) \\ false \\ else \end{cases}$$

Code

```
// 方法一，中心扩展法
class Solution {
```

(continues on next page)

(continued from previous page)

```
public:
    void Palindrome(int i, int j, string s, int& start, int& longest)
    {
        while(i >= 0 && j < s.size() && s.at(i) == s.at(j))
        {
            i--;
            j++;
        }
        i += 1;
        j -= 1;
        if(j-i+1 > longest)
        {
            longest = j-i+1;
            start = i;
        }
    }
    string longestPalindrome(string s) {
        int len = s.size();
        if(len <= 1) return s;
        int start = 0;
        int longest = 1;
        for(int i = 0; i < len-1; ++ i)
        {
            Palindrome(i, i, s, start, longest);
            Palindrome(i, i+1, s, start, longest);
        }
        string str;
        str.assign(s, start, longest);
        return str;
    }
};
```

```
// 方法二，动态规划
class Solution
{
public:
    string longestPalindrome(string s)
    {
        if(s.size() <= 1) return s;
        size_t len = s.size();
        vector<vector<bool>> dp(len, vector<bool>(len, false));
        size_t start = 0;
        size_t longest = 1;
```

(continues on next page)

(continued from previous page)

```
        for(size_t i = 0; i < len; ++i) dp[i][i] = true;
        for(size_t gap = 0; gap < len; ++ gap)
        {
            for(int i = 0; i + gap < len; ++ i)
            {
                int j = i + gap;
                if(s[i] == s[j])
                {
                    if(j - i <= 1 || dp[i+1][j-1])
                    {
                        dp[i][j] = true;
                        longest = j - i + 1; // 由于步长是逐渐增大的，因此最后得到的回文子串一定是最长的
                        start = i;
                    }
                    else dp[i][j] = false;
                }
            }
        }
        vector<vector<bool>>().swap(dp);
        return s.substr(start, longest);
    }
};
```

- [LeetCode] Longest Palindromic Subsequence 最长回文子序列（子序列可以不连续）。Hint：寻找原字符串与翻转字符串的最长公共子序列，动态规划。

  https://leetcode.com/problems/longest-palindromic-subsequence/

Code

```
class Solution
{
public:
    // 寻找字符串 str 与其翻转字符串的最长公共子序列
    int lcsLength(string& str)
    {
        int len = str.size();
        vector<vector<int>> dp(len+1, vector<int>(len+1, 0));
        for(int i = 1; i <= len; ++i)
        {
            for(int j = len - 1; j >= 0; --j) // 注意这里 j 是反向的
            {
                if(str[i-1] == str[j]) dp[i][j] = dp[i-1][j+1] + 1;
```

(continues on next page)

(continued from previous page)

```
                else dp[i][j] = max(dp[i-1][j], dp[i][j+1]);
            }
        }
        int ans = dp[len][0];
        vector<vector<int>>().swap(dp);
        return ans;
    }

    int longestPalindromeSubseq(string s)
    {
        if(s.size() <= 1) return s.size();
        return lcsLength(s);
    }
};
```

12.1.3 C++

1. 虚函数

   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

12.1.4 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

5. 迭代器和生成器

   https://www.cnblogs.com/chongdongxiaoyu/p/9054847.html

12.1.5 机器学习（深度学习）

1. 激活函数

   https://fongyq.github.io/blog/deepLearning/02_activationFunction.html

2. Batch Normalization

   https://fongyq.github.io/blog/deepLearning/03_batchnorm.html

3. 过拟合

   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/alicelmx/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

8. Support Vector Machine

   https://fongyq.github.io/blog/machineLearning/02_svm.html

9. PCA

   https://fongyq.github.io/blog/machineLearning/03_pca.html

12.1.6 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://blog.csdn.net/diamonjoy_zone/article/details/70576775

3. ResNeXt

- ResNeXt

  https://www.cnblogs.com/bonelee/p/9031639.html

- ResNeXt 算法详解

  https://blog.csdn.net/u014380165/article/details/71667916

4. R-CNN 系列

- RCNN（三）：Fast R-CNN

  https://blog.csdn.net/u011587569/article/details/52151871

- 【RCNN 系列】【超详细解析】

  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

- (Mask RCNN)——论文详解 (真的很详细)

  https://blog.csdn.net/wangdongwei0/article/details/83110305

- ROI-Align 原理理解

  https://blog.csdn.net/gusui7202/article/details/84799535

- 为什么 RCNN 用 SVM 做分类而不直接用 CNN 全连接之后 softmax 输出?

  https://www.zhihu.com/question/54117650

5. Focal Loss（样本不均衡：正/负样本数量不均衡（α），简单/困难样本数量不均衡（γ））

$$CE = -y \log y_t - (1-y)\log(1-y_t)$$

[Cross Entropy Loss]

$$FL = -y\alpha(1-y_t)^\gamma \log y_t - (1-y)(1-\alpha)y_t^\gamma \log(1-y_t)$$

[Focal Loss]

即

$$CE = \begin{cases} -\log y_t, \\ y = 1 \\ -\log(1 - y_t), \\ y = 0 \end{cases}$$

$$FL = \begin{cases} -\alpha(1 - y_t)^\gamma \log y_t, \\ y = 1 \\ -(1 - \alpha) y_t^\gamma \log(1 - y_t), \\ y = 0 \end{cases}$$

- 损失函数改进方法之 Focal Loss

  https://blog.csdn.net/sinat_24143931/article/details/79033538

- RetinaNet 论文理解

  https://blog.csdn.net/wwwhp/article/details/83317738

- Focal Loss 理解

  https://www.cnblogs.com/king-lps/p/9497836.html

6. FCN（Fully Convolutional Networks）

- FCN 学习:Semantic Segmentation

  https://zhuanlan.zhihu.com/p/22976342?utm_source=tuicool&utm_medium=referral

- FCN 于反卷积 (Deconvolution)、上采样 (UpSampling)

  https://blog.csdn.net/nijiayan123/article/details/79416764

7. FPN（Feature Pyramid Networks for Object Detection）

   https://www.cnblogs.com/fangpengchengbupter/p/7681683.html

8. CapsuleNet 解读

   https://blog.csdn.net/u013010889/article/details/78722140/

9. 轻量级网络–MobileNet 论文解读

   https://blog.csdn.net/u011974639/article/details/79199306

12.1.7 其他

1. 理解数据库的事务，ACID，CAP 和一致性

   https://www.jianshu.com/p/2c30d1fe5c4e

12.2 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。说白一点就是说:

```
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:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i 2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-1.$$

将高亮语言设置为 C

测试 C

```c
int a = 0;
char c = 'c';
printf("%c\n", c);
```

这里是 C++ :

```cpp
int main()
{
```

(continues on next page)

(continued from previous page)

```
  int i;
  int j;
  cin >> i >> j;
  cout << i << j << endl;
  return 1;
}
// 主函数注释
```

斜体 *text*

将高亮语言设置为 python

测试 python

```
import torch
import numpy as np
print "hello world"
```

这里是 python (code):

```
def foo():
    print "Love Python, Love FreeDome"
    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:

```
# 测试注释
def foo():
    print "Love Python, Love FreeDome"
    print "E 文标点,.0123456789, 中文标点,. "
```

下面是 javescipt 的 rst 源码:

```
.. code-block:: javascript
    :linenos:

    function whatever()
    {
        return "such color"
    }
```

下面是 bash :

```
cd home
echo $PATH
source ~/.bashrc
ls -l
mkdir filefolder
cd ..
```

下面是 python (code-block):

```
class Solution(object):
    def jump_from_i(self, nums, i):
        if i == len(nums) - 1:
            return True
        max_step = min(len(nums), i + nums[i] + 1)
        for t in range(i+1, max_step):
            if self.jump_from_i(nums, t):
                return True
        return False
    def canJump(self, nums):
        """
        https://leetcode.com/problems/jump-game/
        Each element in the array represents your maximum jump length at that position.

        Input: [2,3,1,1,4]
        Output: true
        Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

        :type nums: List[int]
        :rtype: bool
        """
        if nums == []:
            return False
        if len(nums) == 1:
            return True
        return self.jump_from_i(nums, 0)
```

代码显示与隐藏:

Show/Hide Code

Show/Hide Code

```
from plone import api
...
```

插入空行使用 | 。

上面是两个空行。

12.2.1 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://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

https://github.com/rtfd/sphinx_rtd_theme

6. latex 颜色

   http://latexcolor.com/