# C++

## 三大特性

### 封装

定义：封装就是将抽象得到的数据和行为相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成**类**，其中数据和函数都是类的成员，目的在于将对象的使用者和设计者分开，以提高软件的**可维护性**和**可修改性**

　　特性：1. 结合性，即是将属性和方法结合    2. 信息隐蔽性，利用接口机制隐蔽内部实现细节，只留下接口给外界调用    3. 实现代码重用

### 继承

定义：继承就是新类从已有类那里得到已有的特性。 类的派生指的是从已有类产生新类的过程。原有的类成为基类或父类，产生的新类称为派生类或子类，子类继承基类后，可以创建子类对象来调用基类函数，变量等。

**单一继承：**继承一个父类，这种继承称为单一继承，一般情况尽量使用单一继承，使用多重继承容易造成混乱易出问题

**多重继承：**继承多个父类，类与类之间要用逗号隔开，类名之前要有继承权限，假使两个或两个基类都有某变量或函数，在子类中调用时需要加类名限定符如c.a::i = 1；

**菱形继承：**多重继承掺杂隔代继承1-n-1模式，此时需要用到虚继承，例如 B，C虚拟继承于A，D再多重继承B，C，否则会出错

**继承权限：**继承方式规定了如何访问继承的基类的成员。继承方式指定了派生类成员以及类外对象对于从基类继承来的成员的访问权限

**继承权限：**子类继承基类除构造和析构函数以外的所有成员

继承可以扩展已存在的代码，目的也是为了代码重用

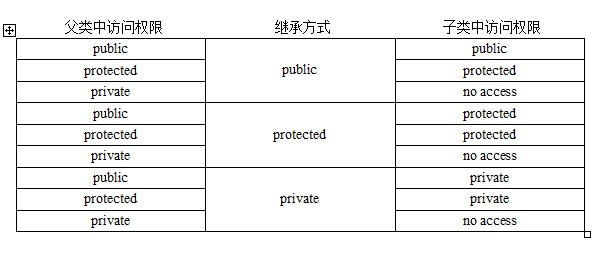
继承也分为接口继承和实现继承：

普通成员函数的接口总是会被继承：　　子类继承一份接口和一份强制实现

普通虚函数被子类重写　　　　　：　　子类继承一份接口和一份缺省实现

纯虚函数只能被子类继承接口　　：　　子类继承一份接口，没有继承实现

访问权限图如下：



### 多态

定义：可以简单概括为“一个接口，多种方法”，即用的是同一个接口，但是效果各不相同，多态有两种形式的多态，一种是静态多态，一种是动态多态

**动态多态：**是指在程序运行时才能确定函数和实现的链接，此时才能确定调用哪个函数，父类指针或者引用能够指向子类对象，调用子类的函数，所以在编译时是无法确定调用哪个函数

　　使用时在父类中写一个虚函数，在子类中分别重写，用这个父类指针调用这个虚函数，它实际上会调用各自子类重写的虚函数。

　　运行期多态的设计思想要归结到类继承体系的设计上去。对于有相关功能的对象集合，我们总希望能够抽象出它们共有的功能集合，在基类中将这些功能声明为虚接口（**虚函数**），然后由子类继承基类去重写这些虚接口，以实现子类特有的具体功能。

　　运行期多态的实现依赖于虚函数机制。**当某个类声明了虚函数时，编译器将为该类对象安插一个虚函数表指针，并为该类设置一张唯一的虚函数表，虚函数表中存放的是该类虚函数地址。**运行期间通过虚函数表指针与虚函数表去确定该类虚函数的真正实现。

**优点：** OO设计重要的特性，对客观世界直觉认识； 能够处理同一个继承体系下的异质类集合

　　　　　　　　　　　　　　vector<Animal\*>anims;

　　　　　　　　　　　　　　Animal \* anim1 = new Dog;

　　　　　　　　　　　　　　Animal \* anim2 = new Cat;

　　　　　　　　　　　　　　 //处理异质类集合

　　　　　　　　　　　　　　anims.push\_back(anim1);

　　　　　　　　　　　　　　anims.push\_back(anim2);

**缺点：**运行期间进行虚函数绑定，提高了程序运行开销；庞大的类继承层次，对接口的修改易影响类继承层次；**由于虚函数在运行期才绑定，所以编译器无法对虚函数进行优化**

**静态多态：**是在编译期就把函数链接起来，此时即可确定调用哪个函数或模板，静态多态是由**模板**和**重载**实现的，在宏多态中，是通过定义变量，编译时直接把变量替换，实现宏多态

**优点：** 带来了泛型编程的概念，使得C++拥有泛型编程与STL这样的武器； 在编译期完成多态，提高运行期效率； 具有很强的适配性和松耦合性，（耦合性指的是两个功能模块之间的依赖关系）

**缺点：** 程序可读性降低，代码调试带来困难；无法实现模板的分离编译，当工程很大时，编译时间不可小觑 ；无法处理异质对象集合

**调用基类指针创建子类对象，那么基类应该有虚析构函数，**因为如果基类没有虚析构函数，那么在删除这个子类对象的时候会调用错误的析构函数而导致删除失败产生不明确行为，

　　　　　　　　　　int main() {

　　　　　　　　　　　　Base \*p = new Derive();　　　　//调用基类指针创建子类对象，那么基类应有虚析构函数，不然当删除的时候会调用错误的析构函数而导致删除失败产生不明确行为，

　　　　　　　　　　　　delete p;　　　　　　　　　　　　//删除子类对象时，如果基类有虚析构函数，那么delete时会先调用子类的析构函数，然后再调用基类的析构函数，成功删除

　　　　　　　　　　　　return 0;　　　　　　　　　　　　//如果基类没有虚析构函数，那么就只会调用父类的析构函数，只删除了对象内的父类部分，造成一个局部销毁，可能导致资源泄露

　　　　　　　　　　}　　　　　　　　　　　　　　　　　　//注：只有当此类希望成为 基类时才会打算声明一个虚析构函数，否则不必要给此类声明一个虚函数

#### **虚函数**

**定义：**用virtual关键字修饰的函数，**本质：**由虚指针和虚表控制，虚指针指向虚表中的某个函数入口地址，就实现了多态，**作用：**实现了多态，虚函数可以被子类重写，虚函数地址存储在虚表中

**虚表：**虚表中主要是一个类的虚函数的地址表，这张表解决了**继承，覆盖**的问题，保证其真实反应实际的函数，当我们用父类指针来指向一个子类对象的时候，虚表指明了实际所应调用的函数

　　基类有一个虚表，可以被子类继承，(当类中有虚函数时该类才会有虚表,该类的对象才有虚指针，子类继承时也会继承基类的虚表)，子类如果重写了基类的某虚函数，那么子类继承于基类的虚表中该虚函数的地址也会相应改变，指向子类自身的该虚函数实现，如果子类有自己的虚函数，那么子类的虚表中就会增加该项，编译器为每个类对象定义了一个虚指针，来定位虚表，所以虽然是父类指针指向子类对象，但因为此时子类重写了该虚函数,该虚函数地址在子类虚表中的地址已经被改变了，所以它实际调用的是子类的重写后的函数，正是由于每个对象调用的虚函数都是通过虚表指针来索引的,也就决定了虚表指针的正确初始化是非常重要的，即是说，在虚表指针没有正确初始化之前，我们是不能调用虚函数的，因为生成一个对象是构造函数的工作，所以**设置虚指针也是构造函数的工作**，编译器在构造函数的开头部分秘密插入能初始化虚指针的代码， **在构造函数中进行虚表的创建和虚指针的初始化。**

　　一但虚指针被初始化为指向相应的虚表，对象就“知道”它自己是什么类型，但只有当虚函数被调用时这种自我认知才有用。

**类中若没有虚函数，类对象的大小正好是数据成员的大小，包含有一个或者多个虚函数的类对象。编译器会向里面插入一个虚指针，指向虚表**，这些都是编译器为我们做的，我们完全不必关心这些，所有有虚函数的类对象的大小是数据成员的大小加一个虚指针的大小；**对于虚继承**，若子类也有自己的虚函数，则它本身需要有一个虚指针，指向自己的虚表，另外子类继承基类时，首先要通过加入一个虚指针来指向基类，因此可能会有两个或多个虚指针（多重继承会多个），其他情况一般是一个虚指针，一张虚表。

　　每一个带有virtual函数的类都有一个相应的虚表，当对象调用某一virtual函数时，实际被调用的函数取决于该对象的虚指针所指向的那个虚表-编译器在其中寻找适当的函数指针。

**效率漏洞：**我们必须明白，编译器正在插入隐藏代码到我们的构造函数中，这些隐藏代码不仅必须初始化虚指针，而且还必须检查this的值(以免operator new返回零)和调用基类构造函数。放在一起，这些代码可以影响我们认为是一个小内联函数的调用，特别是，构造函数的规模会抵消函数调用代价的减少，如果做大量的内联函数调用，代码长度就会增长，而在速度上没有任何好处，当然，也许不会立即把所有这些小构造函数都变成非内联，因为它们更容易写为内联构造函数，但是，当我们正在调整我们的代码时，请务必去掉这些内联构造函数。

**虚函数使用：**将函数声明为虚函数会降低效率，一般函数在编译期其相对地址是确定的，编译器可以直接生成imp/invoke指令，如果是虚函数，那么函数的地址是动态的，譬如取到的地址在eax寄存器里，则在call eax之后的那些已经被预取到流水线的所有指令都将失效， 流水线越长，那么一次分支预测失败的代价越大，建议若不打算让某类成为基类，那么类中最好不要出现虚函数。

**纯虚函数：**含有至少一个纯虚函数的类叫**抽象类**，因为抽象类含有纯虚函数，所以其虚表是不健全的，**在虚表不健全的情况下是不能实例化对象的**，子类继承抽象基类后必须重写基类的所有纯虚函数否则子类仍为纯虚函数子类将抽象基类的纯虚函数全部重写后会将虚表完善，此时子类才能实例化对象，纯虚函数只声明不定义，形如 virtual void print() = 0

## C++函数

### 构造函数

#### 构造函数是干什么的

该类对象被创建时，编译系统对象分配内存空间，并自动调用该构造函数，由构造函数完成成员的初始化工作，故：构造函数的作用：初始化对象的数据成员。

#### 构造函数的种类

1 class Complex

2 {

3

4 private :

5 double m\_real;

6 double m\_imag;

7

8 public:

9

10 // 无参数构造函数

11 // 如果创建一个类你没有写任何构造函数,则系统会自动生成默认的无参构造函数，函数为空，什么都不做

12 // 只要你写了一个下面的某一种构造函数，系统就不会再自动生成这样一个默认的构造函数，如果希望有一个这样的无参构造函数，则需要自己显示地写出来

13 Complex(void)

14 {

15 m\_real = 0.0;

16 m\_imag = 0.0;

17 }

18

19 // 一般构造函数（也称重载构造函数）

20 // 一般构造函数可以有各种参数形式,一个类可以有多个一般构造函数，前提是参数的个数或者类型不同（基于c++的重载函数原理）

21 // 例如：你还可以写一个 Complex( int num)的构造函数出来

22 // 创建对象时根据传入的参数不同调用不同的构造函数

23 Complex(double real, double imag)

24 {

25 m\_real = real;

26 m\_imag = imag;

27 }

28

29 // 复制构造函数（也称为拷贝构造函数）

30 // 复制构造函数参数为类对象本身的引用，用于根据一个已存在的对象复制出一个新的该类的对象，一般在函数中会将已存在对象的数据成员的值复制一份到新创建的对象中

31 // 若没有显示的写复制构造函数，则系统会默认创建一个复制构造函数，但当类中有指针成员时，由系统默认创建该复制构造函数会存在风险，具体原因请查询有关 “浅拷贝” 、“深拷贝”的文章论述

32 Complex(const Complex & c)

33 {

34 // 将对象c中的数据成员值复制过来

35 m\_real = c.m\_real;

36 m\_img = c.m\_img;

37 }

38

39 // 类型转换构造函数，根据一个指定的类型的对象创建一个本类的对象

40 // 例如：下面将根据一个double类型的对象创建了一个Complex对象

41 Complex::Complex(double r)

42 {

43 m\_real = r;

44 m\_imag = 0.0;

45 }

46

47 // 等号运算符重载

48 // 注意，这个类似复制构造函数，将=右边的本类对象的值复制给等号左边的对象，它不属于构造函数，等号左右两边的对象必须已经被创建

49 // 若没有显示的写=运算符重载，则系统也会创建一个默认的=运算符重载，只做一些基本的拷贝工作

50 Complex &operator=(const Complex &rhs)

51 {

52 // 首先检测等号右边的是否就是左边的对象本，若是本对象本身,则直接返回

53 if ( this == &rhs )

54 {

55 return \*this;

56 }

57

58 // 复制等号右边的成员到左边的对象中

59 this->m\_real = rhs.m\_real;

60 this->m\_imag = rhs.m\_imag;

61

62 // 把等号左边的对象再次传出

63 // 目的是为了支持连等 eg: a=b=c 系统首先运行 b=c

64 // 然后运行 a= ( b=c的返回值,这里应该是复制c值后的b对象)

65 return \*this;

66 }

67 };

下面使用上面定义的类对象来说明各个构造函数的用法：

[复制代码](javascript:void(0);)

1 void main()

2 {

3 // 调用了无参构造函数，数据成员初值被赋为0.0

4 Complex c1，c2;

5

6 // 调用一般构造函数，数据成员初值被赋为指定值

7 Complex c3(1.0,2.5);

8 // 也可以使用下面的形式

9 Complex c3 = Complex(1.0,2.5);

10

11 // 把c3的数据成员的值赋值给c1

12 // 由于c1已经事先被创建，故此处不会调用任何构造函数

13 // 只会调用 = 号运算符重载函数

14 c1 = c3;

15

16 // 调用类型转换构造函数

17 // 系统首先调用类型转换构造函数，将5.2创建为一个本类的临时对象，然后调用等号运算符重载，将该临时对象赋值给c1

18 c2 = 5.2;

19

20 // 调用拷贝构造函数( 有下面两种调用方式)

21 Complex c5(c2);

22 Complex c4 = c2; // 注意和 = 运算符重载区分,这里等号左边的对象不是事先已经创建，故需要调用拷贝构造函数，参数为c2

23

24 }

#### 复制构造函数

几个原则：

C++ primer p406 ：复制构造函数是一种特殊的构造函数，具有单个形参，该形参（常用const修饰）是对该类类型的引用。当定义一个新对象并用一个同类型的对象对它进行初始化时，将显示使用复制构造函数。当该类型的对象传递给函数或从函数返回该类型的对象时，将隐式调用复制构造函数。

 C++支持两种初始化形式：复制初始化（int a = 5;）和直接初始化（int a(5);）对于其他类型没有什么区别，对于类类型直接初始化直接调用实参匹配的构造函数，复制初始化总是调用复制构造函数，也就是说：

A x(2);　　//直接初始化，调用构造函数  
A y = x;　　//复制初始化，调用复制构造函数

 必须定义复制构造函数的情况：

只包含类类型成员或内置类型（但不是指针类型）成员的类，无须显式地定义复制构造函数也可以复制；有的类有一个数据成员是指针，或者是有成员表示在构造函数中分配的其他资源，这两种情况下都必须定义复制构造函数。

什么情况使用复制构造函数：

类的对象需要拷贝时，拷贝构造函数将会被调用。以下情况都会调用拷贝构造函数：  
（1）一个对象以值传递的方式传入函数体   
（2）一个对象以值传递的方式从函数返回   
（3）一个对象需要通过另外一个对象进行初始化。

#### 深拷贝和浅拷贝

所谓浅拷贝，指的是在对象复制时，只对对象中的数据成员进行简单的赋值，默认拷贝构造函数执行的也是浅拷贝。在“深拷贝”的情况下，对于对象中动态成员，就不能仅仅简单地赋值了，而应该重新动态分配空间

如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配，这个过程就是深拷贝

上面提到，如果没有自定义复制构造函数，则系统会创建默认的复制构造函数，但系统创建的默认复制构造函数只会执行“浅拷贝”，即将被拷贝对象的数据成员的值一一赋值给新创建的对象，若该类的数据成员中有指针成员，则会使得新的对象的指针所指向的地址与被拷贝对象的指针所指向的地址相同，delete该指针时则会导致两次重复delete而出错。下面是示例：

[复制代码](javascript:void(0);)

1 #include <iostream.h>

2 #include <string.h>

3 class Person

4 {

5 public :

6

7 // 构造函数

8 Person(char \* pN)

9 {

10 cout << "一般构造函数被调用 !\n";

11 m\_pName = new char[strlen(pN) + 1];

12 //在堆中开辟一个内存块存放pN所指的字符串

13 if(m\_pName != NULL)

14 {

15 //如果m\_pName不是空指针，则把形参指针pN所指的字符串复制给它

16 strcpy(m\_pName ,pN);

17 }

18 }

19

20 // 系统创建的默认复制构造函数，只做位模式拷贝

21 Person(Person & p)

22 {

23 //使两个字符串指针指向同一地址位置

24 m\_pName = p.m\_pName;

25 }

26

27 ~Person( )

28 {

29 delete m\_pName;

30 }

31

32 private :

33 char \* m\_pName;

34 };

35

36 void main( )

37 {

38 Person man("lujun");

39 Person woman(man);

40

41 // 结果导致 man 和 woman 的指针都指向了同一个地址

42

43 // 函数结束析构时

44 // 同一个地址被delete两次

45 }

46

47

48 // 下面自己设计复制构造函数，实现“深拷贝”，即不让指针指向同一地址，而是重新申请一块内存给新的对象的指针数据成员

49 Person(Person & chs);

50 {

51 // 用运算符new为新对象的指针数据成员分配空间

52 m\_pName=new char[strlen(p.m\_pName)+ 1];

53

54 if(m\_pName)

55 {

56 // 复制内容

57 strcpy(m\_pName ,chs.m\_pName);

58 }

59

60 // 则新创建的对象的m\_pName与原对象chs的m\_pName不再指向同一地址了

61 }

#### 重载赋值操作符

通过定义operate=的函数，可以对赋值进行定义。像其他任何函数一样，操作符函数有一个返回值和形参表。形参表必须具有与该操作符操作数书目相同的形参（如果操作符是一个成员，则包括隐式this形参）。赋值是二元运算，所以该操作符函数有两个形参：第一个形参（隐含的this指针）对应着左操作数，第二个形参对应右操作数。

 一个应用了对赋值号重载的拷贝构造函数的例子：

[复制代码](javascript:void(0);)

1 #include <iostream>

2

3 using namespace std;

4

5 class A

6 {

7 public:

8 A(int);//构造函数

9 A(const A &);//拷贝构造函数

10 ~A();

11 void print();

12 int \*point;

13 A &operator=(const A &);

14 };

15

16 A::A(int p)

17 {

18 point = new int;

19 \*point = p;

20 }

21

22 A::A(const A &b)

23 {

24 \*this = b;

25 cout<<"调用拷贝构造函数"<<endl;

26 }

27

28 A::~A()

29 {

30 delete point;

31 }

32

33 void A::print()

34 {

35 cout<<"Address:"<<point<<" value:"<<\*point<<endl;

36 }

37

38 A &A::operator=(const A &b)

39 {

40 if( this != &b)

41 {

42 delete point;

43 point = new int;

44 \*point = \*b.point;

45 }

46 }

47

48

49 int main()

50 {

51 A x(2);

52 A y = x;

53 x.print();

54 delete x.point;

55 y.print();

56

57 return 0;

58 }

#### 关于构造函数私有的问题

构造函数私有只是不想在外部直接定义对象出来，并不是说不可以定义对象。**考虑实现一个单例类的例子：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class A{  public:      void show( ){          cout<<val<<endl;      }      static A\* GetInstance( )      {          static A instance;          return &instance;      }  private:      A() { }      static int val;  };  int A::val = 0; |

### 析构函数

#### 定义

作用：对象消亡时，自动被调用，用来释放对象占用的空间  
特点:  
  1.函数名是在类名前加上~，无参数且无返回值。

2.一个类只能有且有一个析构函数，如果没有显式的定义，系统会生成一个缺省的析构函数（合成析构函数）。

3.析构函数不能重载。每有一次构造函数的调用就会有一次析构函数的调用。

在构造函数中，成员的在初始化是在函数体执行前完成的，并按照成员在类中出现的顺序进行初始化，而在析构函数中，首先执行函数体，然后再销毁成员，并且成员按照初始化的逆序进行销毁。

我们一直在说析构函数的作用是在你的类对象离开作用域后释放对象使用的资源，并销毁成员。那么到底这里所说的销毁到底是什么？那么继续往下看：

void test ()

{

int a=10;

int b=20;

}

回想我们在一个函数体内定义一个变量的情况，在test函数中定义了a和b两个变量，那么在出这个函数之后，a和b就会被销毁（栈上的操作）。那么如果是是一个指向动态开辟的一块空间的指针，我们都知道需要自己进行free，否则会造成内存泄漏。

说到这里，其实在类里面的情况和这是一样的，这就是合成析构函数体为空的原因，函数并不需要做什么，当类对象出作用域时系统会释放你的内置类型的那些成员。但是像上面说的一样，如果，我的成员里有一个指针变量并且指向了一块你动态开辟的内存，那么像以前那样也需要自己来释放，此时就需要在析构函数内部写你的释放代码，这样在调用析构函数的时候就可以把你所有的资源进行释放。（其实这才是析构函数有用的地方，对吗）

那么还有一点，当类类型对象的成员还有一个类类型对象，那么在析构函数里也会调用这个对象的析构函数。

#### 析构函数来阻止该类型对象被销毁

我们如果不想要析构函数来对对象进行释放该怎么做呢，不显式的定义显然是不行的，因为编译器会生成默认的合成析构函数。之前我们知道了如果想让系统默认生成自己的构造函数可以利用default，那么其实还有一个东西叫做delete。

class Date

{

public:

Date(int year=1990,int month=1,int day=1)

: \_year(year),\_month(month), \_day(day)

{ }

~Date() = delete;

private:

int \_year=1990;

int \_month;

int \_day;

};

如果我这么写了，又在底下创建Date类型的对象，那么这个对象将是无法被销毁的，其实编译器并不允许这么做，直接会给我们报错。

但其实是允许我们动态创建这个类类型对象的，像这样：Date\* p = new Date;虽然这样是可行的，但当你delete p的时候依然会出错。

### 友元

类的友元函数是定义在类外部，但有权访问类的所有私有（private）成员和保护（protected）成员。尽管友元函数的原型有在类的定义中出现过，但是友元函数并不是成员函数。

友元函数不能被继承；友元函数没有this指针；友元函数破坏了类的封装性。

#include <iostream>

using namespace std;

class Box

{

double width;

public:

friend void printWidth( Box box );

void setWidth( double wid );

};

// 成员函数定义

void Box::setWidth( double wid )

{

width = wid;

}

// 请注意：printWidth() 不是任何类的成员函数

void printWidth( Box box )

{

/\* 因为 printWidth() 是 Box 的友元，它可以直接访问该类的任何成员 \*/

cout << "Width of box : " << box.width <<endl;

}

// 程序的主函数

int main( )

{

Box box;

// 使用成员函数设置宽度

box.setWidth(10.0);

// 使用友元函数输出宽度

printWidth( box );

return 0;

}

当上面的代码被编译和执行时，它会产生下列结果：

Width of box : 10

#### **友元函数的使用**

因为友元函数没有this指针，则参数要有三种情况：

1.要访问非static成员时，需要对象做参数；

2.要访问static成员或全局变量时，则不需要对象做参数；

3.如果做参数的对象是全局对象，则不需要对象做参数.

4.可以直接调用友元函数，不需要通过对象或指针

### Lambda表达式

#### 1. 概述

C++ 11 中的 Lambda 表达式用于定义并创建匿名的函数对象，以简化编程工作。  
Lambda 的语法形式如下：

[函数对象参数] (操作符重载函数参数) mutable 或 exception 声明 -> 返回值类型 {函数体}

可以看到，Lambda 主要分为五个部分：[函数对象参数]、(操作符重载函数参数)、mutable 或 exception 声明、-> 返回值类型、{函数体}.

#### 2. Lambda 语法分析

##### 2.1 [函数对象参数]

标识一个 Lambda 表达式的开始，这部分必须存在，不能省略。函数对象参数是传递给编译器自动生成的函数对象类的构造  
函数的。函数对象参数只能使用那些到定义 Lambda 为止时 Lambda 所在作用范围内可见的局部变量(包括 Lambda 所在类  
的 this)。函数对象参数有以下形式：

* 空。没有任何函数对象参数。
* =。函数体内可以使用 Lambda 所在范围内所有可见的局部变量（包括 Lambda 所在类的 this），并且是值传递方式（相  
  当于编译器自动为我们按值传递了所有局部变量）。
* &。函数体内可以使用 Lambda 所在范围内所有可见的局部变量（包括 Lambda 所在类的 this），并且是引用传递方式  
  （相当于是编译器自动为我们按引用传递了所有局部变量）。
* this。函数体内可以使用 Lambda 所在类中的成员变量。
* a。将 a 按值进行传递。按值进行传递时，函数体内不能修改传递进来的 a 的拷贝，因为默认情况下函数是 const 的，要  
  修改传递进来的拷贝，可以添加 mutable 修饰符。
* &a。将 a 按引用进行传递。
* a，&b。将 a 按值传递，b 按引用进行传递。
* =，&a，&b。除 a 和 b 按引用进行传递外，其他参数都按值进行传递。
* &，a，b。除 a 和 b 按值进行传递外，其他参数都按引用进行传递。

##### 2.2 (操作符重载函数参数)

标识重载的 () 操作符的参数，没有参数时，这部分可以省略。参数可以通过按值（如: (a, b)）和按引用 (如: (&a, &b)) 两种  
方式进行传递。

##### 2.3 mutable 或 exception 声明

这部分可以省略。按值传递函数对象参数时，加上 mutable 修饰符后，可以修改传递进来的拷贝（注意是能修改拷贝，而不是  
值本身）。exception 声明用于指定函数抛出的异常，如抛出整数类型的异常，可以使用 throw(int)。

##### 2.4 -> 返回值类型

标识函数返回值的类型，当返回值为 void，或者函数体中只有一处 return 的地方（此时编译器可以自动推断出返回值类型）  
时，这部分可以省略。

##### 2.5 {函数体}

标识函数的实现，这部分不能省略，但函数体可以为空。

#### 3. 示例

[] (int x, int y) { return x + y; } // 隐式返回类型

[] (int& x) { ++x; } // 没有 return 语句 -> Lambda 函数的返回类型是 'void'

[] () { ++global\_x; } // 没有参数，仅访问某个全局变量

[] { ++global\_x; } // 与上一个相同，省略了 (操作符重载函数参数)

可以像下面这样显示指定返回类型：

[] (int x, int y) -> int { int z = x + y; return z; }

在这个例子中创建了一个临时变量 z 来存储中间值。和普通函数一样，这个中间值不会保存到下次调用。什么也不返回的  
Lambda 函数可以省略返回类型，而不需要使用 -> void 形式。

Lambda 函数可以引用在它之外声明的变量. 这些变量的集合叫做一个闭包. 闭包被定义在 Lambda 表达式声明中的方括  
号 [] 内。这个机制允许这些变量被按值或按引用捕获。如下图的例子：  


##### 3.1 示例 1

std::vector<int> some\_list;

int total = 0;

for (int i = 0; i < 5; ++i) some\_list.push\_back(i);

std::for\_each(begin(some\_list), end(some\_list), [&total](int x)

{

total += x;

});

此例计算 list 中所有元素的总和。变量 total 被存为 Lambda 函数闭包的一部分。因为它是栈变量（局部变量）total 引  
用，所以可以改变它的值。

##### 3.2 示例 2

std::vector<int> some\_list;

int total = 0;

int value = 5;

std::for\_each(begin(some\_list), end(some\_list), [&, value, this](int x)

{

total += x \* value \* this->some\_func();

});

此例中 total 会存为引用, value 则会存一份值拷贝。对 this 的捕获比较特殊，它只能按值捕获。this 只有当包含它的最靠近  
它的函数不是静态成员函数时才能被捕获。对 protect 和 private 成员来说，这个 Lambda 函数与创建它的成员函数有相同  
的访问控制。如果 this 被捕获了，不管是显式还是隐式的，那么它的类的作用域对 Lambda 函数就是可见的。访问 this 的  
成员不必使用 this-> 语法，可以直接访问。

#### 4. 总结

不同编译器的具体实现可以有所不同，但期望的结果是: 按引用捕获的任何变量，Lambda 函数实际存储的应该是这些变量在  
创建这个 Lambda 函数的函数的栈指针，而不是 Lambda 函数本身栈变量的引用。不管怎样，因为大多数 Lambda 函数都  
很小且在局部作用中，与候选的内联函数很类似，所以按引用捕获的那些变量不需要额外的存储空间。

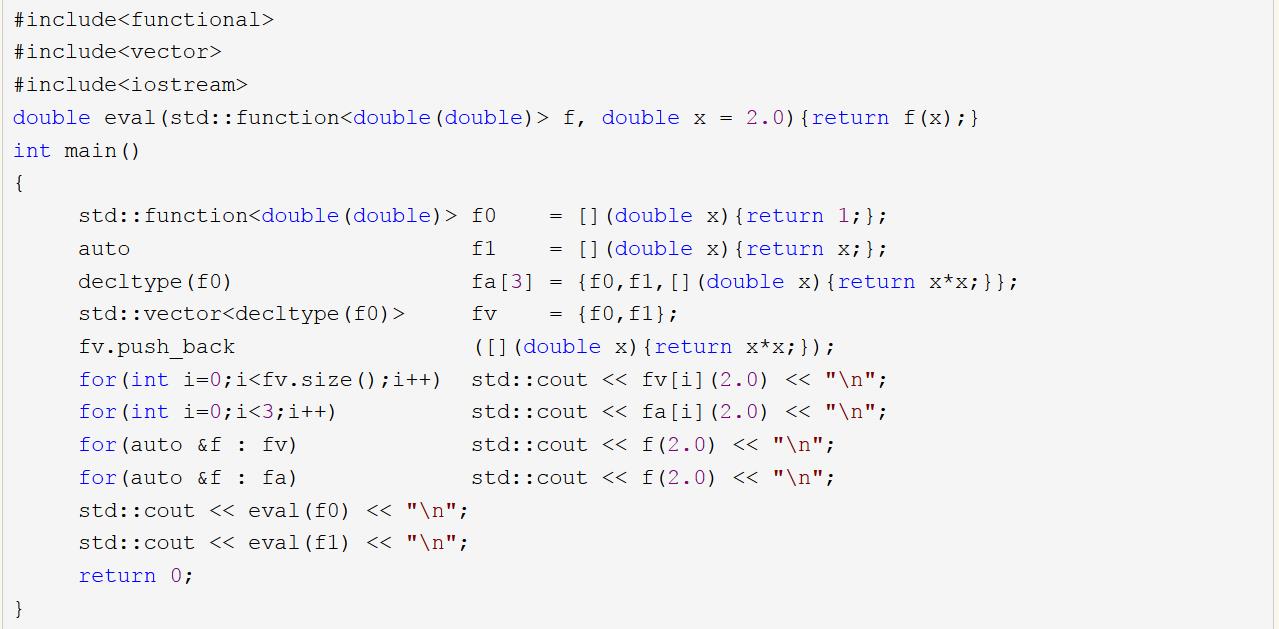
如果一个闭包含有局部变量的引用，在超出创建它的作用域之外的地方被使用的话，这种行为是未定义的!

Lambda 函数是一个依赖于实现的函数对象类型,这个类型的名字只有编译器知道. 如果用户想把 lambda 函数做为一个参数来  
传递, 那么形参的类型必须是模板类型或者必须能创建一个 std::function 类似的对象去捕获 lambda 函数.使用 auto 关键字  
可以帮助存储 lambda 函数,

auto my*\_lambda\_*func = [&](int x) { /\* ... \*/ };

auto my*\_onheap\_*lambda\_func = new auto([=](int x) { /\* ... \*/ });

这里有一个例子, 把匿名函数存储在变量、数组或 vector 中,并把它们当做命名参数来传递



一个没有指定任何捕获的 lambda 函数,可以显式转换成一个具有相同声明形式函数指针.所以,像下面这样做是合法的:

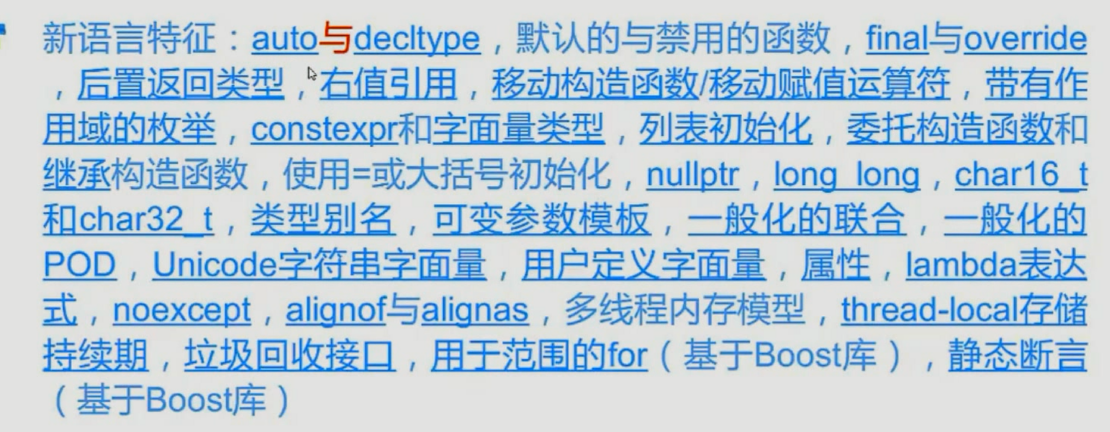
auto a\_lambda\_func = [](int x) { /\* ... \*/ };

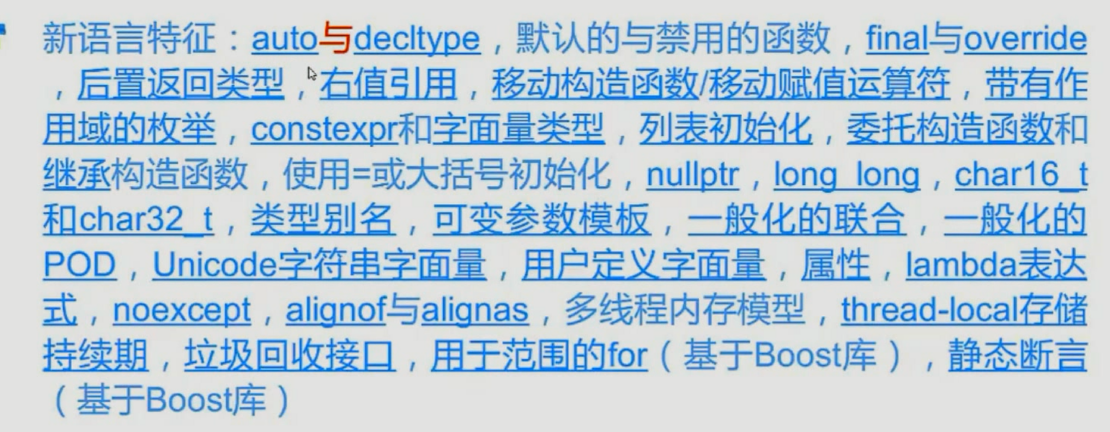
void (\*func\_ptr)(int) = a\_lambda\_func;

func\_ptr(4); // calls the lambda

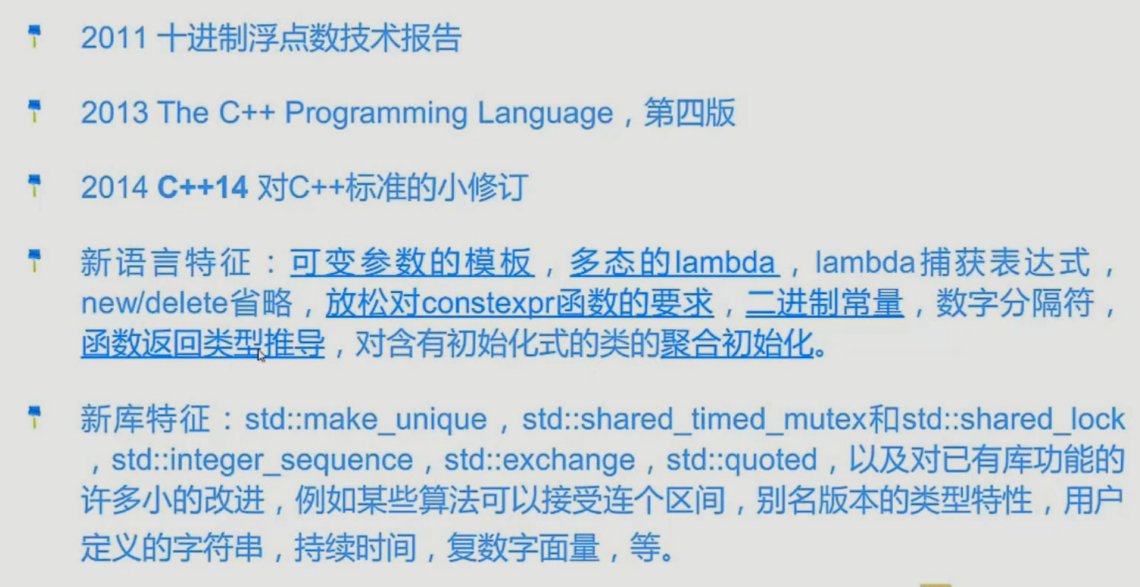
## C++11/14新特性

### c++11





### C++14



# Python3

应用：后台开发（django），脚本编写，机器学习，数据测试。

## 脚本编写

在实际工作中，遇到了一些问题需要快速生成文件，打印测试报告以及序列化的操作等需求。需要用到脚本语言来辅助完成，这样可以极大的提高编码效率。但是不同操作系统用到的脚本语言不同，linux系统中的.sh，windows系统中的批处理文件。仅linux shell就有许多种，常用的为bash shell。而python的出现使得我们有了一个平台无关的编写脚本文件的手段。

利用python内置的文件访问api或是os模块可以轻松进行文件操作，还有sys模块。

## 后台开发

使用django框架可以快速的进行后台开发。

## 机器学习

根据GPU的型号，安装兼容的tensorflow深度学习框架。

## 数据测试

使用faker,random模块产生大量假数据进行测试；使用requests模块编写网络爬虫，从网上爬取大量数据。

# Linux

# 数据库

## 关系型数据库

**关系型数据库：指采用了关系模型来组织数据的数据库。**  
**关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。**

### 关系模型中常用的概念

关系：一张二维表，每个关系都具有一个关系名，也就是表名  
元组：二维表中的一行，在数据库中被称为记录  
属性：二维表中的一列，在数据库中被称为字段  
域：属性的取值范围，也就是数据库中某一列的取值限制  
关键字：一组可以唯一标识元组的属性，数据库中常称为主键，由一个或多个列组成  
关系模式：指对关系的描述。其格式为：关系名(属性1，属性2， ... ... ，属性N)，在数据库中成为表结构

### 关系型数据库的优点

**1.容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解  
2.使用方便：通用的SQL语言使得操作关系型数据库非常方便  
3.易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率**

### 关系型数据库存在的问题

**1.网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈  
2.网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询，效率是非常低的  
3.在基于web的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像web server和app server那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。当需要对数据库系统进行升级和扩展时，往往需要停机维护和数据迁移。  
4.性能欠佳：在关系型数据库中，导致性能欠佳的最主要原因是多表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询。为了保证数据库的ACID特性，必须尽量按照其要求的范式进行设计，关系型数据库中的表都是存储一个格式化的数据结构。**

### ****ACID特性****

**数据库事务必须具备ACID特性，ACID分别是Atomic原子性，Consistency一致性，**  
**Isolation隔离性，Durability持久性。**

**原子性**

**整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。**

**一致性**

**一个事务可以封装状态改变（除非它是一个只读的）。事务必须始终保持系统处于一致的状态，不管在任何给定的时间并发事务有多少。**

**也就是说：如果事务是并发多个，系统也必须如同串行事务一样操作。其主要特征是保护性和不变性(Preserving an Invariant)，以转账案例为例，假设有五个账户，每个账户余额是100元，那么五个账户总额是500元，如果在这个5个账户之间同时发生多个转账，无论并发多少个，比如在A与B账户之间转账5元，在C与D账户之间转账10元，在B与E之间转账15元，五个账户总额也应该还是500元，这就是保护性和不变性。**

**隔离性**

**隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据。**

**持久性**

**在事务完成以后，该事务对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。**

**由于一项操作通常会包含许多子操作，而这些子操作可能会因为硬件的损坏或其他因素产生问题，要正确实现ACID并不容易。ACID建议数据库将所有需要更新以及修改的资料一次操作完毕，但实际上并不可行。**

**目前主要有两种方式实现ACID：第一种是Write ahead logging，也就是日志式的方式(现代数据库均基于这种方式)。第二种是Shadow paging。**

**相对于WAL（write ahead logging）技术，shadow paging技术实现起来比较简单，消除了写日志记录的开销恢复的速度也快(不需要redo和undo)。shadow paging的缺点就是事务提交时要输出多个块，这使得提交的开销很大，而且以块为单位，很难应用到允许多个事务并发执行的情况——这是它致命的缺点。**

**WAL 的中心思想是对数据文件的修改（它们是表和索引的载体）必须是只能发生在这些修改已经记录了日志之后 -- 也就是说，在日志记录冲刷到永久存储器之后。如果我们遵循这个过程，那么我们就不需要在每次事务提交的时候都把数据页冲刷到磁盘，因为我们知道在出现崩溃的情况下，我们可以用日志来恢复数据库：任何尚未附加到数据页的记录 都将先从日志记录中重做（这叫向前滚动恢复，也叫做 REDO） 然后那些未提交的事务做的修改将被从数据页中删除 （这叫向后滚动恢复 - UNDO）。**

### 当今十大主流的关系型数据库

[Oracle](https://www.oracle.com/database/index.html)**，**[Microsoft SQL Server](https://www.microsoft.com/en-us/sql-server/" \t "_blank)**，**[MySQL](https://www.mysql.com/" \t "_blank)**，**[PostgreSQL](https://www.postgresql.org/" \t "_blank)**，**[DB2](https://www.ibm.com/analytics/us/en/db2/" \t "_blank)**，**[Microsoft Access](https://products.office.com/zh-cn/access)**，** [SQLite](https://www.sqlite.org/)**，**[Teradata](https://www.teradata.com.cn/" \t "_blank)**，**[MariaDB](https://mariadb.org/" \t "_blank)**(MySQL的一个分支)，**[SAP](https://www.sap.com/" \t "_blank)

### Mysql

#### MySql 索引的命中规则

**最左匹配原则:**

最左前缀匹配原则, mysql会一只向右匹配直到遇到范围查询(>, <, between, like)就停止匹配, 比如a=1 and b=2 and c>3 and d=4 如果建立了(a,b,c,d)顺序的索引, d是用不到索引的, 如果建立(a,b,d,c)的索引, 则都可以使用到, a,b,d的顺序可以任意调整.

= 和 in 可以乱序, 比如 a=1 and b=2 and c=3 建立(a,b,c)索引可以任意顺序, mysql 的查询优化器会帮你优化成索引可以识别的形式.

**1.联合索引**

使用情况:

对于查询语句"select e.\* from e where e.e1=1 and e.e3=2"涉及到两列, 这个时候我们一般采用一个联合索引(e1, e3); 而不用两个单列索引, 这是因为一条查询语句往往应为mysql优化器的关系只用了一个索引, 就算你有两个索引, 他也只会用到一个(Mysql的最左匹配原则) ; 在只用到一个的基础上, 联合索引是会比单列索引要快的;

命中规则:

示例: create table e(e1 int, e2 varchat(9), e3 int, primary key(e1, e3));

这样就建立了一个联合索引: e1, e3

使用联合索引的全部索引键, 可触发索引的使用.

如: select e.\* from e where e.e1=1 and e.e3=2

查询条件中包含索引的前缀部分, 也就是 e1, 可以触发索引的使用

如: select e.\* from e where e.e1=1

使用部分索引键, 但不包含索引的前缀部分，不可触发索引的使用。

如: select e.\* from e where e.e3=1

使用联合索引的全部索引键, 但不是AND操作, 不可以触发索引的使用

如: select e.\* from e where e.e3=2 or e.e1=1

**2.普通索引**

就是最基本的索引, 查他就能命中

**3. 唯一索引**

和普通索引类似, 不同的就是索引的列必须是唯一存在的, 可以为空

**4. 全文索引**

只支持老版本的MySql 也就是引擎为MyISAM的数据表

#### 测试时关闭缓存

原因：

1.数据库缓存影响性能；

2.从性能测试的角度看，可以避免性能失真。（我感觉）

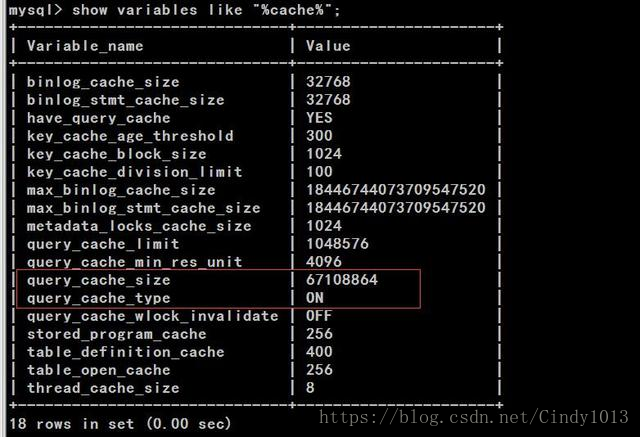
方法：

有两种方法，一个是临时性的 一个是永久性的。

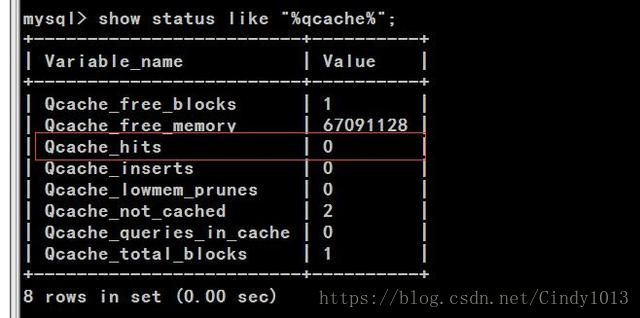
1.临时性

（1）进入数据库

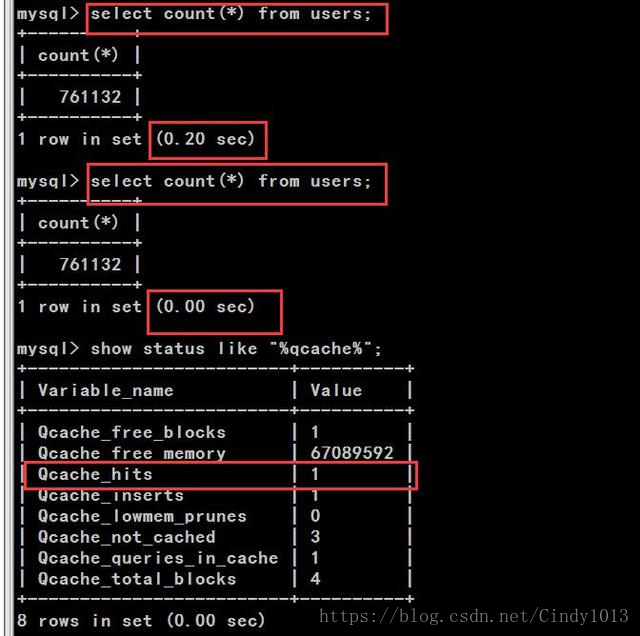
（2）查看缓存是否开启，show variables like '%cache%';



（3）查看缓存命中情况



（4）然后对表查询两次，然后看缓存命中情况



**（5）临时的直接执行**

**set global query\_cache\_size=0**

**set global query\_cache\_type=0**

**（6）永久的修改配置文件my.cnf ,添加下面的配置即可。  
query\_cache\_type=0**

**query\_cache\_size=0**

**开启缓存的方法：**

**再开启缓存的情况下我们对sql语句做一些改动**

**Select sql\_no\_cache count(\*) from users; 不缓存**

**Select sql\_cache count(\*) from users; 缓存（也可以不加，默认缓存已经开启了）**

## 非关系型数据库

NoSQL，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题。

### NoSQL数据库的四大分类

键值(Key-Value)存储数据库

这一类数据库主要会使用到一个哈希表，这个表中有一个特定的键和一个指针指向特定的数据。Key/value模型对于IT系统来说的优势在于简单、易部署。但是如果DBA只对部分值进行查询或更新的时候，Key/value就显得效率低下了。举例如：Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB.

列存储数据库。

这部分数据库通常是用来应对分布式存储的海量数据。键仍然存在，但是它们的特点是指向了多个列。这些列是由列家族来安排的。如：Cassandra, HBase, Riak.

文档型数据库

文档型数据库的灵感是来自于Lotus Notes办公软件的，而且它同第一种键值存储相类似。该类型的数据模型是版本化的文档，半结构化的文档以特定的格式存储，比如JSON。文档型数据库可 以看作是键值数据库的升级版，允许之间嵌套键值。而且文档型数据库比键值数据库的查询效率更高。如：CouchDB, MongoDb. 国内也有文档型数据库SequoiaDB，已经开源。

图形(Graph)数据库

图形结构的数据库同其他行列以及刚性结构的SQL数据库不同，它是使用灵活的图形模型，并且能够扩展到多个服务器上。NoSQL数据库没有标准的查询语言(SQL)，因此进行数据库查询需要制定数据模型。许多NoSQL数据库都有REST式的数据接口或者查询API。如：Neo4J, InfoGrid, Infinite Graph.

因此，我们总结NoSQL数据库在以下的这几种情况下比较适用：1、数据模型比较简单；2、需要灵活性更强的IT系统；3、对数据库性能要求较高；4、不需要高度的数据一致性；5、对于给定key，比较容易映射复杂值的环境。

### 适用场景

NoSQL数据库在以下的这几种情况下比较适用：

1、数据模型比较简单；

2、需要灵活性更强的IT系统；

3、对数据库性能要求较高；

4、不需要高度的数据一致性；

5、对于给定key，比较容易映射复杂值的环境。

## 总结

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库 类型 | 特性 | 优点 | 缺点 |
| 关系型数据库 SQLite、Oracle、mysql | 1、关系型数据库，是指采用了关系模型来组织 数据的数据库； 2、关系型数据库的最大特点就是事务的一致性； 3、简单来说，关系模型指的就是二维表格模型， 而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。 | 1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型相对网状、层次等其他模型来说更容易理解； 2、使用方便：通用的SQL语言使得操作关系型数据库非常方便； 3、易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率； 4、支持SQL，可用于复杂的查询。 | 1、为了维护一致性所付出的巨大代价就是其读写性能比较差； 2、固定的表结构； 3、高并发读写需求； 4、海量数据的高效率读写； |
| 非关系型数据库 MongoDb、redis、HBase | 1、使用键值对存储数据； 2、分布式； 3、一般不支持ACID特性； 4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。 | 1、无需经过sql层的解析，读写性能很高； 2、基于键值对，数据没有耦合性，容易扩展； 3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，而关系型数据库则只支持基础类型。 | 1、不提供sql支持，学习和使用成本较高； 2、无事务处理，附加功能bi和报表等支持也不好； |

注1：数据库事务必须具备ACID特性，ACID是Atomic原子性，Consistency一致性，Isolation隔离性，Durability持久性。

注2：数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库。

## 相关资料

<https://blog.csdn.net/clover_lily/article/details/79991300>