

面向对象程序设计基础

(OOP)

黄民烈

aihuang@tsinghua.edu.cn

<http://coai.cs.tsinghua.edu.cn/html/>

课程团队：刘知远 姚海龙 黄民烈

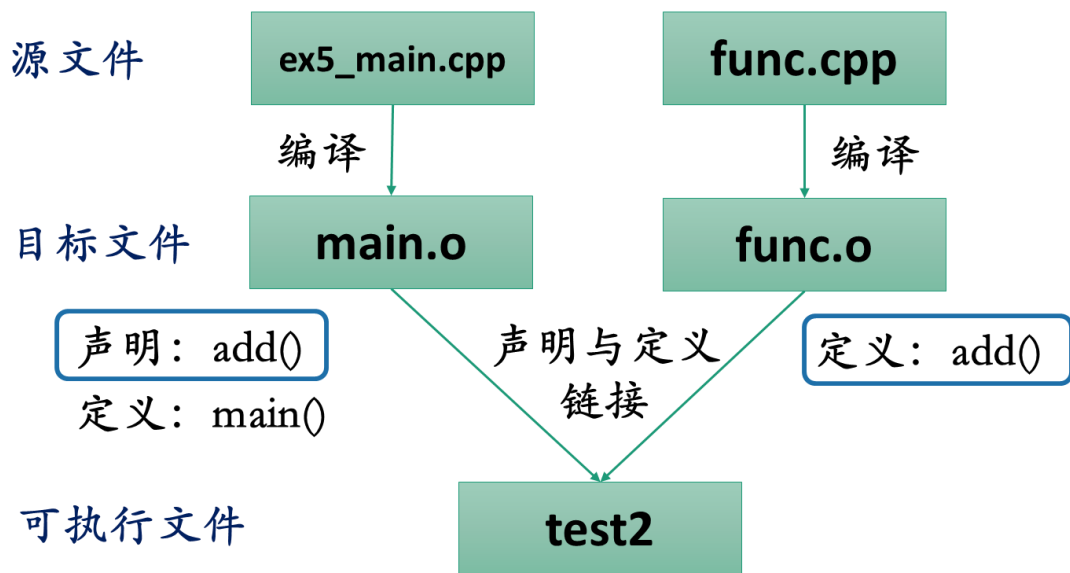
本讲内容题要

- 课程复习
- 实用技巧

基础知识

■ 编译、链接

- 编译：生成目标模块（.o/.obj）
- 链接：链接目标模块，形成可执行程序；外部函数的定义（实现）被寻找和添加



基础知识

■ 声明与定义（函数）

- 函数声明
 - `int ADD(int a, int b);`
- 函数定义（实现）
 - `int ADD(int a, int b) {return a + b;}`
- 同一个函数可以有多次声明，但只能有一次实现；否则链接错误

■ 声明与定义（变量）

- 定义=声明+内存分配
- 变量定义
 - `int x = 0; // 定义并初始化`
 - 全局变量不初始化会设置默认值0
- 变量声明：`extern`
 - `extern int x;`

基础知识

■ 宏定义

- 最简单形式: `#define` <宏名> <字符串>
 - `#define` PI 3.1415926535
- 带参数宏定义: `#define` <宏名>(<参数表>) <字符串>
 - `#define` sqr(x) ((x) * (x))
 - 注意添加括号以保证替换正确性
- 例:
 - 宏定义 `#define` M(y) y*y+3*y, 则M(1+1)的值是?

$$M(1+1) = 1+1*1+1+3*1+1 = 7$$

基础知识

■ 宏定义的使用

- `#include` 直接复制头文件内容
- 宏定义可以用于防止头文件被重复包含
- `#ifndef`

```
#ifndef __BODYDEF_H__  
#define __BODYDEF_H__  
    // 头文件内容  
#endif
```

- `#pragma once`

基础知识

■ Make与Makefile

- 自动检测修改，编译修改模块，链接目标程序
- 运行Makefile
 - make
 - make <任务名>

```
# 注释用#开头
all: main test
main: main.cpp student.cpp
    g++ -o main main.cpp student.cpp
test: student.cpp student_test.cpp
    g++ -o test student_test.cpp student.cpp
clean:
    rm main test
```

冒号为“任务”
的“条件”

冒号前为
“任务”名

指令前必须为
Tab

完成“任务”的
指令（过程）

基础知识

■ 函数重载

- 同一名称的函数，有两个以上不同的实现
- 条件
 - 至少有一个参数的类型不同；或参数数目不同
 - 返回值，参数名称等不能作为区分标识
- 优先调用类型匹配的函数实现，否则才进行类型转换

基础知识

■ 函数参数的缺省值

- 缺省值必须是最后一个参数
- 参数顺序：先放无缺省值的，再放有缺省值的
 - `void print(char* name, int score=0, char* msg="pass")`
- 缺省值导致函数调用二义性

```
void fun(int a, int b=1) {  
    cout << a + b << endl;  
}
```

```
void fun(int a) {  
    cout << a << endl;  
}
```

//测试代码

`fun(2);`//编译器不知道该调用第一个还是第二个函数

基础知识

■ 其他

- 空指针: `NULL` vs. `nullptr`
- `auto` 关键字
 - 代替冗长的类型声明, 需根据上下文进行推导
- `decltype` 关键字
 - `auto+decltype` 追踪返回类型
- 内联函数
 - 提高程序运行效率

类和对象基础

- 类 (class) = “属性/数据” + “服务/函数”
- 在头文件中声明类，在实现文件中定义成员函数

```
// matrix.h  
  
#ifndef MATRIX_H  
#define MATRIX_H  
  
class Matrix {  
    int data[6][6];  
public:  
    void fill(char dir);  
};  
  
#endif
```

```
// matrix.cpp  
  
#include "matrix.h"  
  
// 类外实现需要用类名限定  
void Matrix::fill(char dir) {  
    ... // 函数实现  
}
```

类和对象基础

■ 访问权限

- public、private(默认权限)、protected
- 不允许在类外(非该类的成员函数、非友元)操作访问对象的私有成员和保护成员

类和对象基础 – 创建与销毁

■ 构造函数

- 没有返回值类型，函数名与类名相同
- 可以重载，即可以使用不同的函数参数进行对象初始化

```
class Student {  
    int ID;  
public:  
    Student(int id) { ID = id; }  
    Student(int year, int order) {  
        ID = year * 10000 + order;  
    }  
    Student(int id) : ID2(id), ID1(ID2) { }  
    ...  
};
```

类和对象基础 – 创建与销毁

■ 构造函数

- 就地初始化，不通过构造函数（C++11）
- 默认构造函数
 - 不带任何参数的构造函数，或每个形参提供默认实参的构造函数
 - 何时调用？
 - `ClassName a;` //调用默认构造函数
 - `ClassName b = ClassName();` //同样调用默认构造函数
 - 自动调用成员变量的默认构造函数
 - 若无构造函数，编译器会隐式合成默认构造函数
 - 若定义了其他构造函数，则不会隐式合成默认构造函数

类和对象基础 – 创建与销毁

■ 构造函数

- 默认构造函数
 - 显式声明默认构造函数
`A() = default;`
 - 显式删除构造函数
`A(char ch) = delete;`
- 对象数组的初始化
 - `A a[50];` // 调用默认构造函数
 - `A a[3] = {1, 3, 5}` // 构造函数只有一个参数
 - `A a[3] = {A(1, 2), A(3, 5), A(0, 7)};` // 构造函数有多个参数

类和对象基础 – 创建与销毁

■ 析构函数

- 对象的“死”
 - 当执行到“包含对象定义范围结束处”时，编译器自动调用对象的析构函数。
 - 动态分配的内存是一种典型的需要释放的资源。
- 一个类只有一个析构函数，名称是“~类名”，没有函数返回值，没有函数参数

```
class Classroom {  
    int num;  
    int* ID_list;  
public:  
    Classroom() : num(0), ID_list(nullptr) {}  
    ...  
    ~Classroom() { // 析构函数  
        if (ID_list) delete[] ID_list; // 释放内存  
    }  
};
```


类和对象基础 – 创建与销毁

■ 析构函数

- 自身销毁时，自动调用成员变量的析构函数
 - 先执行自己的析构函数，再执行成员变量的析构函数
- 当用户没有自定义析构函数时，编译器会自动合成一个隐式的析构函数 `~A() {}`
- 局部对象
 - 在局部对象生命周期结束、即所在作用域结束后被析构
- 全局对象
 - 在main()函数调用之前进行初始化
 - 在同一编译单元（文件）中，按照定义顺序进行初始化
 - 不同编译单元中，对象初始化顺序不确定
 - 在main()函数执行完return之后，对象被析构

类和对象基础 – 创建与销毁

■ 运算符重载

- 为了让自定义类型“像”基本类型，模仿基本类型的基本操作
- 运算重载一般有两种方式
 - 全局函数的运算符重载
 - `A operator+(A a, A b) {...}`
 - 访问private成员怎么办？声明为友元
 - 成员函数的运算符重载

```
class A{  
    int data;  
public:  
    A operator+(A b) {...};  
};
```

类和对象基础 – 创建与销毁

■ 运算符重载

- 前缀/后缀运算符

- 前缀

- ```
ClassName operator++();
```

- ```
ClassName operator--();
```

- 后缀

- ```
ClassName operator++(int dummy);
```

- ```
ClassName operator--(int dummy);
```

- 哑元可以没有变量名

- 函数运算符()重载

- ```
ReturnType operator()(Parameters);
```

- ```
Obj(real_parameters); //调用
```

- 使得对象看起来像是一个函数（“函数对象”）

类和对象基础 – 创建与销毁

■ 运算符重载

- 数组下标运算符[]重载
 - 如果返回类型是引用，则数组运算符调用可以出现在等号左边，接受赋值
`Obj[index] = value;`
 - 如果返回类型不是引用，则只能出现在等号右边
`Var = Obj[index];`
- 流运算符>>、<<重载（一般情况下只能全局重载）
 - `istream& operator>> (istream& in, Test& dst);`
 - `ostream& operator<< (ostream& out, const Test& src);`

类和对象基础 – 创建与销毁

■ 友元

- 被声明为友元的函数或类，具有对出现友元声明的类的private及protected成员的访问权限，即可以访问该类的一切成员
- 友元的声明
 - 只能在类内进行
 - 友元函数
 - 友元类

```
class A {  
    friend void foo(A &a);  
    friend void B::foo(A &a);  
    friend B::B(A &a), X::~~X();  
    friend C; //友元类  
};
```

类和对象基础 – 创建与销毁

■ 静态变量/函数（使用static修饰）

- 静态变量
 - 静态局部变量存储在静态存储区，生命周期将持续到整个程序结束
 - 静态全局变量作用域仅限其声明的文件，不能被其他文件所用，可以避免和其他文件中的同名变量冲突
 - 离开作用域不析构，程序运行最后析构
- 静态函数
 - 定义示例：`static int func() {...}`
 - 静态函数作用域仅限其声明的文件，不能被其他文件所用，可以避免和其他文件中的同名函数冲突

类和对象基础 – 创建与销毁

■ 静态成员变量

- 属于整个类的“类变量”，被该类的所有对象共享
- 对象/类名访问
- 类似于全局变量，在程序开始前初始化

■ 静态成员函数

- 属于整个类，被该类的所有对象共享
- 使用对象/类名访问
- 由于不属于某个对象而属于整个类，因此不能访问非静态成员

类和对象基础 – 创建与销毁

■ 常量

- 使用 `const` 修饰
- 修饰变量：必须就地初始化，在生命周期内值不改变

■ 常量数据成员

- 在对象生命周期内不能更改
- 初始化：初始化列表/就地初始化，不能赋值

■ 常量成员函数

- 实现不能修改类的数据成员
- `ReturnType Func(...) const {...}`
- 常量对象(`const ClassName a;`)只能调用常量成员函数

类和对象基础 – 创建与销毁

■ 常量静态变量

- 定义/初始化
 - 类外定义; int和enum可就地初始化
- 不存在常量静态函数

```
class foo {  
    static const char* cs; // 不可就地初始化  
    static const int i = 3; // 可以就地初始化  
    static const int j; // 也可以在类外初始化  
};
```

```
const char* foo::cs = "foo C string";  
const int foo::j = 4;
```

类和对象基础 – 引用与复制

■ 左值引用（“别名”）

- 左值：可以取地址、有名字的值。
- 格式：类型名 & 引用名 = 变量名
- 必须在定义时进行初始化，且不能修改引用指向
- 和指针的区别
 - 不存在空引用
 - 不能被指向到另一个对象
 - 引用必须在创建时被初始化为一个对象
- 函数返回值可以是左值引用，但不得指向函数的临时变量

类和对象基础 – 引用与复制

■ 右值引用

- 右值：不能取地址、没有名字的值；常见于常值、函数返回值、表达式

`int &&e = a + b;` 😊

- 右值引用可以延续即将销毁变量的生命周期，可以减少拷贝带来的开销

类和对象基础 – 引用与复制

■ 拷贝构造函数

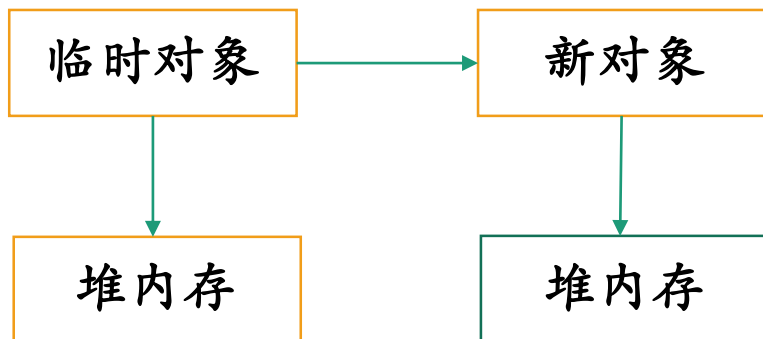
- 使用左值引用作为参数的构造函数叫做拷贝构造函数
 - `ClassName(ClassName& VariableName);`
- 在发生拷贝操作时调用
 - `Test t0; Test t1 = t0;`
 - `Test t0; Test t1(t0);`
- 区分 拷贝构造函数与拷贝赋值运算符
 - `Test t0; Test t1 = t0;` 调用（默认）拷贝构造函数
 - `Test t0; t0 = t0;` 调用（默认）拷贝赋值运算符

类和对象基础 – 引用与复制

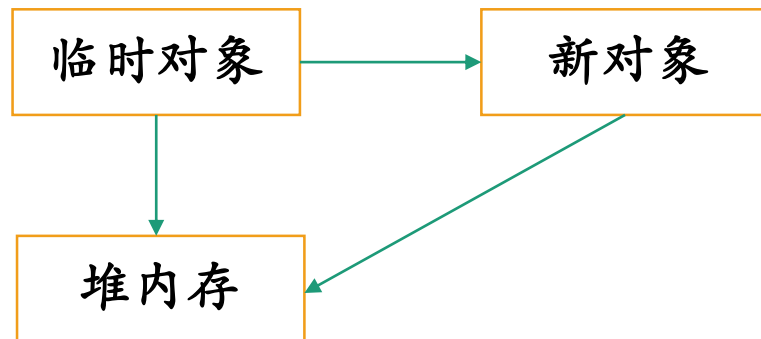
■ 移动构造函数

- 使用右值引用作为参数的构造函数叫做移动构造函数
 - `ClassName(ClassName&& VariableName);`
- 对于一些即将析构的临时类，移动构造函数直接利用了原来临时对象中的堆内存，新的对象无需开辟内存，临时对象无需释放内存，从而大大提高计算效率。

拷贝构造函数



移动构造函数



类和对象基础 – 引用与复制

■ `std::move`

- 移动构造函数加快了右值初始化的构造速度
- 使用`move`函数则可以使用移动构造函数，加快左值初始化的构造速度
- 本身只进行类型转换（左值转换成右值），对于对象本身的操作在移动构造函数中进行

```
std::string str = "Hello";
```

```
std::move(str);
```

```
cout<<str; // 输出 Hello
```

```
std::string str = "Hello";
```

```
std::string str2 = std::move(str);
```

```
cout<<str; // 输出空字符串
```

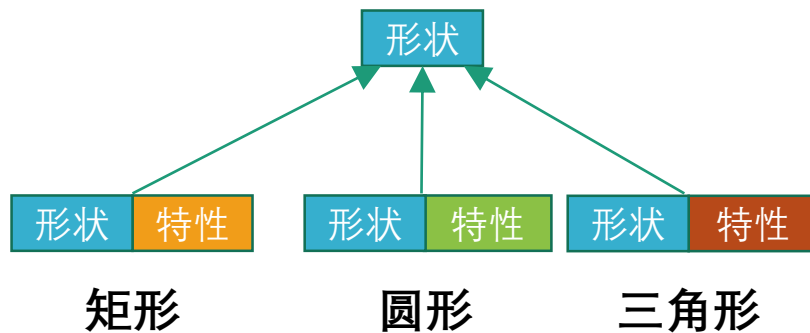
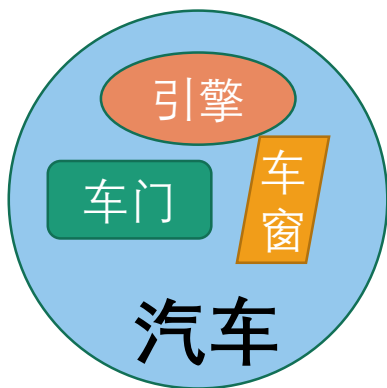
进一步阅读:

<https://stackoverflow.com/questions/13127455/what-does-the-standard-library-guarantee-about-self-move-assignment>

类和对象基础 – 组合与继承

■ 对象之间的关系

- has-a（整体-部分）：组合
- is-a（一般-特殊）：继承

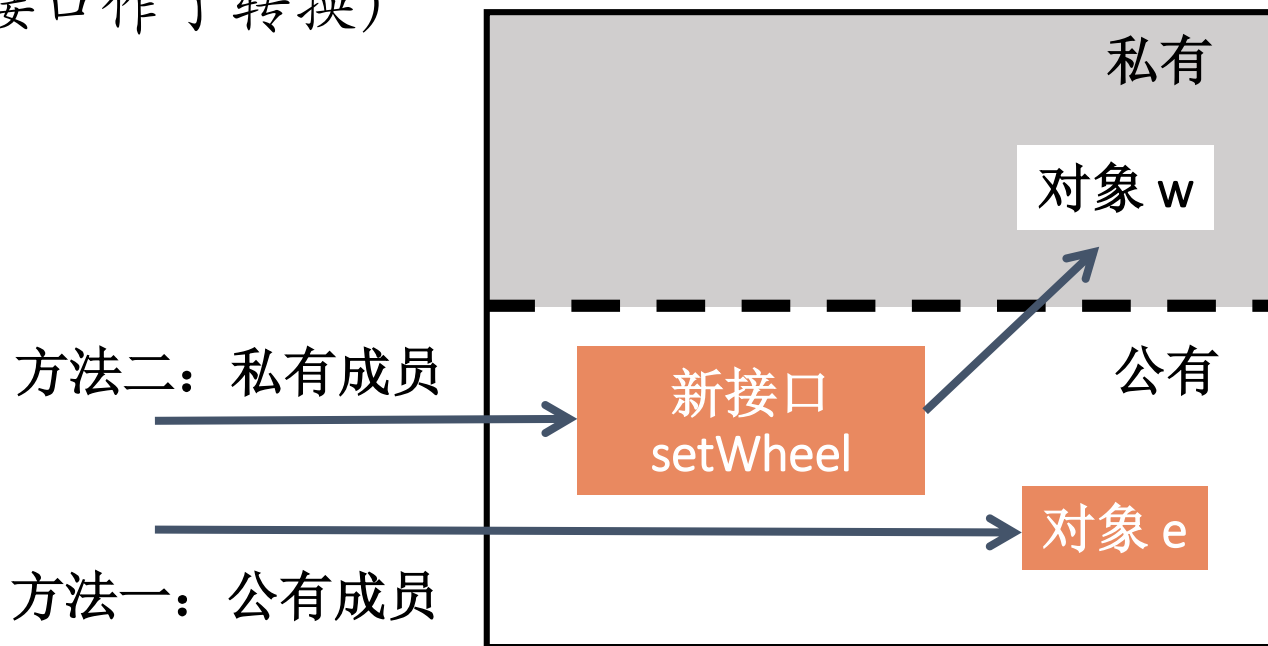


- 恰当的利用组合及继承，可以实现代码重用，实现增量开发

类和对象基础 – 组合与继承

■ 组合的两种实现方法

- 已有类的对象作为新类的**公有**数据成员，这样通过允许**直接访问子对象**而“提供”旧类接口
- 已有类的对象作为新类的**私有**数据成员。新类可以调整旧类的对外接口，可以不使用旧类原有的接口（相当于对接口作了转换）



类和对象基础 – 组合与继承

■ 继承

- 数据、接口都会被继承给子类
- 基类的构造函数、析构函数、友元函数不会被继承
- 成员访问权限

继承表		继承方法					
		public		private		protected	
基类中 成员类型	public	OK	pub/yes	OK	prv/no	OK	pro/no
	private	NO	prv/no	NO	prv/no	NO	prv/no
	protected	OK	pro/no	OK	prv/no	OK	pro/no

默认

类和对象基础 – 组合与继承

■ 继承中接口的重写隐藏

- 重写：函数名相同，参数不同，作用域相同
- 重写隐藏：
 - 在派生类中重新定义基类函数，实现派生类的特殊功能
 - 函数名相同，参数可不同，派生类与基类
- 重写隐藏可用using关键字恢复

```
class Base {  
public:  
    void f() {}  
    void f(int i) {} /// 重载  
};  
class Derive : public Base {  
public:  
    void f(int i) {} ///重写隐藏  
};
```

```
int main() {  
    Derive d;  
    d.f(10);  
    // d.f();    // 被屏蔽，编译错误  
    return 0;  
}
```

类和对象基础 – 虚函数

■ 向上类型转换

- 凡是接受基类对象/引用/指针的地方（如函数参数），都可以使用派生类对象/引用/指针，编译器会自动将派生类对象转换为基类对象以便使用。（只对public继承有效）

■ 对象切片

- 当派生类的对象被转换为基类的对象时，派生类的对象被切片为对应基类的子对象
- 对象切片将使得派生类的新数据、新接口丢失

类和对象基础 – 虚函数

■ 函数调用捆绑

- 早捆绑：运行之前已经决定了函数调用代码到底进入哪个函数
- 晚捆绑：要求在运行时能确定对象的实际类型，并绑定正确的函数

■ 虚函数实现晚捆绑（动态多态）

- 对于被派生类重新定义的成员函数，若它在基类中被声明为虚函数，则通过基类指针或引用调用该成员函数时，编译器将根据对象的实际类型决定是调用基类中的函数，还是调用派生类重写的函数

类和对象基础 – 虚函数

■ 虚函数

- 只对指针和引用有效
- 晚绑定通过虚函数表实现
 - 虚函数表VTABLE：包含虚函数的类存储虚函数地址的表
 - 编译时建立，记录该类和该类的基类中所有已声明的虚函数入口地址
 - 虚函数指针VPTR：包含虚函数的对象中，指向该类VTABLE的指针
 - 运行时设立，在构造函数中发生
 - 实质上，通过指向特定VTABLE，起到记录对象类型的作用，从而可以进行晚捆绑

类和对象基础 – 虚函数

■ 虚函数

- 虚函数与构造函数
 - 构造函数不能也不必是虚函数
 - 不能：在构造函数调用前，VPTR未初始化，无法使用虚函数
 - 不必：构造函数的作用是提供类中成员初始化，调用时明确指定要创建对象的类型，没有必要是虚函数
- 虚函数与析构函数
 - 应当总是将基类的析构函数设置为虚析构函数
 - 析构需要根据对象的实际类型进行析构
- 在构造函数和析构函数中调用一个虚函数，被调用的只是这个函数的本地版本，即虚机制在构造函数和析构函数中不工作

类和对象基础 – 虚函数

■ 虚函数的重写覆盖

- 派生类重新定义基类中的虚函数，函数名必须相同，函数参数必须相同，返回值一般情况应相同。
- 重载、重写隐藏、重写覆盖

	重载(overload)	重写隐藏(redefining)	重写覆盖(override)
作用域	相同(同一个类中，或者均为全局函数)	不同(派生类和基类)	不同(派生类和基类)
函数名	相同	相同	相同
函数参数	不同	相同/不同	相同
其他要求	—	如果函数参数相同，则基类函数不能为虚函数	基类函数为虚函数

类和对象基础 – 抽象类

■ 纯虚函数

- **virtual** 返回类型 函数名(形式参数) = **0**; (此处0填充在虚表中)
- 由于编译器不允许被调用函数的地址为0, 所以该类不能生成对象。
- 在它的派生类中, 除非重写此函数, 否则也不能生成对象。

类和对象基础 – 类型转换

■ 两种类型转化

- `static_cast`

- 可以进行内置数据类型的转换，也可以进行类的指针或者类的引用的强制转换

- `dynamic_cast`

- 主要用于将基类类型转化为子类类型
 - 根据虚函数表的信息判断实际类型
 - 不能用于内置基本数据类型的强制转换
 - 不允许两个没有关联的类的指针相互转换

类和对象基础 – 模板

■ 模板

- 继承与组合提供了重用对象代码的方法，而C++的模板特征提供了重用源代码的方法。

- 函数模板

```
template <typename T> ReturnType Func(Args);
```

- 函数模板在调用时，编译器能自动推导出实际参数的类型（这个过程叫做实例化）
- 对模板的处理是在编译期进行的，每当编译器发现对模板的一种参数的使用，就生成对应参数的一份代码

类和对象基础 – 模板

■ 模板

- 类模板

```
template <typename T> class A { ... }
```

- 类模板中成员函数的类外定义

```
template<typename T>  
void A<T>::func() { ... }
```

- 模板使用泛型标记，使用同一段代码，来关联不同但相似的特定行为，最后可以获得不同的结果。模板也是多态的一种体现

Git、BASH脚本、 Markdown语法简介

目录

■ Git、Github

- 什么是Git
- 为什么用Git
- Git简单操作介绍
- 使用Github管理仓库

■ BASH脚本

■ Markdown

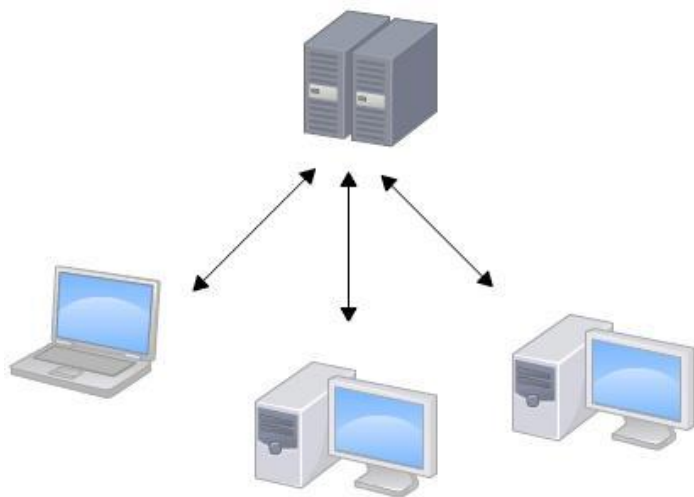
- 语法简单介绍
- 操作演示

仅做介绍
期末不考

什么是Git?

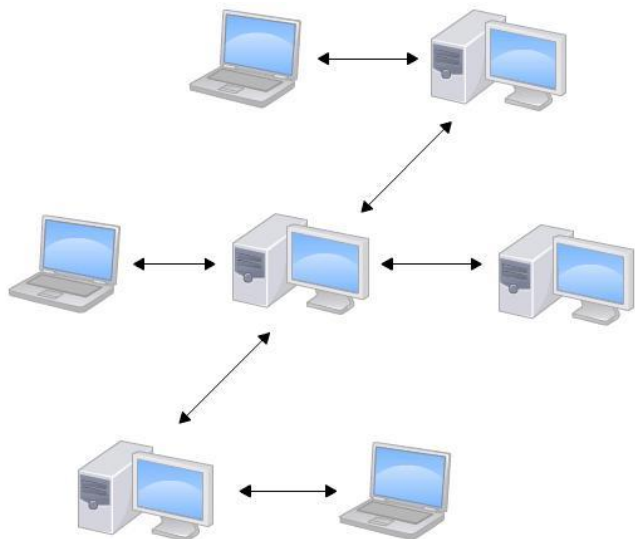
- Linux之父Linus Torvalds为开发Linux内核而建立的一个分布式版本控制软件

分布式



版本控制软件

我改了哪里
刚刚还好好的
怎么突然有BUG了
? ? ? ?



版本控制软件

我改了哪里
刚刚还好好的
怎么突然有BUG了
? ? ? ?



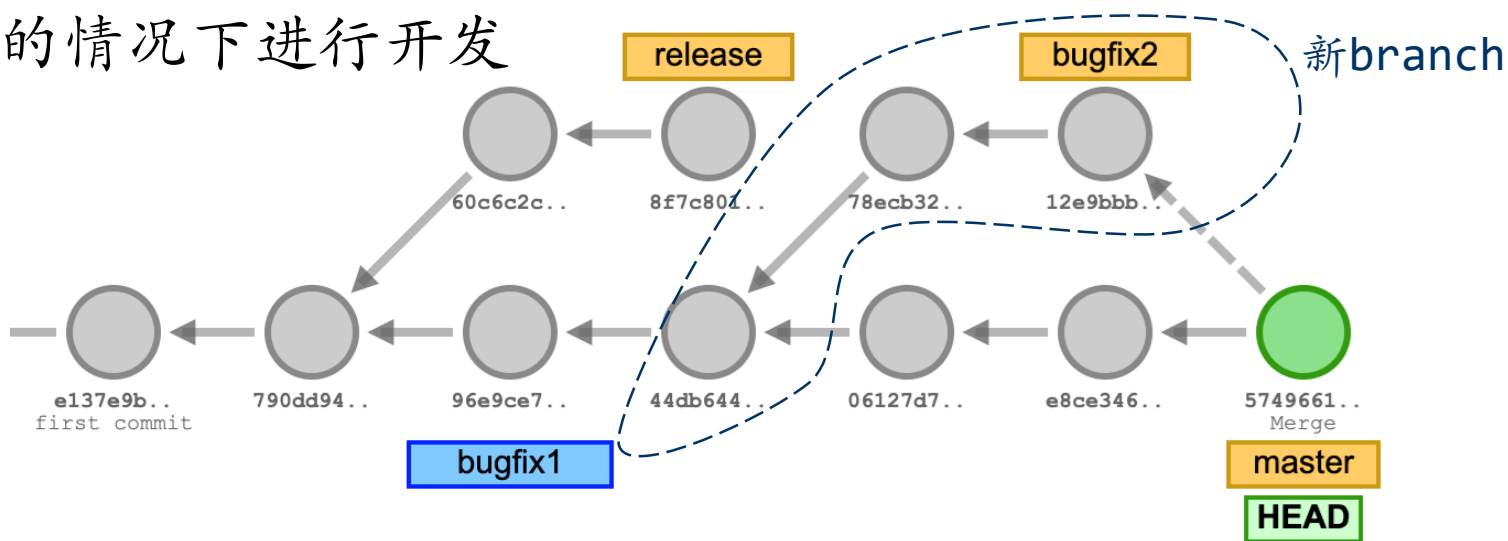
版本控制软件帮助使用者找出

- 不同版本之间的差异
- 什么时候做出了这个修改
- 谁做出了这个修改
- 做出修改的人给出的修改理由

为什么用git?

■ 除了版本控制软件本身的优势以外:

- 通过查看 `git history`, 开发者可以看到一个项目开发的时间线
- 通过 `git branch`, 开发者可以在不用担心影响主代码的情况下进行开发



从git repository开始

■git repo

- 包含了一个项目的所有文件、文件夹，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”

History for [pytorch](#) / README.md

Commits on Apr 23, 2021

README: Minor improvements ([#56193](#)) ...



garymm authored and facebook-github-bot committed 7 days ago ✖

Commits on Apr 1, 2021

update build tutorial - choose the correct VS version ([#54933](#)) ...

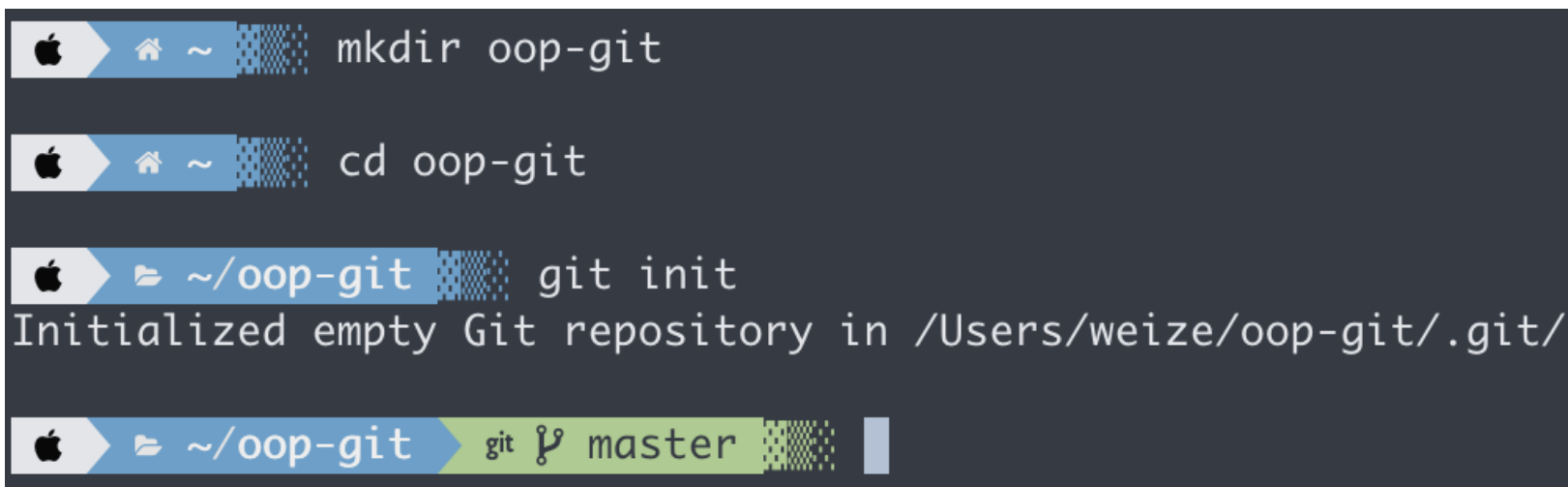


mszhanyi authored and facebook-github-bot committed 28 days ago ✖

从git repository开始

■创建仓库

- 创建一个新文件夹作为你的第一个repo, 在命令行中进入该文件夹, 输入`git init`, 以使用git来管理这个文件夹



```
Apple > ~ █ mkdir oop-git

Apple > ~ █ cd oop-git

Apple > ~/oop-git █ git init
Initialized empty Git repository in /Users/weize/oop-git/.git/

Apple > ~/oop-git git master █
```

从git repository开始

■添加文件

- 在这个文件夹中添加文件（例如我们写了一个空的 test.py），使用 `git add test.py` 将文件到目前为止的修改放入git的暂存区。

```
🍏 ~/oop-git git master ?1 ls
test.py

🍏 ~/oop-git git master ?1 git add test.py

🍏 ~/oop-git git master +1
```

从git repository开始

■记录修改

- 当所有的修改都用git add加入到暂存区后，就使用 `git commit -m “备注内容”` 将所有的暂存区里的修改提交至本地仓库

```
🍏 ~ /oop-git git master +1 git commit -m "modify test.py"
[master (root-commit) c9df5e6] modify test.py
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 test.py
```

从git repository开始

- 为了展示git的作用，我们在test.py中添加两行代码

```
import torch
import numpy as np
```

- 然后再次执行 `git add test.py; git commit -m “...”` 提交更改

```
🍏 ~/oop-git git master +1 git commit -m "add imports in test.py"
[master 1aada33] add imports in test.py
1 file changed, 2 insertions(+)
```

从git repository开始

■查看历史

- 通过`git log`, 我们可以查看之前的commit记录, 以及对应的sha编码

```
commit 1aada3314db2df6fabb8092ea6c6a5dd47764fdf (HEAD -> master)
Author: chenweize1998 <chenweize1998@gmail.com>
Date:   Fri Apr 30 14:36:57 2021 +0800

    add imports in test.py

commit c9df5e6335c7a5b8d1053e300bb5c1e017ba73dd
Author: chenweize1998 <chenweize1998@gmail.com>
Date:   Fri Apr 30 14:34:26 2021 +0800

    modify test.py
```

从git repository开始

■查看修改

- 如果想要查看某次commit相对上次的改动，可以记录下此次commit的sha编码，通过`git show 编码`来查看

```
commit 1aada3314db2df6fabb8092ea6c6a5dd47764fdf (HEAD -> master)
Author: chenweize1998 <chenweize1998@gmail.com>
Date:   Fri Apr 30 14:36:57 2021 +0800

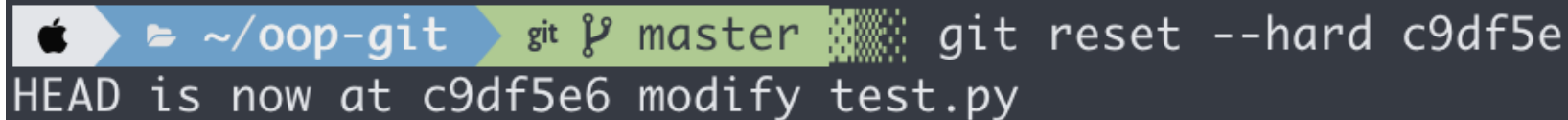
    add imports in test.py

diff --git a/test.py b/test.py
index e69de29..749fb11 100644
--- a/test.py
+++ b/test.py
@@ -0,0 +1,2 @@
+import torch
+import numpy as np
(END)
```


从git repository开始

- 如果想要回退到某一次commit, 可以使用

```
$ git reset --hard sha编码
```

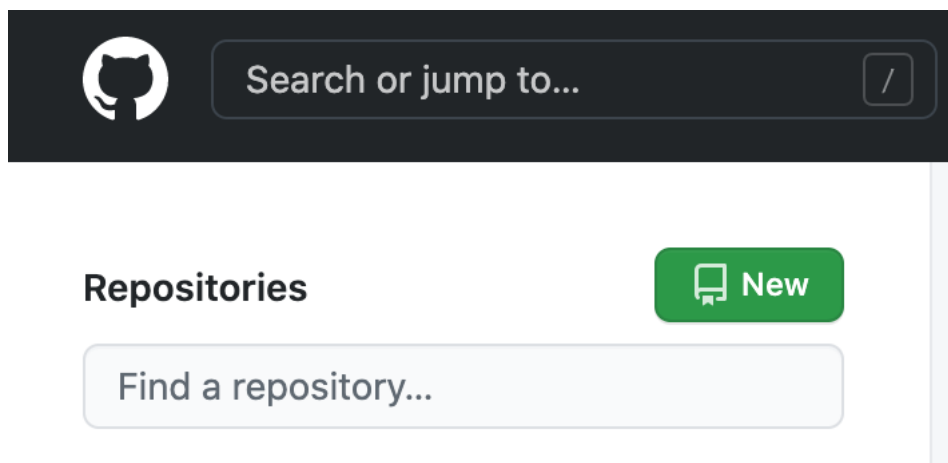


A terminal window screenshot showing a git reset command. The window title bar includes an Apple logo, a folder icon, and the path ~/oop-git. The terminal prompt shows the user is on the master branch. The command executed is git reset --hard c9df5e. The output indicates that the HEAD is now at c9df5e6 and that the file test.py has been modified.

```
git master git reset --hard c9df5e  
HEAD is now at c9df5e6 modify test.py
```

远程repository

- 登陆github后，在左侧可以创建新repo



远程repository

- 登陆github后，在左侧可以创建新repo

Owner *



chenweize1998 ▾

/

Repository name *

oop-git



Great repository names are short and memorable. Need inspiration? How about **fantastic-octo-engine**?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

远程repository

- 打开电脑命令行，创建文件夹，进入后按照github的提示进行输入

```
git remote add origin git@github.com:chenweize1998/oop-git.git
git branch -M main
git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 212 bytes | 106.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:chenweize1998/oop-git.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

注：你可能需要先进行本地的git配置

git config --global user.email "you@example.com"

git config --global user.name "Your Name"

远程repository

- 现在，你有了你的第一个github repository!
开始和同学合作工作吧！

The screenshot shows a GitHub repository page for 'chenweize1998 / oop-git'. The repository is private and has 1 unwatch, 0 stars, and 0 forks. The 'Code' tab is selected, showing a file named 'test.py' modified 43 minutes ago. A button 'Add a README' is visible. The right sidebar shows sections for 'About' (No description, website, or topics provided), 'Releases' (No releases published), and 'Packages' (No packages published).

chenweize1998 / oop-git Private

Unwatch 1 Star 0 Fork 0

Code Issues Pull requests Actions Projects Wiki ...

main Go to file Add file Code 加速

chenweize1998 modify test.py 43 minutes ago 1

test.py modify test.py 43 minutes ago

Add a README with an overview of your project. Add a README

About No description, website, or topics provided.

Releases No releases published Create a new release

Packages No packages published Publish your first package

Git常用命令

- `git init`: 把当前文件夹作为一个全新的git repo
- `git add filename`: 当你修改了filename这个文件, 通过这条命令可以把修改暂存, 以供之后 `commit` 提交
- `git commit`: 把通过 `git add` 放到暂存区里的所有修改提交到本地repo

Git常用命令

- `git push`: 把本地commit的所有修改推送到远程repo (推送到github上)
- `git pull`: 把远程repo同步到本地
- `git status`: 查看本地repo中文件追踪的情况
- `git branch`: 查看repo不同分支情况、开新分支
- `git merge`: 合并两个分支
- `git clone`: 把一个远程repo克隆到本地

创建远程repo的命令解释

```
$ git remote add origin git@github...  
$ git branch -M main  
$ git push -u origin main
```

1. 把github托管的那个repo链接与本地关联，并用origin作为简写
2. 将当前的分支命名为main
3. 把本地的main分支推送到origin代表的远程仓库

更多资源

- <https://try.github.io/> 列举了几个学习git 的链接
- <https://git-scm.com/book/zh/v2> 提供中文版的《Pro git》，详细但复杂
- <https://www.liaoxuefeng.com/wiki/896043488029600> 廖雪峰的中文git教程

BASH脚本

什么是BASH脚本

「任何在命令行中能正常执行的命令都可以被写进一个BASH脚本并完成一样的事，反之亦然」

- 便于一次性执行大量命令
- 一般使用.sh作为文件后缀
- 一般在命令行使用 `$ bash xxx.sh` 启动脚本
- 如在sh文件第一行通过特定指令指定解释器，如 `#!/bin/bash`，则可以在使用 `$./xxx.sh` 启动脚本

BASH脚本示例

■ 批量修改文件名(.txt→.cpp)

```
for name in `ls *.txt`; do
    mv $name ${name%.txt}.cpp
done
```

■ 批量输入文件

```
INPUT_DIR=testcases # 测例所在文件夹
OUTPUT_DIR=output # 输出文件夹
mkdir -p $OUTPUT_DIR # 若不存在输出文件夹，则创建
for input in `ls $INPUT_DIR/case*.txt`; do
    # '<'用于输入重定向，将$input表示的文件里的内容
    # 输入到test程序里。
    # '>'用于输出重定向。${input##*/}表示获取$input的
    # 文件名，如${aaa/bcd.txt##*/} --> bcd.txt
    ./test < $input > $OUTPUT_DIR/${input##*/}
done
```

BASH基本语法

■ 空格或tab区分参数

- `$ command foo bar`表示foo和bar为command的两个参数

■ 使用分号隔开不同命令表示顺序执行这些命令

- `$ clear; ls`表示先执行clear再执行ls，与两条指令分两行效果相同

■ `$ cmd1 && cmd2`:若cmd1成功，才执行cmd2

■ `$ cmd1 || cmd2`:若cmd1失败，才执行cmd2

BASH基本语法

■ 变量声明: `variable=value`

- 等号两侧不能有空格!
- Bash没有数据类型的概念, 所有的变量值都是字符串

■ 读取变量: `$variable`或`${variable}`

■ 特殊变量:

- `$?`: 上一个命令的退出码, 成功为0, 失败为非0
- `$#`: 传递给脚本的参数个数
- `$@`: 传递给脚本的全部参数

BASH基本语法

■ 获取脚本参数

- 对于命令行调用 `$ bash script.sh arg1 arg2`
- `$0` 为脚本文件名, `$1` 为 `arg1`, `$2` 为 `arg2`

```
#!/bin/bash
# script.sh
echo "全部参数: " $@
echo "命令行参数数量: " $#
echo '$0 = ' $0
echo '$1 = ' $1
echo '$2 = ' $2
echo '$3 = ' $3
```

```
bash script.sh a b c
全部参数: a b c
命令行参数数量: 3
$0 = script.sh
$1 = a
$2 = b
$3 = c
```

BASH基本语法

■ 数组声明: `array=(value1 value2 value3...)`

- 例如: `tmp=(1 2 3)`

- 可以不使用连续下标, 如 `array[0]=1; array[5]=0`

■ 读取数组: `${array[n]}`

- 例如: `${tmp[0]}`

- `${tmp[@]}` 可获得tmp数组所有元素

- `${#tmp[@]}` 可获得tmp数组长度

BASH基本语法

■ 条件判断

```
if commands; then
    commands
elif commands; then
    commands
else
    commands
fi
```

```
#!/bin/bash
echo -n "输入一个1到3之间的数字 > "
read character
if [ "$character" = "1" ]; then
    echo 1
elif [ "$character" = "2" ]; then
    echo 2
elif [ "$character" = "3" ]; then
    echo 3
else
    echo 输入不符合要求
fi
```

BASH基本语法

■ 循环：while循环、for循环

```
while condition; do  
    commands  
done
```

```
for variable in list; do  
    commands  
done
```

```
for (( expression1; expression2; expression3 )); do  
    commands  
done
```

BASH基本语法

■ 循环：while循环、for循环

```
#!/bin/bash
for i in word1 word2;
do
    echo $i
done
```

```
for variable in list; do
    commands
done
```

A terminal window with a dark background. The title bar shows an Apple logo, a home icon, a tilde icon, and a window icon. The text in the terminal reads: "bash script.sh", "word1", and "word2".

```
bash script.sh
word1
word2
```

BASH

- 函数、重定向等其他内容，请自行查阅相关资料
- 在实践中学习

Markdown

什么是Markdown?

- Markdown是一种轻量级标记语言，它允许人们使用易读易写的纯文本格式编写文档。
- Markdown能被使用来撰写电子书，如：Gitbook
- GitHub、简书、reddit等网站都支持Markdown的显示

Md编辑器推荐

■Typora

- 即时渲染效果
- 简洁易用

■VsCode的插件Markdown All in One

■用纯文本编辑器也不是不行

Md基本语法

■ 标题

- 使用#标记，可表示1-6级标题，有几个#就是几级标题，例如

- # 一级标题

- ## 二级标题

- ### 三级标题

- 注意#号后的空格

- 显示效果见右

一级标题

二级标题

三级标题

四级标题

五级标题

六级标题

Md基本语法

■ 斜体: `*斜体文本*`

斜体文本

■ 粗体: `**粗体文本**`

粗体文本

■ 删除线: `~~删除线~~`

~~删除线~~

■ 下划线: `<u>下划线</u>`

下划线

Md基本语法

■ 列表

- 无序列表: `+ 第一项`
- 有序列表: `1. 第一项`

- 第一项
- 第二项
- 第三项

1. 第一项
2. 第二项
3. 第三项

Md基本语法

■ 引用： > 引用文本

引用文本

■ 代码块（可以指定语言）：

```
```c++
int main() {
 int a = 0;
 return 0;
}
```
```

```
int main() {
    int a = 0;
    return 0;
}
```

Md基本语法

■ 链接：[链接名称](链接地址)

- 例如：请点击[这里](www.baidu.com) 请点击[这里](#)

■ 图片：![alt 属性文本](图片路径)

- 如要指定图片宽度高度，需要使用html语言：
```

# Md基本语法

## ■ 表格

表头	表头
----	----
单元格	单元格
单元格	单元格

表头	表头
单元格	单元格
单元格	单元格

# Md基本语法

## ■ 公式

行内公式: `$\eta=\frac{dy}{dx}$` 行内公式示例  $\eta = \frac{dy}{dx}$

行间公式:

```
$$
a=
\begin{bmatrix}
1 & 2\\
3 & 4
\end{bmatrix}
$$
```

行间公式示例

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

公式位于两行之间、正中央

# Md基本语法

## ■ 更多

- 支持HTML代码
- 甚至可以在typora中直接画流程图、UML图、甘特图等图形
- 自行探索

**结 束**