

## 自我介绍

---

- 面试官您好！我叫高宜琛，本科就读于大连理工大学软件学院，专业为软件工程，现在是北京大学信息科学与技术系计算机应用专业的一名研二学生，毕业时间是2020年7月。
- 实验室方面，自己的研究方向主要是计算机字库的生成和压缩，完成了使用深度学习和传统图形学方法结合，通过用户输入的少量字符集，来生成完整矢量字库的任务，目前该成果已经投稿至Siggraph Asia 2019 Technical Briefs。
- 跟游戏相关的项目，主要是从5月份开始，在北京Funplus公司实习期间完成的，岗位是游戏客户端研发，项目是一个SLG和COC结合的手游，截止到目前完成了若干个功能和需求，包括地图编辑器功能的增加和优化、Unity编辑器的资源锁、prefab的树形结构展示和游戏中背包的功能等。前面三个功能主要是为了使游戏美术和策划的工作更加便利，而背包功能主要包括物品的显示、使用和宝箱的使用等，涉及界面与数据的绑定、与服务器的通信等模块，相关内容已经整理到github。（其他一些简单的就是数字的压缩显示等）
- 之所以来应聘游戏岗位，最重要的原因是对于游戏和游戏开发的热情和兴趣，从小就觉得游戏带给了自己很多欢乐和感动，很希望能够投入到其中，给别人带来同样的感受。

## 渲染管线

---

- OpenGL中的坐标转换:

四种坐标系:

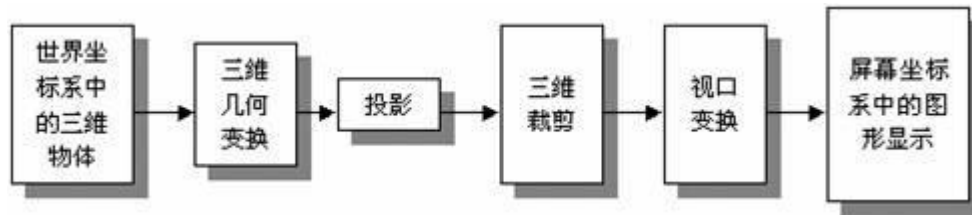
1. 世界坐标
2. 局部坐标
3. 视坐标
4. 屏幕坐标

从三维物体到二维图象，就如同用相机拍照一样，通常都要经历以下几个步骤：

1. 将相机置于三角架上，让它对准三维景物，它相当于OpenGL中调整视点的位置，即视点变换（Viewing Transformation）。
2. 将三维物体放在场景中的适当位置，它相当于OpenGL中的模型变换（Modeling Transformation），即对模型进行旋转、平移和缩放。
3. 选择相机镜头并调焦，使三维物体投影在二维胶片上，它相当于OpenGL中把三维模型投影到二维屏幕上的过程，即OpenGL的投影变换(Projection Transformation)，OpenGL中投影的方法有两种，即正射投影和透视投影。为了使显示的物体能以合适的位置、大小和方向显示出来，必须要通过投影。有时为了突出图形的一部分，只把图形的某一部分显示出来，这时可以定义一个三维视景体（Viewing Volume）。正射投影时一般是一个长方体的视景体，透视投影时一般是一个棱台似的视景体。只有视景体内的物体能被投影在显示平面上，其他部分则不能。注：这里需要进行裁剪。
4. 冲洗底片，决定二维相片的大小，它相当与OpenGL中的视口变换（Viewport Transformation）（在屏幕窗口内可以定义一个矩形，称为视口（Viewport），视景体投影后的图形就在视口内显

示) 规定屏幕上显示场景的范围和尺寸。

通过上面的几个步骤, 一个三维空间里的物体就可以用相应的二维平面物体表示了, 也就能在二维的电脑屏幕上正确显示了。总的来说, 三维物体的显示过程如下:



- 模型的变换和变换矩阵的几种类型

三维坐标变换原理-平移, 旋转, 缩放

平移矩阵:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

缩放矩阵:

$$M = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

旋转矩阵:

矩阵M为

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

绕Y轴旋转时

$$M = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

绕Z轴旋转时

$$M = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

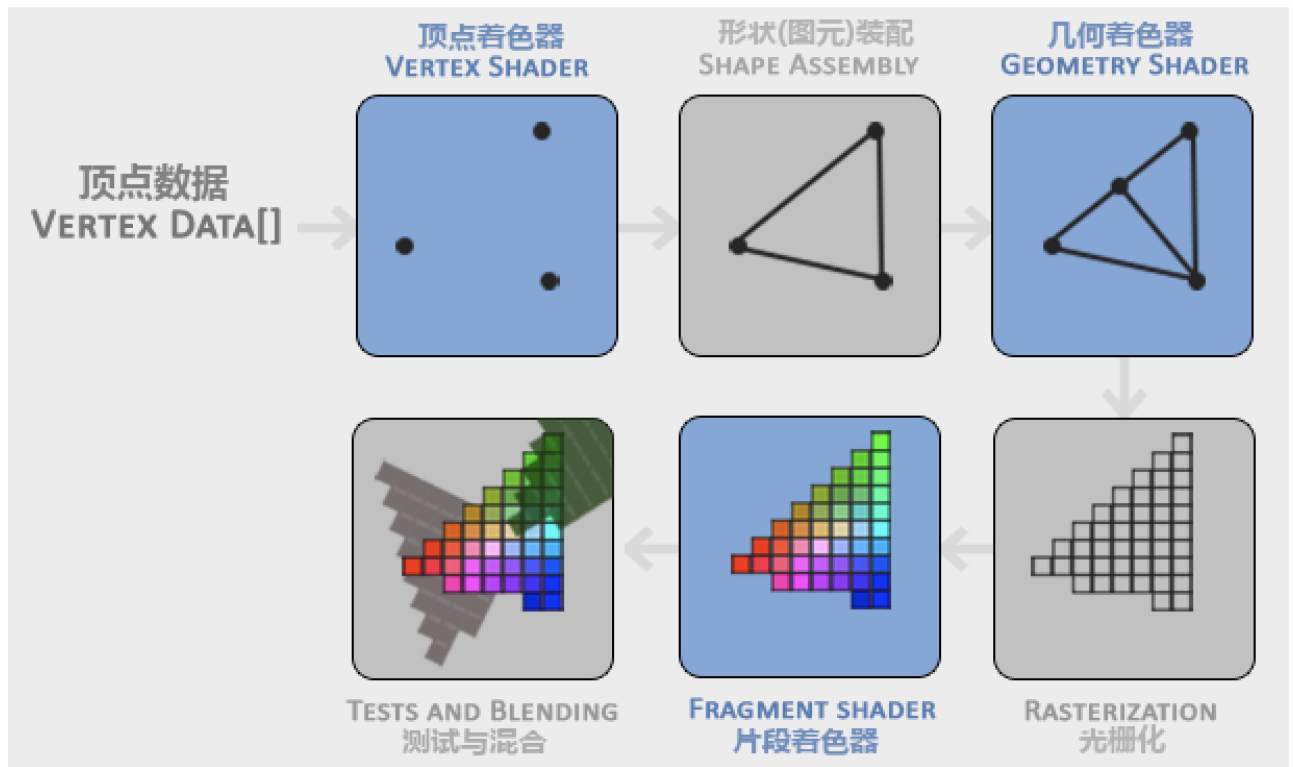
欧拉变换是绕3个旋转轴的旋转矩阵的乘积

变换矩阵必须是点乘，而且是右乘到原坐标上（原坐标是行的形式，即齐次坐标(x,y,z,1)）。

变换顺序必须是缩放->旋转->平移

- 原因：在物体刚刚放入世界坐标系的时候使用的是本地坐标，也就是本地和全局坐标系的原点和坐标轴都是重合的（当然两者分别使用了左右手坐标系时除外 - 那是BUG），此时所有物体都“把世界坐标系当做自己的本地坐标系”。
- 而经过了坐标变换之后：
  - 缩放变换不改变坐标轴的走向，也不改变原点的位置，所以两个坐标系仍然重合。
  - 旋转变换改变坐标轴的走向，但不改变原点的位置，所以两个坐标系坐标轴不再处于相同走向。
  - 平移变换不改变坐标轴走向，但改变原点位置，两个坐标系原点不再重合。

- OpenGL中，如何在屏幕上绘制一个三角形



- 对于任何一个模型，不管3D、2D，最初都是点的数据。比如一个正方形，就是4个角的坐标，立方体，就是8个点的坐标，一个游戏里的复杂人物，实际是许多的点拼接起来的，搜一下“三维模型”图片就有直观感受。所以整个流程输入的是顶点的数据，即Vertex Data。
- 而最终呈现给用户的是显示屏上的图像，而图像是一个个像素构成，所以输出的是每一个像素的颜色。整个流程要做的就是：怎么把一个个坐标数据变成屏幕上正确的像素颜色呢？
- 这张图片还是很直观的。而蓝色部分就是现代OpenGL可以让我们编写参与的部分。shader译作“着色器”，它是流程中的一段子程序，负责处理某一个阶段的任务，就像流水线上有很多不同的机器人和人，它们负责一部分工作。

1. 顶点数据(Vertex): 物体的3D坐标的数据的集合。
2. 顶点着色器(Vertex Shader)是第一个shader，它负责处理输入的顶点数据，比如坐标变换，把3D坐标转为另一种3D坐标，同时顶点着色器允许我们对顶点属性进行一些基本处理。
3. 图元装配(Primitive Assembly): 将顶点装配成形状。
4. 几何着色器(Geometry Shader): 通过产生新顶点或取其他顶点来生成新形状。
5. 光栅化阶段(Rasterization Stage): 将形状映射为最终屏幕上相应的像素，生成的东西叫片段(Fragment)。这个阶段会执行裁切，丢弃超出你的视图以外的所有像素，用来提升执行效率。（即投影）
6. 片段着色器(Fragment Shader): 计算一个像素的最终颜色。这时接受的已经不是顶点，而是fragment，有碎片的意思，它就对应着一个像素单位。这一阶段主要就是要计算颜色，比如光照计算：在有N个光源的时候，这个fragment的颜色是什么，光的颜色、物体本身的颜色、这个fragment朝向等都要考虑。
7. Alpha测试和混合(Blending): 进一步透明度、组合等处理。

- 对于大多数情况，只有三步需要我们操作：准备顶点数据，配置顶点着色器和片段着色器（因为GPU中没有默认的顶点/片段着色器）。

## Lua

- Lua 是一种轻量小巧的脚本语言，用标准C语言编写并以源代码形式开放，其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。
- 八种基本数据类型：nil、boolean、number、string、function、userdata、thread、table。
- 语句组(chunk)和语句块(block)的**区别**：一个chunk是一系列顺序执行的语句，chunk之间的运行是相互独立的，可以理解为一个函数；一个block仅仅是一系列语句，因此所有的chunk都是block，但不是所有的block都是chunk。

## Unity

### Untiy中的Monobehaviour

```
Start()
Update()
FixedUpdate()
LateUpdate()
OnGUI()
OnDisable()
OnEnable()
```

### UGUI和NGUI的区别

A	B	C	D
ngui与ugui优缺点及性能比较			
	ngui	ugui	对比结果
同一界面性能对比	1. drawcall:7-9 2. Used Textures:16.3 MB 3. VRAM usage :11.4MB-25.4MB 4. 发布后APK大小:19.8MB	1. drawcall:11-13 2. Used Textures:13.5 MB 3. VRAM usage :9.0MB-22.6MB 4. 发布后APK大小:18.5MB	同样界面(其中图片资源约为1MB),ugui的DrawCall高2个.但是内存等其他明显减小.安装包也小1.3MB
自适应	1. PixelPerfect 完美像素,直接显示设定好的像素.当屏幕高度低于minimum Height时按比例缩小,当屏幕高度大于maximum Height时按比例扩大. 2. FixedSize 按比例缩放,在设定好的基础上,直接按比例缩放. 3. FixedSizeOnMobiles 合体版, android和ios为FixedSize方式,其它按照PixelPerfect方式。	1. 有三种renderMode: overlay: 始终在最前面并充满屏幕,不可设置大小.可设置canvas层级显示的顺序 camera: 始终充满屏幕可设置摄像机,不可设置大小,不一定在最前可设置层级显示的顺序。 worldSpace: 可设置大小,可设置层级.可用于人物血条等的制作	ugui的更好用
text文本	1. 若需要的中文字体较少,可以减少最终安装包的大小。 2. 对中文支持不友好,需要用fontMaker, BMFont等工具制作成图集,比较麻烦。 3. 无法支持动态字体的图文混排。	1. 可直接导入ttf字体,无需制作成图集.支持富文本格式,方面易用。 2. 支持动态字体的图文混排。 3. 若需要的中文字体比较少,则会加大最终安装包的大小 4. 图文混排需要自己编写	ugui的性能更强大,对动态字体中文等支持更好,但是有些功能需要自己写组件
图集方面	1. 使用图片前必须要打成图集。 2. 可设置图集大小	1. 可直接使用小图..发布时自动打成图集.或者你也可以自己设置图集大小 2. 可设置图集大小 3. 可以使用material	ugui的更友好,但是无法在一个图集中动态获取其中的一个小图 无太大区别.
深度排序	1. 相同panel 相同atlas sprite受depth控制 2. 相同panel 不同atlas sprite受z轴控制(同时受1的影响) 3. 不同panel 相同atlas sprite受z轴控制(同时受1的影响) 4. 不同panel 不同atlas sprite受z轴控制(同时受1的影响)	在Hierarchy视图中,越在下方显示越靠前,可用SetSiblingIndex 和 GetSiblingIndex等来设置深度	ngui更复杂
事件系统			无太大区别.都可使用委托等
Tween缓动	有常用的一些缓动效果.	无	ngui自带缓动.ugui(需用插件,如dotween, itween等)

# C++

## C++中的模板类以及特化、偏特化

模板特化（**template specialization**）不同于模板的实例化，模板参数在某种特定类型下的具体实现称为模板特化。模板特化有时也称之为模板的具体化，分别有函数模板特化和类模板特化。

- 函数模板特化

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

template<typename T> T Max(T t1,T t2)
{
    return (t1>t2)?t1:t2;
}

typedef const char* CCP;
template<> CCP Max<CCP>(CCP s1,CCP s2)
{
    return (strcmp(s1,s2)>0)?s1:s2;
}

int main()
{
    //隐式调用实例: int Max<int>(int,int)
    int i=Max(10,5);

    //显式调用特化版本: const char* Max<const char*>(const char*,const char*)
    const char* p=Max<const char*>("very","good");
    cout<<"i:"<<i<<endl; // 输出 10
    cout<<"p:"<<p<<endl; // 输出 very
}
```

在函数模板显式特化定义（**Explicit Specialization Definition**）中，显式关键字**template**和一对尖括号**<>**，然后是函数模板特化的定义。该定义指出了模板名、被用来特化模板的模板实参，以及函数参数表和函数体。在上面的程序中，如果不给出函数模板**Max< T>**在**T**为**const char\***时的特化版本，那么在比较两个字符串的大小时，比较的是字符串的起始地址的大小，而不是字符串的内容在字典序中的先后次序。

- 类模板特化

类模板特化类似于函数模板的特化，即类模板参数在某种特定类型下的具体实现。

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

template<typename T>class A
{
```

```

    T num;
public:
    A()
    {
        num=T(6.6);
    }
    void print()
    {
        cout<<"A'num:"<<num<<endl;
    }
};

template<> class A<char*>
{
    char* str;
public:
    A(){
        str="A' special definition ";
    }
    void print(){
        cout<<str<<endl;
    }
};

int main()
{
    A<int> a1;        //显示模板实参的隐式实例化
    a1.print(); // 输出 6
    A<char*> a2;      //使用特化的类模板
    a2.print(); // 输出 A' special definition
}

```

- 模板偏特化

模板偏特化（Template Partial Specialization）是模板特化的一种特殊情况，指显示指定部分模板参数而非全部模板参数，或者指定模板参数的部分特性分而非全部特性，也称为模板部分特化。与模板偏特化相对的是模板全特化，指对所有的模板参数进行特化。模板全特化与模板偏特化共同组成模板特化。

模板偏特化主要分为两种，一种是指对部分模板参数进行全特化，另一种是对模板参数特性进行特化，包括将模板参数特化为指针、引用或是另外一个模板类。

- 函数模板偏特化

假如我们有一个compare函数模板，在比较数值大小时没有问题，如果传入的是数值的地址，我们需要比较两个数值的大小，而非比较传入的地址大小。此时我们需要对compare函数模板进行偏特化。考察如下代码：

```

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

```

```

//函数模板
template<typename T, class N> void compare(T num1, N num2)
{
    cout << "standard function template" << endl;
    if(num1>num2)
        cout << "num1:" << num1 << " > num2:" << num2 <<endl;
    else
        cout << "num1:" << num1 << " <= num2:" << num2 << endl;
}

//对部分模板参数进行特化
template<class N> void compare(int num1, N num2)
{
    cout<< "partial specialization" <<endl;
    if (num1>num2)
        cout << "num1:" << num1 << " > num2:" << num2 << endl;
    else
        cout << "num1:" << num1 << " <= num2:" << num2 << endl;
}

//将模板参数特化为指针
template<typename T, class N> void compare(T* num1, N* num2)
{
    cout << "new partial specialization" << endl;
    if (*num1>*num2)
        cout << "num1:" << *num1 << " > num2:" << *num2 << endl;
    else
        cout << "num1:" << *num1 << " <= num2:" << *num2 << endl;
}

//将模板参数特化为另一个模板类
template<typename T, class N> void compare(std::vector<T>& vecLeft,
std::vector<T>& vecRight)
{
    cout << "to vector partial specialization" << endl;
    if (vecLeft.size()>vecRight.size())
        cout << "vecLeft.size()" << vecLeft.size() << " > vecRight.size():"
<< vecRight.size() << endl;
    else
        cout << "vecLeft.size()" << vecLeft.size() << " <= vecRight.size():"
<< vecRight.size() << endl;
}

int main()
{
    compare<int,int>(30,31); //调用非特化版本compare<int,int>(int num1, int
num2)

    compare(30,'1'); //调用偏特化版本compare<char>(int num1, char
num2)

    int a = 30;
    char c = '1';
    compare(&a,&c); //调用偏特化版本compare<int,char>(int* num1,

```



```

char* num2)

    vector<int> vecLeft{0};
    vector<int> vecRight{1,2,3};
    compare<int,int>(vecLeft,vecRight); //调用偏特化版本compare<int,char>(int*
num1, char* num2)
}

```

输出结果:

```

standard function template
num1:30 <= num2:31
partitital specialization
num1:30 <= num2:1
new partitital specialization
num1:30 <= num2:1
to vector partitital specialization
vecLeft.size()1 <= vecRight.size():3

```

- 类模板偏特化

```

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

//类模板
template<typename T, class N> class TestClass
{
public:
    static bool comp(T num1, N num2)
    {
        cout <<"standard class template"<< endl;
        return (num1<num2) ? true : false;
    }
};

//对部分模板参数进行特化
template<class N> class TestClass<int, N>
{
public:
    static bool comp(int num1, N num2)
    {
        cout << "partitital specialization" << endl;
        return (num1<num2) ? true : false;
    }
};

//将模板参数特化为指针
template<typename T, class N> class TestClass<T*, N*>

```

```

{
public:
    static bool comp(T* num1, N* num2)
    {
        cout << "new partial specialization" << endl;
        return (*num1<*num2) ? true : false;
    }
};

//将模板参数特化为另一个模板类
template<typename T, class N> class TestClass<vector<T>,vector<N>>
{
public:
    static bool comp(const vector<T>& vecLeft, const vector<N>& vecRight)
    {
        cout << "to vector partial specialization" << endl;
        return (vecLeft.size()<vecRight.size()) ? true : false;
    }
};

int main()
{
    //调用非特化版本
    cout << TestClass<char, char>::comp('0', '1') << endl;

    //调用部分模板参数特化版本
    cout << TestClass<int,char>::comp(30, '1') << endl;

    //调用模板参数特化为指针版本
    int a = 30;
    char c = '1';
    cout << TestClass<int*, char*>::comp(&a, &c) << endl;

    //调用模板参数特化为另一个模板类版本
    vector<int> vecLeft{0};
    vector<int> vecRight{1,2,3};
    cout << TestClass<vector<int>, vector<int>>::comp(vecLeft,vecRight) <<
endl;
}

```

输出结果:

```

standard class template
1
partial specialization
1
new partial specialization
1
to vector partial specialization
1

```

- 模板类调用优先级：

对主版本模板类、全特化类、偏特化类的调用优先级从高到低进行排序是：全特化类>偏特化类>主版本模板类。这样的优先级顺序对性能也是最好的。

但是模板特化并不只是为了性能优化，更多是为了让模板函数能够正常工作，最典型的例子就是STL中的`iterator_traits`。`algorithm`中大多数算法通过`iterator`对象来处理数据，但是同时允许以指针代替`iterator`对象，这是为了支持C-Style Array。如果直接操作`iterator`，那么为了支持指针类型，每个算法函数都需要进行重载，因为指针没有`::value_type`类型。为了解决这个问题，STL使用了`iterator_traits`对`iterator`特性进行封装，并为指针类型做了偏特化处理，算法通过它来操作`iterator`，不需要知道实际操作的是`iterator`对象还是指针。

```
template<typename IteratorClass> class iterator_traits
...
template<typename ValueType> class iterator_traits<ValueType*>
...
template<typename ValueType> class iterator_traits<ValueType const*>
...
```

后面两是针对指针类型的偏特化，也是偏特化的一种常见形式。

## 深浅拷贝

C++中类的拷贝有两种：深拷贝，浅拷贝：当出现类的等号赋值时，即会调用拷贝函数

- 在未定义显式拷贝构造函数的情况下，系统会调用默认的拷贝函数——即浅拷贝，它能够完成成员的一一复制。当数据成员中没有指针时，浅拷贝是可行的；但当数据成员中有指针时（有对其他资源（如堆、文件、系统等）的引用时（引用可以是指针或引用）），如果采用简单的浅拷贝，则两类中的两个指针将指向同一个地址，当对象快结束时，会调用两次析构函数，而导致指针悬挂现象（指针指向非法的内存地址，那么这个指针就是悬挂指针，也叫野指针，意为无法正常使用的指针。它与内存泄漏正好是相反的概念），所以，此时必须采用深拷贝。
- 如果在类中没有显式地声明一个拷贝构造函数，编译器将会自动生成一个默认的拷贝构造函数，该构造函数完成对象之间的位拷贝，位拷贝又称浅拷贝。深拷贝与浅拷贝的区别就在于深拷贝会在堆内存中另外申请空间来储存数据，从而也就解决了指针悬挂的问题。简而言之，当数据成员中有指针时，必须要用深拷贝。
- 写法：

```
class MyString
{
private:
    char *m_data;
    int m_length;

public:
    MyString(const char *source="")
    {
        assert(source); // make sure source isn't a null string
    }
}
```

```

        // Find the length of the string
        // Plus one character for a terminator
        m_length = std::strlen(source) + 1;

        // Allocate a buffer equal to this length
        m_data = new char[m_length];

        // Copy the parameter string into our internal buffer
        for (int i=0; i < m_length; ++i)
            m_data[i] = source[i];

        // Make sure the string is terminated
        m_data[m_length-1] = '\0';
    }

    ~MyString() // destructor
    {
        // We need to deallocate our string
        delete[] m_data;
    }

    // Copy constructor
    MyString(const MyString& source)
    {
        // because m_length is not a pointer, we can shallow copy it
        m_length = source.m_length;

        // source.m_data is a pointer, so we need to deep copy it if it
        // is non-null
        // 注意这里判断的是source，也就是赋值对象的指针域，而不是被赋值对象，下同
        if (source.m_data)
        {
            // allocate memory for our copy
            m_data = new char[m_length];

            // do the copy
            for (int i=0; i < m_length; ++i)
                m_data[i] = source.m_data[i];
        }
        else
            m_data = nullptr;
    }

    // Assignment operator
    MyString& operator=(const MyString & source)
    {
        // check for self-assignment
        if (this == &source)
            return *this;

        // first we need to deallocate any value that this string is
        // holding!
        delete[] m_data;
    }

```

```

        // because m_length is not a pointer, we can shallow copy it
        m_length = source.m_length;

        // m_data is a pointer, so we need to deep copy it if it is non-
null
        if (source.m_data)
        {
            // allocate memory for our copy
            m_data = new char[m_length];

            // do the copy
            for (int i=0; i < m_length; ++i)
                m_data[i] = source.m_data[i];
        }
        else
            m_data = nullptr;

        return *this;
    }

    char* getString() { return m_data; }
    int getLength() { return m_length; }

```

- 从上面可以看到，如果类内存在指针或动态申请的内存，我们就需要深拷贝，而定义深拷贝时，往往还需要写其他相关的函数：

1. 构造函数：需要动态申请内存
2. 拷贝构造函数：同样需要动态申请内存
3. 析构函数：释放动态申请的内存
4. 重载等于号：处理赋值操作，需要释放内存后重新申请

- 重载等于号和拷贝构造函数很像，因为参数都是另外一个实例，但主要有三点区别：

1. 需要检查是否是自我赋值
2. 返回值是 `*this`
3. 需要先释放内存，再重新申请，防止内存泄漏

本质上是因为赋值和构造的区别，开始构造时，待构造对象一定是空的，而赋值之前，待赋值对象则可能有值。

- C++中的STL内部已经定义好了深拷贝，如vector、string等，所以可以放心使用，减少错误的概率
- 仿照上面的写法写一个栈，支持pop、push、top以及拷贝、赋值等操作

```

#include <iostream>

using namespace std;

class MyStack
{

```

```
private:
    int *m_data;
    int m_length;

public:
    MyStack()
    {
        m_length = 0;

        m_data = new int[1000];
    }

    ~MyStack() // destructor
    {
        delete[] m_data;
    }

    // Copy constructor
    MyStack(const MyStack &source) // 注意引用
    {
        m_length = source.m_length;

        // 防止爆栈
        if (source.m_data && m_length < 1000)
        {
            // allocate memory for our copy
            m_data = new int[1000];

            // do the copy
            for (int i=0; i < m_length; ++i)
                m_data[i] = source.m_data[i];
        }
        else
            m_data = NULL;
    }

    // Assignment operator
    MyStack& operator=(const MyStack &source) // 注意引用
    {
        // check for self-assignment
        if (this == &source)
            return *this;

        delete[] m_data;

        // because m_length is not a pointer, we can shallow copy it
        m_length = source.m_length;

        // 防止爆栈
        if (source.m_data && m_length < 1000)
        {
            // allocate memory for our copy
            m_data = new int[1000];
```

```
        // do the copy
        for (int i=0; i < m_length; ++i)
            m_data[i] = source.m_data[i];
    }
    else
        m_data = NULL;

    return *this;
}

void push(int n)
{
    if(m_length == 1000) return;
    m_data[m_length] = n;
    ++m_length;
}
void pop()
{
    if(m_length == 0) return;
    --m_length;
}
int top()
{
    if(m_length==0) return -1;
    return m_data[m_length-1];
}
void print()
{
    for(int i=0; i<m_length; i++)
    {
        cout<<m_data[i]<<' ';
    }
    cout<<endl;
}
};

int main()
{
    MyStack stk;
    stk.push(1);
    stk.push(2);
    stk.push(3);
    stk.print();
    stk.push(4);
    stk.print();
    stk.pop();
    stk.pop();
    stk.print();
    stk.push(5);
    stk.print();
    MyStack stk2 = stk;
    stk2.print();
    return 0;
}
```

## 内存泄漏、内存溢出、内存碎片

- 内存泄漏（Memory Leak）：是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。
- 内存溢出（Out of Memory）：是指程序在申请内存时，没有足够的内存空间供其使用。
- 区别：内存泄漏是申请内存后没有释放，但它的地址已经找不到了，所以这块内存无法再被使用，也无法再被回收。内存溢出就是简单的分配空间不足以放下数据，比如栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。
- 同时还需要知道内存泄漏和指针悬挂的区别，见上文。
- 内存碎片：描述一个系统中所有不可用的空闲内存，即采用分区式存储管理的系统，在储存分配过程中产生的、不能供用户作业使用的主存里的小分区称成“内存碎片”。内存碎片分为内部碎片和外部碎片。内部碎片就是已经被分配出去（能明确指出属于哪个进程）却不能被利用的内存空间，通常内部碎片难以完全避免；外部碎片指的是还没有被分配出去（不属于任何进程），但由于太小了无法分配给申请内存空间的新进程的内存空闲区域。

## new/delete 和 malloc/free

- malloc与free是C++/C 语言的标准库函数（function），new/delete 是C++的运算符（operator）。
- delete 用于释放 new 分配的空间，free 有用释放 malloc 分配的空间。
- 调用free之前需要检查需要释放的指针是否为空，使用delete 释放内存则不需要检查指针是否为NULL。
- 实例：

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int x;
    int *ptr1 = &x;
    int *ptr2 = (int *)malloc(sizeof(int));
    int *ptr3 = new int;
    int *ptr4 = NULL;

    /* delete Should NOT be used like below because x is allocated
       on stack frame */
    delete ptr1;

    /* delete Should NOT be used like below because x is allocated
       using malloc() */
    delete ptr2;

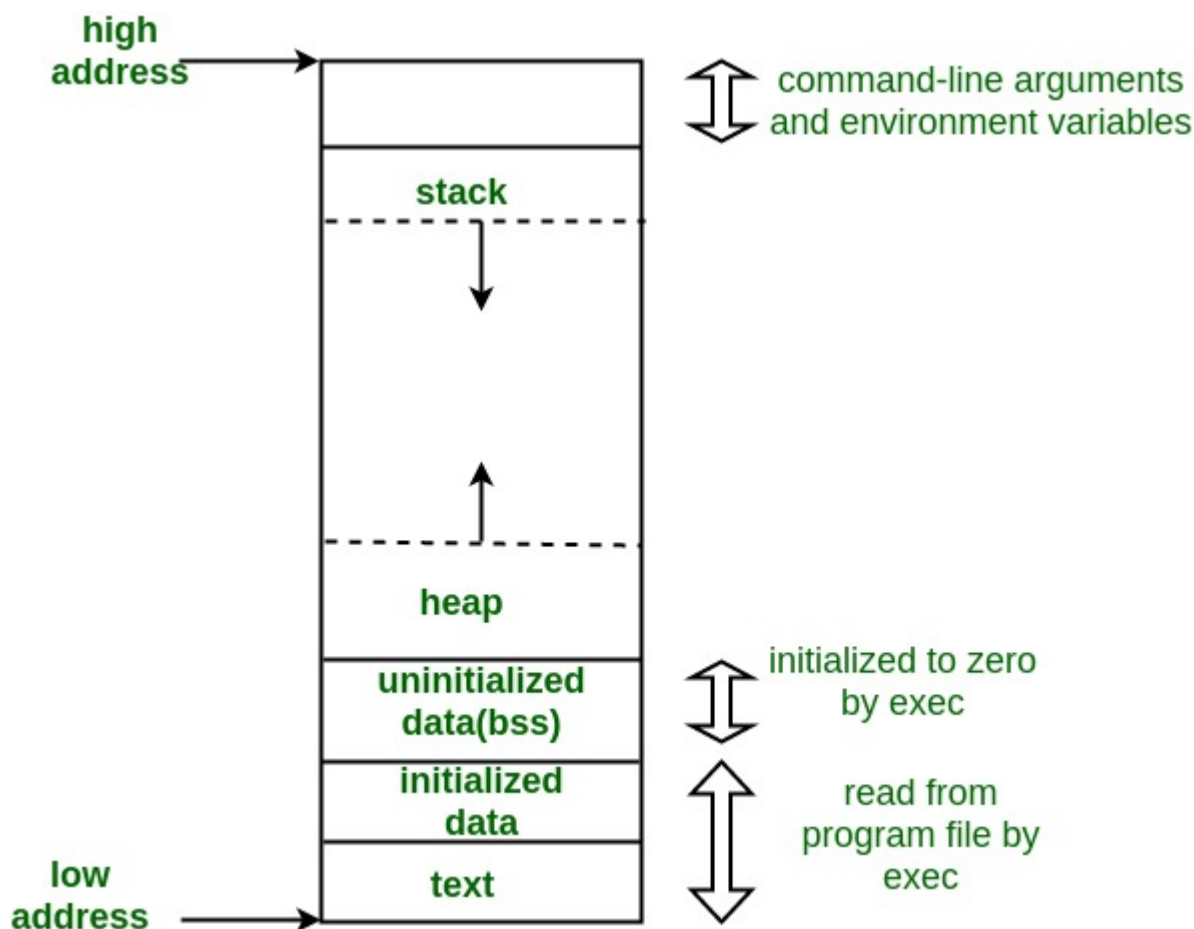
    /* Correct uses of delete */
    delete ptr3;
    delete ptr4;

    getchar();
}
```



```
return 0;
}
```

## C/C++中的内存布局



- 内存分为五个部分
  - 代码区(Text segment): 存放CPU执行的机器指令 (machine instructions)。通常，代码区是可共享的 (即另外的执行程序可以调用它)，因为对于频繁被执行的程序，只需要在内存中有一份代码即可。代码区通常是只读的，使其只读的原因是防止程序意外地修改了它的指令。另外，代码区还规划了局部变量的相关信息。
  - 未初始化数据区 (Uninitialized Data Segment): 亦称BSS区 (uninitialized data segment)，存入的是全局未初始化变量。BSS这个叫法是根据一个早期的汇编运算符而来，这个汇编运算符标志着一个块的开始。BSS区的数据在程序开始执行之前被内核初始化为0或者空指针 (NULL)。例如一个不在任何函数内的声明：

```
long sum[1000];
```

- 全局初始化数据区/静态数据区 (Initialized Data Segment): 该区包含了在程序中明确被初始化的全局变量、静态变量 (包括全局静态变量和局部静态变量) 和常量数据 (如字符串常量)。例如，一个不在任何函数内的声明 (全局数据)：

```
int maxcount = 99;
```

使得变量maxcount根据其初始值被存储到初始化数据区中。

```
static mincount = 100;
```

这声明了一个静态数据，如果是在任何函数体外声明，则表示其为一个全局静态变量，如果在函数体内（局部），则表示其为一个局部静态变量。另外，如果在函数名前加上static，则表示此函数只能在当前文件中被调用。

- 堆区（Heap）：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。

```
int main()
{
    // This memory for 10 integers
    // is allocated on heap.
    int *ptr = new int[10];
}
```

- 栈区（Stack）：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，比在堆上申请一个空间要快，但是分配的内存容量有限。

```
int main()
{
    // All these variables get memory
    // allocated on stack
    int a;
    int b[10];
    int n = 20;
    int c[n];
}
```

## 引用和指针的区别

- 相同点：

1. 都是地址的概念； 指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名。

- 不同点：

1. 引用不能为空，指针可以为空；
2. 指针是一个实体，而引用仅是个别名；
3. 引用使用时无需解引用(\*)，指针需要解引用；
4. 引用只能在定义时被初始化一次，之后不可变；指针可变；

5. 引用没有 `const`，指针有 `const`（具体指没有 `int& const a` 这种形式，而 `const int& a` 是有的，前者指引用本身即别名不可以改变，这是当然的，所以不需要这种形式，后者指引用所指的值不可以改变）；
6. “`sizeof` 引用”得到的是所指向的变量(对象)的大小，而“`sizeof 指针`”得到的是指针本身(所指向的变量或对象的地址)的大小；
7. 指针和引用的自增(++)运算意义不一样；
8. 从内存分配上看：程序为指针变量分配内存区域，而引用不需要分配内存区域。
9. 引用是类型安全的，而指针不是（引用比指针多了类型检查）。因此没有指针的语言都是类型安全的语言。

## 值传递、指针传递、引用传递

- 值传递：形参是实参的拷贝，改变形参的值并不会影响外部实参的值。从被调用函数的角度来说，值传递是单向的（实参->形参），参数的值只能传入，不能传出。当函数内部需要修改参数，并且不希望这个改变影响调用者时，采用值传递。
- 指针传递：形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作
- 引用传递：形参相当于是实参的“别名”，对形参的操作其实就是对实参的操作，在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。
- 来自大佬的一句话，“引用本质上就是指针，C++语言中所有传递都是值传递，只不过引用传递是编译器带来的检查”。[解释一下就是](#)，引用实际是通过指针实现的，是一个常量指针，因此引用在内存中占4个字节。在对引用定义时，需要对这个常量指针初始化。C++设计引用，并用常量指针来从编译器的角度实现它，目的是为了提供比指针更高的安全性，因为常量指针一旦与变量地址绑定将不能更改，这样降低了指针的危险系数，它提供了一种一对一的指针。
- 关于引用占用内存的大小，可以用下面的代码进行验证：
  - 因为在表达式中，使用引用实际上就像使用变量本身一样，所以直接用`sizeof`是得不到引用本身的大小的。

```
double var = 42.0;
double& ref = var;
cout << sizeof var << endl; // print 8
cout << sizeof ref << endl; // print 8
```

- 我们可以通过定义一个只含有引用的类来解决这个问题：

```
class refClass{
private:
    double& ref;
public:
    refClass(double var = 42.0) :ref(var){}
};
cout << sizeof refClass << endl; // print 4
```

## 左值、右值、左值引用、右值引用

- 左值是可以放在赋值号左边可以被赋值的值；左值必须要在内存中有实体；右值当在赋值号右边取出值赋给其他变量的值；右值可以在内存也可以在CPU寄存器。一个对象被用作右值时，使用的是它的内容(值)，被当作左值时，使用的是它的地址。
- 只能绑定到左值，不能绑定到右值的引用是左值引用。只能绑定到右值，而不能绑定到左值的引用是右值引用。
- 右值引用就是必须绑定到右值（一个临时对象、将要销毁的对象）的引用，一般表示对象的值。
- 右值引用可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding），它的主要目的有两个方面：
  - 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
  - 能够更简洁明确地定义泛型函数。
- 左值引用的基本语法：`type &引用名 = 左值表达式`。
- 右值引用的基本语法：`type &&引用名 = 右值表达式`。右值引用的“&&”中间不可以有空格。

```
int & i = 42; // 错误，因为42是右值，而i是左值引用，所以i不能绑定到42，下面这样就可以
int j = 42;

int &i = j; // 正确，j是左值，所以i可以绑定到j;

int && i = 42; // 正确，&&表示右值引用，所以i可以绑定到右值42

int j = 42;

int &&i = j; // 错误，因为j是左值，而i是右值引用，所以不能绑定到j。
```

如何找到一个变量的右值引用？待整理，目前还不知道是什么意思。

`std::move`和`std::forward`

`placement new` 和 `operator new`  
<https://www.cnblogs.com/slgkaifa/p/6887887.html>

## Const

- 修饰变量，说明该变量不可以被改变；
- 修饰指针，分为指向常量的指针和指针常量；
- 常量引用，经常用于形参类型，即避免了拷贝，又避免了函数对值的修改；
- 修饰成员函数，说明该成员函数内不能修改成员变量。
- `const`和`private`等，在最终的代码中并不存在，仅仅是编译器提供给程序员的一种安全的方式。

```

// 类
class A
{
private:
    const int a;                // 常对象成员，只能在初始化列表赋值

public:
    // 构造函数
    A() : a(0) { };
    A(int x) : a(x) { };        // 初始化列表

    // const可用于对重载函数的区分
    int getValue();             // 普通成员函数
    int getValue() const;       // 常成员函数，不得修改类中的任何数据成员的值
};

void function()
{
    // 对象
    A b;                        // 普通对象，可以调用全部成员函数、更新常成员变量
    const A a;                  // 常对象，只能调用常成员函数
    const A *p = &a;            // 常指针
    const A &q = a;              // 常引用

    // 指针
    char greeting[] = "Hello";
    char* p1 = greeting;        // 指针变量，指向字符数组变量
    const char* p2 = greeting;   // 指针变量，指向字符数组常量
    char* const p3 = greeting;   // 常指针，指向字符数组变量
    const char* const p4 = greeting; // 常指针，指向字符数组常量
}

// 函数
void function1(const int Var);   // 传递过来的参数在函数内不可变
void function2(const char* Var); // 参数指针所指内容为常量
void function3(char* const Var); // 参数指针为常指针
void function4(const int& Var);  // 引用参数在函数内为常量

// 函数返回值
const int function5();           // 返回一个常数
const int* function6();          // 返回一个指向常量的指针变量，使用：const int *p =
function6();
int* const function7();          // 返回一个指向变量的常指针，使用：int* const p =
function7();

```

## Static

- 修饰普通变量，修改变量的存储区域和生命周期，使变量存储在静态区，在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。
- 修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。

- 修饰成员变量，修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
- 修饰成员函数，修饰成员函数使得不需要生成对象就可以访问该函数，但是在 `static` 函数内不能访问非静态成员。

## volatile

```
volatile int i = 10;
```

- `volatile` 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素（操作系统、硬件、其它线程等）更改。所以使用 `volatile` 告诉编译器不应对这样的对象进行优化。
- `volatile` 关键字声明的变量，每次访问时都必须从内存中取出值（没有被 `volatile` 修饰的变量，可能由于编译器的优化，从 CPU 寄存器中取值）
- `const` 可以是 `volatile`（如只读的状态寄存器）
- 指针可以是 `volatile`

## explicit

- `explicit` 修饰构造函数时，可以防止隐式转换和复制初始化
- `explicit` 修饰转换函数时，可以防止隐式转换，但按语境转换除外

```
struct A
{
    A(int) { }
    operator bool() const { return true; }
};

struct B
{
    explicit B(int) {}
    explicit operator bool() const { return true; }
};

void doA(A a) {}

void doB(B b) {}

int main()
{
    A a1(1);           // OK: 直接初始化
    A a2 = 1;          // OK: 复制初始化
    A a3{ 1 };         // OK: 直接列表初始化
    A a4 = { 1 };      // OK: 复制列表初始化
    A a5 = (A)1;       // OK: 允许 static_cast 的显式转换
    doA(1);            // OK: 允许从 int 到 A 的隐式转换
    if (a1);           // OK: 使用转换函数 A::operator bool() 的从 A 到
bool 的隐式转换
    bool a6 (a1);      // OK: 使用转换函数 A::operator bool() 的从 A 到
bool 的隐式转换
    bool a7 = a1;      // OK: 使用转换函数 A::operator bool() 的从 A 到
bool 的隐式转换
```

```

    bool a8 = static_cast<bool>(a1); // OK : static_cast 进行直接初始化

    B b1(1); // OK: 直接初始化
    B b2 = 1; // 错误: 被 explicit 修饰构造函数的对象不可以复制初始
化
    B b3{ 1 }; // OK: 直接列表初始化
    B b4 = { 1 }; // 错误: 被 explicit 修饰构造函数的对象不可以复制列表
初始化
    B b5 = (B)1; // OK: 允许 static_cast 的显式转换
    doB(1); // 错误: 被 explicit 修饰构造函数的对象不可以从 int
到 B 的隐式转换
    if (b1); // OK: 被 explicit 修饰转换函数 B::operator bool()
的对象可以从 B 到 bool 的按语境转换
    bool b6(b1); // OK: 被 explicit 修饰转换函数 B::operator bool()
的对象可以从 B 到 bool 的按语境转换
    bool b7 = b1; // 错误: 被 explicit 修饰转换函数 B::operator
bool() 的对象不可以隐式转换
    bool b8 = static_cast<bool>(b1); // OK: static_cast 进行直接初始化

    return 0;
}

```

## C++中new的过程

- 在内存（自由存储区）中开辟空间；
- 将this指向该空间地址；
- 通过this执行构造函数，给该空间添加属性和方法；
- 把this返回给外部变量；

## allocator

### 详细解析

对C++的new而言，它首先会(1)分配内存，然后自动的完成(2)对象构造。这里可以用侯捷先生翻译的《深度探索C++对象模型》一书中的伪代码来表示new的过程：

```

Point* heap = __new(sizeof(Point)); //开辟内存
if (head != 0) {
    head->Point::Point(); //对象构造（内存构造）
}

```

注意，\_\_new不表示new（它只是完成内存申请），以上整个伪代码过程为new所完成的功能。正是因为new的这一连串的操作，造成了性能的下降。比如，

```

auto p = new string[100];
for (int i = 0; i < 5; ++i){

```

```

        p[i] = "balaba...";
    }

```

实际上，我只需要5个string，而new把100个对象全部构造好了（每个string已经被初始化为空字符串，也就是""）。

然后，接着又将p[0-4]赋值为balaba...

也就是前面将p[0-4]赋值为空字符串的操作，变得毫无意义。

既然，new有它自身的局限性。对于性能要求极高的STL肯定是不使用new的。好在有一个allocator类——它也是一个模板类，同时就是用来处理内存问题的，使用allocator可以将内存分配、对象构造分离开：

allocator类将new的内存分配、对象构造，视作两个独立的过程，并由独立的函数负责。举个例子：

```

allocator<char> str;
char* base = str.allocate(10), *p = base; //内存分配
str.construct(p++, 'a'); //对象构造并初始化
str.construct(p++, 'b');
cout << base[0] << base[1];

```

因为allocator是模板类，所以需要指定类型。接着，调用allocate(10)函数来分配内存（申请了10个char内存）。然后，使用construct函数构造base[0]这块内存，并赋以初值a。

这便将new内存分配、内存构造给分离开了。一切，都像我们看到的那样。

同样，将delete的过程也拆分了开来。这是必须的，我们不能用delete去释放allocate分配的内存。

```

str.destroy(--p); //销毁对象
str.destroy(--p);
str.deallocate(base, 10); //释放内存

```

## C++中new失败的处理

- C语言中，使用 malloc/calloc 等分配内存的函数时，一定要检查其返回值是否为“空指针”（亦即检查分配内存的操作是否成功），这是良好的编程习惯，也是编写可靠程序所必需的。但是，如果你简单地把这一招应用到 new 上，那可就不一定正确了。我经常看到类似这样的代码：

```

int* p = new int[SIZE];
if ( p == 0 ) // 检查 p 是否空指针
    return -1;
// 其它代码

```

其实，这里的 if (p == 0) 完全是没啥意义的。



- C++ 里，如果 `new` 分配内存失败，默认是抛出异常(`bad_alloc`)的。所以，如果分配成功，`p == 0` 就绝对不会成立；而如果分配失败了，也不会执行 `if (p == 0)`，因为分配失败时，`new` 就会抛出异常跳过后面的代码。如果你想检查 `new` 是否成功，应该捕捉异常：

```
try {
    int* p = new int[SIZE];
    // 其它代码
} catch ( const bad_alloc& e ) {
    return -1;
}
```

- 当然，标准 C++ 亦提供了一个方法来抑制 `new` 抛出异常，而返回空指针

```
int* p = new (std::nothrow) int; // 这样如果 new 失败了，就不会抛出异常，而是返回空指针
if ( p == 0 ) // 如此这般，这个判断就有意义了
    return -1;
// 其它代码
```

- 除了捕捉异常外，还有一种方法可以处理 `new` 失败的情况，那就是重新申请，使用 `set_new_handler` 函数处理 `new` 失败。`set_new_handler` 的输入参数是 `operator new` 分配内存失败时要调用的出错处理函数的指针，返回值是 `set_new_handler` 没调用之前就已经在起作用的旧的出错处理函数的指针。可以像下面这样使用 `set_new_handler`：

```
void nomorememory()
{
    cerr << "unable to satisfy request for memory\n";
    abort();
}
int main()
{
    set_new_handler(nomorememory);
    int *pbigdataarray = new int[100000000];
    ...
}
```

- `operator new` 不能满足内存分配请求时，`new-handler` 函数不只调用一次，而是不断重复，直至找到足够的内存。一个设计得好的 `new-handler` 函数必须实现下面功能中的一种：
  - 产生更多的可用内存。这将使 `operator new` 下一次分配内存的尝试有可能获得成功。实施这一策略的一个方法是：在程序启动时分配一个大的内存块，然后在第一次调用 `new-handler` 时释放。释放时伴随着一些对用户的警告信息，如内存数量太少，下次请求可能会失败，除非又有更多的可用空间。
  - 安装另一个不同的 `new-handler` 函数。如果当前的 `new-handler` 函数不能产生更多的可用内存，可能它会知道另一个 `new-handler` 函数可以提供更多的资源。这样的话，当前的 `new-handler` 可以安装另一个 `new-handler` 来取代它(通过调用 `set_new_handler`)。下一次 `operator new` 调用 `new-`

handler时，会使用最近安装的那个。(这一策略的另一个变通办法是让new-handler可以改变它自己的运行行为，那么下次调用时，它将做不同的事。方法是使new-handler可以修改那些影响它自身行为的静态或全局数据。)

- 卸除new-handler。也就是传递空指针给set\_new\_handler。没有安装new-handler，operator new分配内存不成功时就会抛出一个标准的std::bad\_alloc类型的异常。
- 抛出std::bad\_alloc或从std::bad\_alloc继承的其他类型的异常。这样的异常不会被operator new捕捉，所以它们会被送到最初进行内存请求的地方。(抛出别的不同类型的异常会违反operator new异常规范。规范中的缺省行为是调用abort，所以new-handler要抛出一个异常时，一定要确信它是从std::bad\_alloc继承来的。)
- 没有返回。典型做法是调用abort或exit。abort/exit可以在标准c库中找到(还有标准c++库)。

## 智能指针

由于 C++ 语言没有自动内存回收机制，程序员每次 new 出来的内存都要手动 delete。如果流程太复杂，很容易最终导致没有 delete，而抛出异常导致程序过早退出，并造成内存泄露。如此c++引入智能指针，在<memory>头文件的 std 命名空间中定义。引入智能指针的作用是为了保证在出现异常时，也能保证堆内存会被释放掉，同时智能指针使用了引用计数的想法，从此我们不再需要手动释放内存。主要包括：

std::shared\_ptr, std::unique\_ptr, std::weak\_ptr

- shared\_ptr: 多个shared\_ptr可指向同一个动态对象，并维护了一个共享的引用计数器，记录了引用同一对象的shared\_ptr实例的数量。当最后一个指向动态对象的shared\_ptr销毁时，会自动销毁其所指对象(通过delete操作符)，从而不需要手动调用 delete，当引用计数变为零的时候就会自动的将对象删除。使用std::make\_shared会创建传入参数中的对象，并返回这个对象类型的shared\_ptr指针：

```
auto pointer = std::make_shared<int>(10);
```

当pointer离开其作用域的时候，会自动释放资源。就这样，程序中再也看不见 new和 delete了

- unique\_ptr: 一种独占的智能指针，它禁止其他智能指针与其共享同一个对象，从而保证了代码的严格安全，既然是独占，就不要想可以复制了。但是，这里可以使用 std::move将对象所有权转移给其他的unique\_ptr，其他的复制方式就是不允许的了。
- weak\_ptr: 如果我们仔细想一下最开始介绍的 shared\_ptr的话，其实有这么一种情况是无法释放资源的：对象1和对象2都由智能指针管理，对象1的智能指针成员指向对象2，对象2的智能指针成员指向对象1。那么，在离开对象1和对象2的作用域时，对象1和对象2的引用是没有了，但是对象的智能指针成员还是互相引用着，这样就造成了循环引用，导致资源无法释放。上面循环引用问题的解决办法就是使用weak\_ptr，weak\_ptr是一种弱引用（与 shared\_ptr对比而言），弱引用不会引起智能指针的引用计数增加。比如上面的问题，我们只要把对象的智能指针成员由 shared\_ptr类型换为 weak\_ptr类型就可以解决。

示例：注意程序中共有4个智能指针，两个指向A和B的实例，每个实例中还各有一个智能指针，指向另一个类。

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

```

class B;
class A
{
public:// 为了省去一些步骤这里 数据成员也声明为public
    //weak_ptr<B> pb;
    shared_ptr<B> pb;
    void doSomething()
    {
//        if(pb.lock())
//        {
//
//        }
    }

    ~A()
    {
        cout << "kill A\n";
    }
};

class B
{
public:
    //weak_ptr<A> pa;
    shared_ptr<A> pa;
    ~B()
    {
        cout <<"kill B\n";
    }
};

int main(int argc, char** argv)
{
    shared_ptr<A> sa(new A());
    shared_ptr<B> sb(new B());
    if(sa && sb)
    {
        sa->pb=sb;
        sb->pa=sa;
    }
    cout<<"sa use count:"<<sa.use_count()<<endl;
    return 0;
}

```

## 循环引用

现在有两个类A和B需要定义，定义A的时候需要用到B，定义B的时候需要用到A。

A和B的定义和调用都放在一个文件中肯定是不可以的，这样就会造成两个循环调用的死循环。根本原因是：定义A的时候，A的里面有B，所以需要去查看B的占空间大小，但是查看的时候又发现需要知道A的占空间大小，造成死循环。

解决方法:

- 写两个头文件A.h和B.h分别用于声明类A和B;
- 写两个.cpp文件分别用于定义类A和B;
- 在A的头文件中导入B的头文件;
- 在B的头文件中不导入A的头文件, 但是用`extern`的方式声明类A, 并且, 在B中使用A的时候要用指针的形式。

原理: 在B中用指针调用A, 那么在A需要知道B占空间大小的时候, 就会去找到B的定义文件, 虽然B的定义文件中并没有导入A的头文件, 不知道A的占空间大小, 但是由于在B中调用A的时候用的指针形式, B只知道指针占4个字节就可以, 不需要知道A真正占空间大小, 也就是说, A也是知道B的占空间大小的。

## 强制类型转换运算符

- `static_cast`: 用于非多态类型的转换
- `dynamic_cast`: 用于多态类型转换

使用情形:

```
class A
{
public:
    virtual void print()
    {
        std::cout << "Welcome to WorldA!" << std::endl;
    }

    void test()
    {
        cout << "Test in A" << endl;
    }
};

class B : public A
{
public:
    B() :a(0), b(0) {}
    ~B() {}
    virtual void print()
    {
        std::cout << "Welcome to WorldB!" << std::endl;
    }
    void test()
    {
        cout << "Test in B" << endl;
    }
private:
    double a, b;
};
```

之后在程序中执行：

```
A* pa = new B();
pa->print(); // 输出: Welcome to WorldB!
pa->test(); // 输出: Test in A
```

原因是：

虚函数调用是运行时类型判断，根据指针实际指向的对象类型，因此调用了B::print()

而非虚函数调用是编译期类型判断，根据指针类型，因此调用A::test()

现在，如果我们希望通过指针pa调用B::test,该怎么做呢？

```
dynamic_cast<B*>(pa)->print(); // 输出: Welcome to WorldB!
dynamic_cast<B*>(pa)->test(); // 输出: Test in B
```

即dynamic\_cast执行安全的向下转换:dynamic\_cast<new\_type> (expression),在执行downcast时，如果expression实际指向的类型与new\_type一致，转换才能成功，否则转换失败。

## 重写cout类

今天被同学问了这样一道题，需要实现一个自定义标准输出对象，mycout。要求：

- 对常用类型，如int、char、double等支持"<<"操作符
- 支持操作符"<<"的连续调用，如mycout<<a<<b;
- 支持复数类ComplexNumber的输出，形式为3+4i
- 支持输出换行符endl

上网搜了一下，主要是endl支持比较复杂，因为std::endl本质上是一个函数，前三个条件很好满足。代码如下：

```
#include <iostream>

// 复数类
class ComplexNumber
{
public:
    ComplexNumber(int real,int imaginary)
        :m_real(real),m_imaginary(imaginary){}
    int getReal()const {
        return m_real;
    }
    int getImaginary()const {
        return m_imaginary;
    }
}
```

```
private:
    int m_real;
    int m_imaginary;
};

struct MyCout
{
    // 普通输出
    template <typename T>
    MyCout& operator<<(const T& x)
    {
        std::cout << x;

        return *this;
    }

    // 输出复数
    MyCout& operator<<(ComplexNumber const & x)
    {
        std::cout<<x.getReal()<<'+'<< x.getImaginary() << 'i';

        return *this;
    }

    // 定义自己的endl
    // function that takes a custom stream, and returns it
    typedef MyCout& (*MyCoutManipulator)(MyCout&);

    // take in a function with the custom signature
    MyCout& operator<<(MyCoutManipulator manip)
    {
        // call the function, and return it's value
        return manip(*this);
    }

    // define the custom endl for this stream.
    // note how it matches the `MyStreamManipulator`
    // function signature
    static MyCout& endl(MyCout& mycout)
    {
        // print a new line
        std::cout << std::endl;

        // do other stuff with the stream
        // std::cout, for example, will flush the stream
        mycout << "Called MyStream::endl!" << std::endl;

        return mycout;
    }
}
```

```

// 输出std::endl
// this is the type of std::cout
typedef std::basic_ostream<char, std::char_traits<char> > CoutType;

// this is the function signature of std::endl
typedef CoutType& (*StandardEndLine)(CoutType&);

// define an operator<< to take in std::endl
MyCout& operator<<(StandardEndLine manip)
{
    // call the function, but we cannot return it's value
    manip(std::cout);

    return *this;
}
};

int main(void)
{
    MyCout mycout;
    mycout << 24<<'>'<<23<<std::endl;
    mycout << "This is a string!";
    mycout << std::endl;
    ComplexNumber n(10,5);
    mycout<< n;
    return 0;
}

```

上面的代码中，仅需要重载实现第一个参数为x的函数，就可以实现对"<<"的支持，而且还支持连续输出。

## 友元函数

类的友元函数是定义在类外部（因为友元函数是类外的函数，所以它的声明可以放在类的私有段或公有段且没有区别。），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。

友元可以是一个函数，该函数被称为友元函数；友元也可以是一个类，该类被称为友元类，在这种情况下，整个类及其所有成员都是友元。

```

#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;
}
```

- 优点:

在实现类之间数据共享时, 减少系统开销, 提高效率。如果类A中的函数要访问类B中的成员(例如: 智能指针类的实现), 那么类A中该函数要是类B的友元函数。具体来说: 为了使其他类的成员函数直接访问该类的私有变量。即: 允许外面的类或函数去访问类的私有变量和保护变量, 从而使两个类共享同一函数。

- 缺点: 友元函数破坏了封装机制, 尽量不使用成员函数, 除非不得已的情况下才使用友元函数。

## Lambda表达式

lambda表达式是C++11中引入的一项新技术, 利用lambda表达式可以编写内嵌的匿名函数, 用以替换独立函数或者函数对象, 并且使代码更可读。但是从本质上来讲, lambda表达式只是一种语法糖, 因为所有其能完成的工作都可以用其它稍微复杂的代码来实现。但是它简便的语法却给C++带来了深远的影响。如果从广义上说, lambda表达式产生的是函数对象。在类中, 可以重载函数调用运算符(), 此时类的对象可以将具有类似函数的行为, 我们称这些对象为函数对象(Function Object)或者仿函数(Functor)。相比lambda表达式, 函数对象有自己独特的优势。下面我们开始具体讲解这两项黑科技。

### 一个例子

对于代码:

```
int t = 2;
auto f = [t](int a, float b)->float {return t*a*b; };
```



```
t = 3; // 这时修改t不会影响f
std::cout << f(2, 3); // 输出12
```

编译时会生成一个新类，并重载调用运算符()：

```
class A{
public:
    A(int t):t(t){}; //该形参对应捕获的变量
    //该调用运算符的返回类型、形参和函数体都与lambda一致
    float operator()(int a,float b) const
    {
        return t*a*b;
    }
private:
    int t;
};
```

目前认为lambda表达式就是使代码更加简洁并易于维护。

## C++中构造函数能否调用虚函数

结论：不要在类的构造或者析构过程中调用虚函数，因为这样的调用永远不会沿类继承树往下传递到子类中去。

原因：

- 构造函数：我们在构造一个派生类时会首先执行基类的构造函数，来构造其基类的部分，如果基类构造函数中定义了虚函数，则它对应的派生类函数此时并没有被构造，所以就会执行自己的虚函数，从而失去了多态的意义。
- 析构函数：析构函数也一样，在析构一个派生类对象时，先会调用其派生类的析构函数，然后调用其基类的析构函数，所以在基类调用析构函数时，派生类已经被清除了，所以调用依然是基类自己的析构函数。

看一下代码：

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

class A
{
public:
    virtual void show(){
        cout<<"in A"<<endl;
    }
    A()
    {
        show();
    }
    virtual ~A(){show();}
```

```
};

class B:public A
{
public:
    void show(){
        cout<<"in B"<<endl;
    }
    B()
    {
        show();
    }
    virtual ~B(){show();}
};

int main()
{
    A a;
    cout<<"---"<<endl;
    B b;
    cout<<"---"<<endl;
    return 0;
}

/*
输出:

in A
---
in A
in B
---
in B
in A
in A

注: 析构B时输出两行, in B和in A。

*/
```

解释: 构造类B时, 先调用A的构造函数, 然后再调用B的构造函数。此时即使A的构造函数中存在虚函数, 也会像不是虚函数一样被执行。同理, 析构B时, A中析构函数调用的虚函数也依然是自己的虚函数。

## 其他知识点

- c++中赋值操作的返回值是值本身, 所以支持连续赋值, 如`a=b=10`。
- `int *f()`: 这个函数的返回值是一个指向int类型的指针;
- `int(*f)()`: 这是一个函数的指针.它要指向一个函数才能有用.指向一个函数之后可以用它来代替该函数.之后使用这个指针相当于使用该函数.

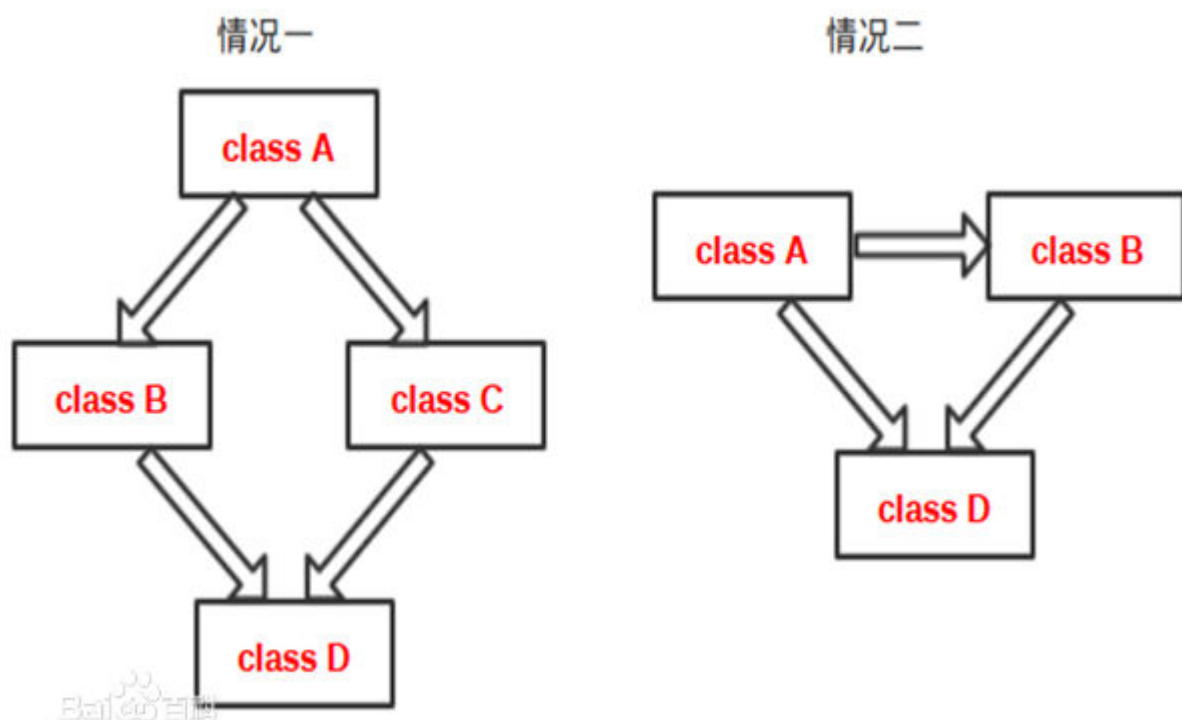
- 下面代码的输出结果：

```
int a=10, *b = &a, **c = &b, ***d=&c;
cout<<a<<endl<<b<<endl<<c<<endl<<d<<endl;
/*
输出：
10
0x69fed8
0x69fed4
0x69fed0
*/
```

只要你想可以一直\*下去。

- 如果短整型的长度是2字节，有一个短整型指针p的值是0xFFED1200，那么p+1的值为(0xFFED1202)
- 设置虚基类的目的是（2）
  1. 简化程序
  2. 消除二义性
  3. 提高运行效率
  4. 减少目标代码

多重继承图示：（以左边为例）



多重继承定义：一个派生类（D）有2个或2个以上的基类（B和C）；

多重继承引起的二义性：假如上述这些基类（B和C）具有相同的基类A，A中的成员数据和成员函数，最终都会以双份的形式拷贝到类D中，那么调用的时候就会出现二义性问题。

虚基类：专门用来解决多重继承引起的二义性问题；（可以理解为D直接从A继承）

- 观察代码运行结果

```
#define MAX(a,b) a>b?a:b

int a = 2, b =3;
int c = 2, d =3;
cout<< (MAX(MAX(a++,b),MAX(a++, b)))<<endl;
cout<<a<<endl;

// 输出 5 6
cout<<max(max(c++,d),max(c++,d))<<endl;
cout<<c<<endl;

// 输出 3 4
```

对于define，要从最外层开始替换，替换完之后再执行

- 观察代码运行结果

```
int a[10];
int *b = a;
cout<<sizeof(a)<<endl;
cout<<sizeof(b)<<endl;

// 输出40 4
```

- string类型的length()

下面代码执行的结果为：

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string s = "abc\0def";
    cout<<s.size()<<endl<<s.length()<<endl;
    cout<<sizeof(s)<<endl<<s[5]<<endl;
    return 0;
}
```

不论是`size()`还是`length()`，程序都输出3，`sizeof()`输出4，`s[5]`为不确定的任意值。

- 观察下面代码的运行结果

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main()
{
    char p1[] = "Hello";
    char *p2 = "Hi";
    char *p3 = (char*)malloc(100);
    char p4[10][10];

    cout<<sizeof(p1)<<endl;
    cout<<sizeof(p2)<<endl;
    cout<<sizeof(p3)<<endl;
    cout<<sizeof(p4)<<endl;

    // 依次输出 6 4 4 100

    return 0;
}
```

- 观察下面代码

```
int main()
{
    char p1[] = "Hello World!";
    char p2[] = "Hello World!";

    //在栈上创建两个数组，分别存放字符串，首地址不一样
    if(p1 == p2) {

    } else {

    }

    char *p3 = "Hello World!";
    char *p4 = "Hello World!";

    //在栈上创建两个指针，分别指向存储在静态数据区的字符串，地址一样
    if(p3 == p4) {

    } else {

    }

}
```

结果显示，p1和p2不相等，p3和p4相等。

《c和指针》上有一段：指针和数组并不是相等的，当声明一个数组时，它同时也分配了一段内存空间，用于存储数组元素，但当声明一个指针时，它只分配了用于容纳指针的空间（32位中4个字节的大小）。

p1、p2、p3和p4都具有指针值，都可以进行间接访问（解引用）和下标引用操作。但是它们还是存在区别和联系：

1. `char p[] = "Hello World"` 的意义是在栈上创建数组，数组直接存放Hello World字符串，数组名p代表这段内存的首地址，是一个常量，`p++`操作是错误的！而`char * p = "Hello World"` 的意义是在栈上创建指针，指向存储在静态数据区的字符串，指针p是一个指针变量，`p++`操作是正确的。
2. 当一个数组名作为函数参数时，数组名的值就是指向数组第一个元素的指针，所以此时传递给函数的是指针的拷贝。指针作为形参和数组名作为形参的效果是一样的

## 数据结构

### 数组、链表、Vector的区别

- 存储方面：数组和vector的底层实现都是数组，即静态分配内存，在内存中是一整块连续的空间，而链表则是动态分配内存，内存中不连续。数组元素在栈区，链表元素在堆区。另外三者都是无序且可以重复的。vector在程序员角度上是可变长的，但本质上还是定长的，如果一开始分配的空间不够的话，会重新重新分配一个更大的空间，然后进行元素拷贝。
- 读取、增加、删除略，注意数组和vector需要移动元素，复杂度为 $O(n)$ 即可。
- STL中的list和链表不一样，list的底层实现是双向链表

### 一些STL的底层实现

stl容器包含顺序容器和关联容器。关联容器主要有vector，list，deque，关联容器主要是pair、set、map、multiset和multimap，所以总共算是7种。

- vector为数组，list是双向链表
- deque为双端队列
- stack或queue都是双端队列(deque)实现（有文章也说用list实现，目前还不确定），不用vector可能是因为扩容耗时
- priority\_queue为vector+max\_heap.
- set和map都是红黑树，红黑树的增删改查复杂度都是 $O(\log n)$ 。
- 前面带hash的STL，比如hash\_map（可以认为就是unordered\_map）和hash\_set都是哈希表，增删改查复杂度都是 $O(1)$ ，但最坏情况为 $O(n)$ 。哈希表涉及桶的概念，类似于桶排序，另外哈希表占用的内存会比红黑树大一些，因此常用于空间换时间或查询频繁的情形。
- 还需要注意一点，对于.size()的操作，不同的STL时间复杂度也不同，对于vector、deque、vector（如果使用deque实现）、queue（如果使用deque实现），他们的size()都是用直接用尾部元素下标减去首部元素下标，所以是 $O(1)$ 的复杂度。而set和map底层都是红黑树，size()操作也是 $O(1)$ 级别，原因是额外存储了一个size变量还是底层用数组实现还不清楚。
- 而list()的size()根据编译器不同，实现也不同，有的直接计算首节点到尾结点的距离，复杂度为 $O(n)$ ，有的则记录一个额外的变量，但在进行链表剪切(splince())时，即将一个长链表从中间断开，形成两个新链表时，size()的 $O(1)$ 特性会丢失。

## 支持随机访问迭代器的容器

随机访问迭代器是可以随机访问容器中的元素的双向迭代器。这种迭代器在Vector, Dequeue, String, 和数组类型Array容器上使用。在双向迭代器上定义的操作也适用于随机访问迭代器。而list不支持随机访问迭代器。

## 解决哈希冲突的方法

- 开放定址法：从发生冲突的那个单元起，按照一定的次序，从哈希表中找到一个空闲的单元。然后把发生冲突的元素存入到该单元的一种方法。开放定址法需要的表长度要大于等于所需要存放的元素。
  - 线性探查法：线性探查法是开放定址法中最简单的冲突处理方法，它从发生冲突的单元起，依次判断下一个单元是否为空，当达到最后一个单元时，再从表首依次判断。直到碰到空闲的单元或者探查完全部单元为止。
  - 平方探查法：平方探查法即是发生冲突时，用发生冲突的单元 $d[i]$ , 加上  $1^2$ 、 $2^2$ 等。即 $d[i] + 1^2$ ,  $d[i] + 2^2$ ,  $d[i] + 3^2$ ...直到找到空闲单元。
  - 双散列函数探查法：对于线性探测法，当聚焦问题严重或者表接近满时，要搜索一个关键字，往往要逐个检查很多个无关项。双重散列算法的基本策略和线性探测法一项，唯一不同是：它不是检查冲突位置后的每一个位置，而是采用另一个散列函数产生一个固定的增量。（跳跃式检查和插入，减小聚焦大小）。这种方法使用两个散列函数 $h1$ 和 $h2$ 。其中 $h1$ 和前面的 $h$ 一样，以关键字为自变量，产生一个0至 $m-1$ 之间的数作为散列地址； $h2$ 也以关键字为自变量，产生一个1至 $m-1$ 之间的、和 $m$ 互素的数(即 $m$ 不能被该数整除， $m$ 为哈希表表长)作为探查序列的地址增量(即步长)
- 链地址法（拉链法）：链接地址法的思路是将哈希值相同的元素构成一个同义词的单链表，并将单链表的头指针存放在哈希表的第 $i$ 个单元中，查找、插入和删除主要在同义词链表中进行。链表法适用于经常进行插入和删除的情况。

如下一组数字,(32、40、36、53、16、46、71、27、42、24、49、64)哈希表长度为13，哈希函数为 $H(key)=key\%13$ ,则链表法结果如下：

```

0
1  -> 40 -> 27 -> 53
2
3  -> 16 -> 42
4
5
6  -> 32 -> 71
7  -> 46
8
9
10 -> 36 -> 49
11 -> 24
12 -> 64

```

- 再哈希法:就是同时构造多个不同的哈希函数：  $H_i = RH_i(key)$   $i = 1, 2, 3 \dots k$ ; 当 $H1 = RH1(key)$  发生冲突时，再用 $H2 = RH2(key)$  进行计算，直到冲突不再产生，这种方法不易产生聚集，但是增加了计算时间。
- 建立公共溢出区：将哈希表分为公共表和溢出表，当溢出发生时，将所有溢出数据统一放到溢出区。

## 几种树的概念

- 满二叉树：每层节点都排满的二叉树
- 完全二叉树：满二叉树在最后一层从右向左去掉一些节点组成的树（所有节点与满二叉树的前n个节点对应）
- 二叉堆：分为最大堆和最小堆，要求每个父节点比其子节点都要大，或都要小。
- 二叉搜索树（BST），也叫二叉排序树、二叉查找树，要求左子树上所有节点都比根节点小，右子树上所有节点都比根节点大。本质上借鉴了二分的思想，但存在退化的缺点。
- 平衡二叉树，即AVL树，本质上还是一颗二叉搜索树，但解决了退化的问题，里面的二叉树左右子树高度大致相等，具有左旋、右旋、左右双旋和右左双旋的操作。
- 红黑树，同样是一种二叉搜索树，具有变色和左旋、右旋的操作，具有以下五条性质：
  - 节点是红色或黑色
  - 根节点是黑色
  - 每个叶子节点都是黑色的空节点，即插入一个节点时，它本身是红色的，且它要附带两个空的黑色的子节点
  - 每个红色节点的两个子节点都是黑色的
  - 从任意节点到其每个叶子节点的所有路径都包含相同的黑色节点
- B树、B+树、B\*树：同样是平衡的，但叶子节点可以是M个， $M > 2$ ，属于多叉树又名平衡多路查找树（查找路径不只两个），数据库索引技术和磁盘读写里大量使用者B树和B+树的数据结构，这里不做详细介绍。

## 前置++ 和后置++ 的区别

# 设计模式

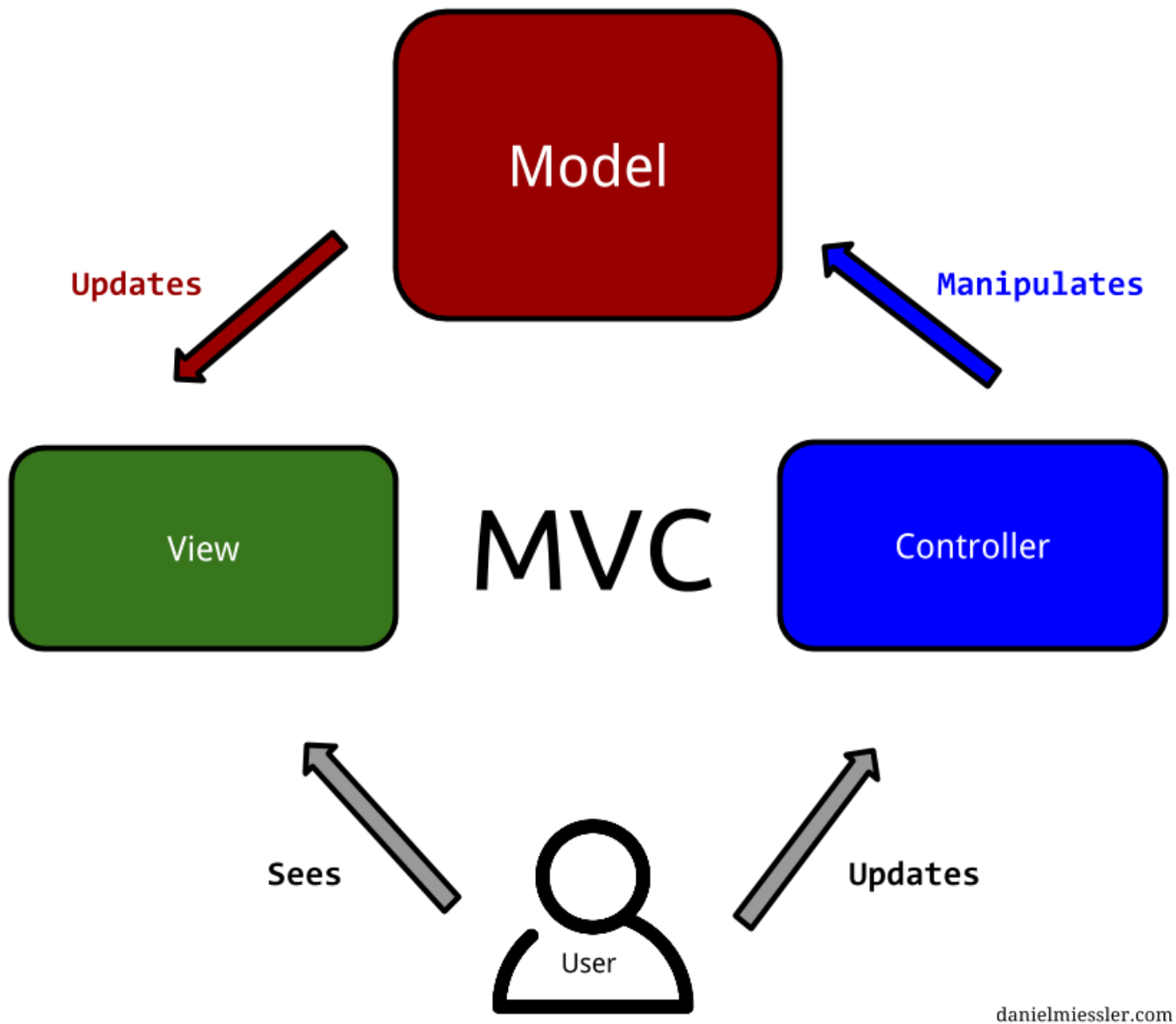
---

## MVC

MVC 模式代表 Model-View-Controller（模型-视图-控制器）模式。这种模式用于应用程序的分层开发。

- **Model（模型）**：模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
- **View（视图）**：视图代表模型包含的数据的可视化。
- **Controller（控制器）**：控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

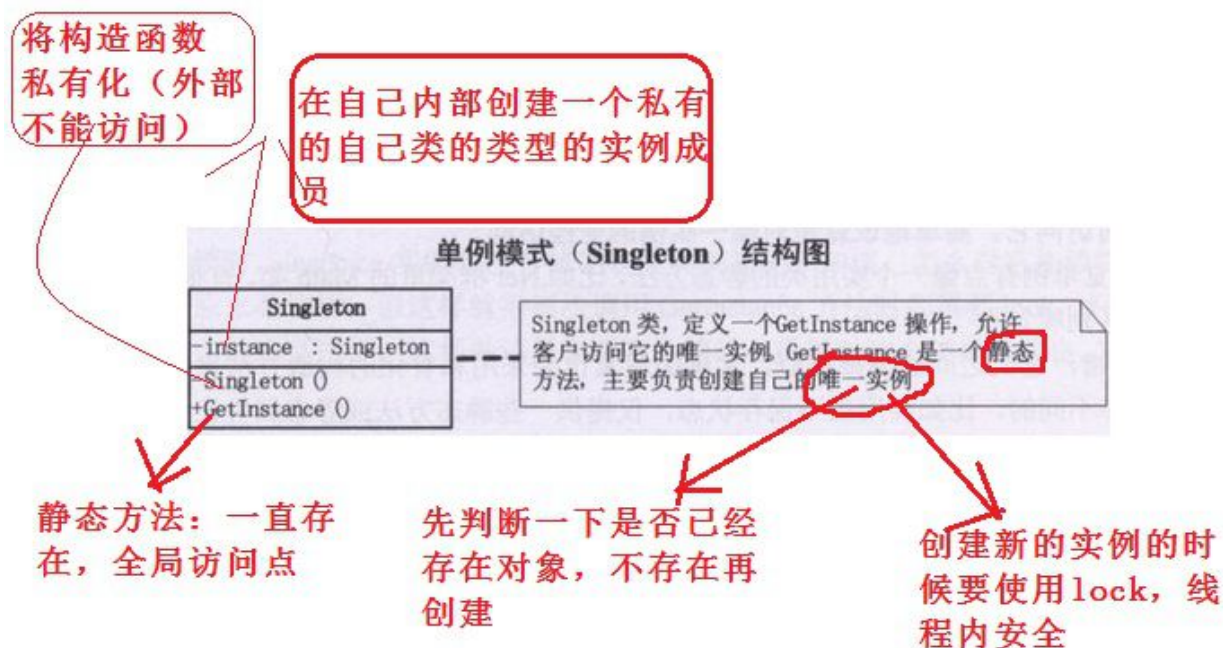




示例：

- StudentModel: Student对象，存放student中的姓名、年龄等数据。
- StudentView: 是一个把学生详细信息输出到控制台的视图类
- StudentController: 是负责存储数据到 Student 对象中的控制器类，并相应地更新视图StudentView。

单例模式



单例模式，也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。

实现单例模式的思路是：一个类能返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用 `getInstance` 这个名称）；当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用；同时我们还将该类的构造函数定义为私有方法，这样其他处的代码就无法通过调用该类的构造函数来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。

单例模式在多线程的应用场合下必须小心使用。如果当唯一实例尚未创建时，有两个线程同时调用创建方法，那么它们同时没有检测到唯一实例的存在，从而同时各自创建了一个实例，这样就有两个实例被构造出来，从而违反了单例模式中实例唯一的原则。解决这个问题的办法是为指示类是否已经实例化的变量提供一个互斥锁（虽然这样会降低效率）。

单例模式的作用在于保证整个程序在一次运行的过程中，被单例模式声明的类的对象要有且只有一个。（注意，不是说这个类只有孤立的一个，没有继承，而是说这个类的实例对象只能有一个）

单例模式的从实现步骤上来讲，分为三步

- 构造方法私有，保证无法从外部通过 `new` 的方式创建对象。
- 对外提供获取该类实例的静态方法
- 类的内部创建该类的对象，通过第 2 步的静态方法返回

```
public class BasicSingleton {
```

```
//创建唯一实例
```

```
private static final BasicSingleton instance = new BasicSingleton();

//第二步暴露静态方法返回唯一实例
public static BasicSingleton getInstance() {
    return instance;
}

//第一步构造方法私有
private BasicSingleton() {
}
}
```

## 其他知识点

- (2) 设计模式将抽象部分与它的实现部分相分离。
  1. Singleton (单例)
  2. Bridge (桥接)
  3. Composite (组合)
  4. Facade (外观)

## 操作系统

---

### 操作系统四个基本特征

#### 并发和并行

- 并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。
- 并行需要硬件支持，如多流水线、多核处理器或者分布式计算系统。
- 操作系统通过引入进程和线程，使得程序能够并发运行。

#### 共享

- 共享是指系统中的资源可以被多个并发进程共同使用。
- 有两种共享方式：互斥共享和同时共享。
- 互斥共享的资源称为临界资源，例如打印机等，在同一时刻只允许一个进程访问，需要用同步机制来实现互斥访问。

#### 虚拟

- 虚拟技术把一个物理实体转换为多个逻辑实体。
- 主要有两种虚拟技术：时（时间）分复用技术和空（空间）分复用技术。
- 多个进程能在同一个处理器上并发执行使用了时分复用技术，让每个进程轮流占用处理器，每次只执行一小段时间片并快速切换。

- 虚拟内存使用了空分复用技术，它将物理内存抽象为地址空间，每个进程都有各自的地址空间。地址空间的页被映射到物理内存，地址空间的页并不需要全部在物理内存中，当使用到一个没有在物理内存的页时，执行页面置换算法，将该页置换到内存中。

## 异步

- 异步指进程不是一次性执行完毕，而是走走停停，以不可知的速度向前推进。

## 操作系统的四大功能

进程管理、内存管理、文件管理、设备管理

## 进程和线程

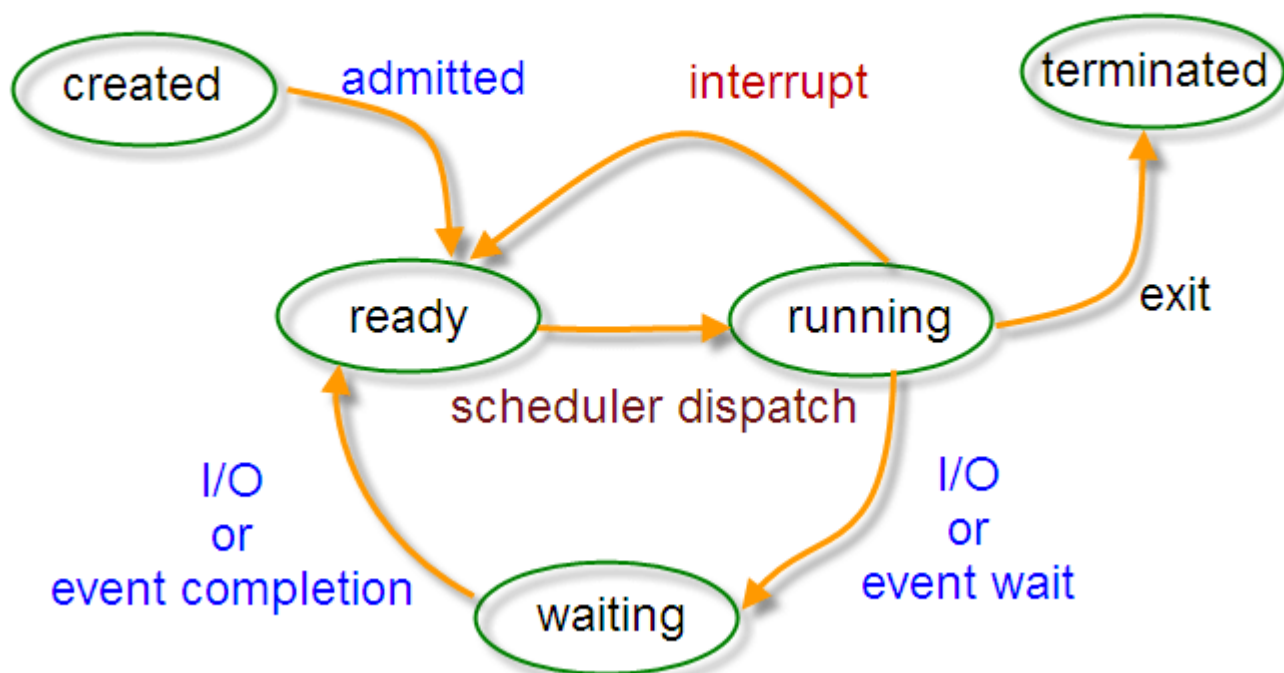
- 进程
  - 进程是资源分配的基本单位。
  - 进程控制块 (Process Control Block, PCB) 描述进程的基本信息和运行状态，所谓的创建进程和撤销进程，都是指对 PCB 的操作。
  - 进程可以通过时分复用来并发执行
- 线程
  - 线程是独立调度的基本单位。
  - 一个进程中可以有多个线程，它们共享进程资源。
  - QQ 和浏览器是两个进程，浏览器进程里面有很多线程，例如 HTTP 请求线程、事件响应线程、渲染线程等等，线程的并发执行使得在浏览器中点击一个新链接从而发起 HTTP 请求时，浏览器还可以响应用户的其它事件。
- 区别
  - 拥有资源：进程是资源分配的基本单位，线程不拥有资源，线程只可以访问隶属进程的资源。
  - 调度：线程是独立调度的基本单位，在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。如果是多核CPU，则同一进程中的不同线程可以实现真正的并行执行。（还有一种情况，即一台计算机中有多个CPU，这样就可以实现多进程的并行，这里只需要记住一点即可，就是**CPU只能看到线程**，每个CPU看到的是一个进程中的不同线程，而不是多个进程）
  - 系统开销：进程大，线程小。由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。
  - 通信：线程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助IPC（Inter-Process Communication）机制。

## 进程的状态切换

- 创建(Created)状态：进程在创建时需要申请一个空白PCB，向其中填写控制和管理进程的信息，完成资源分配。如果创建工作无法完成，比如资源无法满足，就无法被调度运行，把此时进程所处状态称为创建状态
- 就绪(Ready)状态：进程已经准备好，已分配到所需资源，只要分配到CPU就能够立即运行
- 执行(Running)状态：进程处于就绪状态被调度后，进程进入执行状态

- 阻塞(Waiting)状态：正在执行的进程由于某些事件（I/O请求，申请缓存区失败）而暂时无法运行，进程受到阻塞。在满足请求时进入就绪状态等待系统调用
- 终止(Terminated)状态：进程结束，或出现错误，或被系统终止，进入终止状态。无法再执行
- 状态转换：只有running和ready状态是可以相互转换的，发生在时间片用完和重新分配。然后IO阻塞会导致程序从running转到waiting，然后IO执行完成后，回到ready状态。

## Process State



## 进程同步

- 临界区、同步与互斥、信号量、管程

## 线程同步

- 临界区、同步与互斥、信号量、事件

## 进程通信

- 进程同步与进程通信很容易混淆，它们的区别在于：
  - 进程同步：控制多个进程按一定顺序执行；
  - 进程通信：进程间传输信息。
  - 进程通信是一种手段，而进程同步是一种目的。也可以说，为了能够达到进程同步的目的，需要让进程进行通信，传输一些进程同步所需要的信息。
- 方法：
  - 管道：只支持半双工通信（只允许交替单向传输，而不是双向同时传输。另外单工指只能单向传输，全双工指可以同时传输）；只能在父子进程或者兄弟进程中使用。

- FIFO：也称为命名管道，去除了管道只能在父子进程中使用的限制。
- 消息队列：相比于 FIFO，消息队列具有以下优点：
  - 消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；
  - 避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；
  - 读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。
- 信号量：它是一个计数器，用于为多个进程提供对共享数据对象的访问。
- 共享存储：允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种IPC。需要使用信号量用来同步对共享存储的访问。多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用内存的匿名段。
- 套接字Socket：与其它通信机制不同的是，它可用于不同机器间的进程通信。

## 死锁

- 定义：是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。
- 四个必要条件：
  1. 互斥：一个资源每次只能被一个进程使用。
  2. 占有并等待：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
  3. 不可抢占：进程已获得的资源，在未使用完之前，不能强行剥夺。
  4. 循环等待：若干进程之间形成一种头尾相接的循环等待资源关系（即有一系列进程试图请求相同的资源，并且每个进程都占有一部分但不是全部资源）
- 解决死锁的基本方法：
  1. 死锁预防：通过设置某些限制条件，去破坏死锁的四个条件中的一个或几个条件，来预防发生死锁。但由于所施加的限制条件往往太严格，因而导致系统资源利用率和系统吞吐量降低。
  2. 死锁避免：允许前三个必要条件，但通过明智的选择，确保永远不会到达死锁点，因此死锁避免比死锁预防允许更多的并发。
  3. 死锁检测：无需采取任何限制性措施，而是允许系统在运行过程发生死锁，但可通过系统设置的检测机构及时检测出死锁的发生，并精确地确定于死锁相关的进程和资源，然后采取适当的措施，从系统中将已发生的死锁清除掉。
  4. 死锁解除：与死锁检测相配套的一种措施。当检测到系统中已发生死锁，需将进程从死锁状态中解脱出来。常用方法：撤销或挂起一些进程，以便回收一些资源，再将资源分配给已处于阻塞状态的进程。死锁检测盒解除有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。
- 死锁预防：
  1. 破坏互斥条件（不可以）：它是设备的固有属性所决定的，不仅不能改变，还应该加以保证；
  2. 破坏占有并等待条件（低效）：为预防占有并等待条件，可以要求进程一次性的请求所有需要的资源，并且阻塞这个进程直到所有请求都同时满足。这个方法比较低效；
  3. 破坏不可抢占条件：允许进程抢占其他程序的资源或者进程在申请资源失败时，要释放已有的资源（跟2不同，2是一次性请求全部资源）；
  4. 破坏循环等待条件（低效）：通过定义资源类型的线性顺序来预防，即如果一个进程已经分配了R类资源，那么接下来请求的资源只能是那些排在R类型之后的资源类型，类似于先修课程。

- 死锁避免：
  - 进程启动拒绝：如果一个进程的请求会导致死锁，则不启动该进程
  - 资源分配拒绝(银行家算法)：如果一个进程增加的资源请求会导致死锁，则不允许此分配。
- 死锁检测和解除：剥夺资源和撤销进程。

## 乐观锁与悲观锁

- 悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。
- 乐观锁：总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。
- 从上面两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反而是降低了性能，所以一般多写的场景下用悲观锁就比较合适。
- 乐观锁一般会使用版本号机制或CAS算法实现。

## 内存管理

- 为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。这些页被映射到物理内存，但不需要映射到连续的物理内存，也不需要所有页都必须在物理内存中。当程序引用到不在物理内存中的页时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令。从上面的描述中可以看出，虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序成为可能。
- 分页系统地址映射 内存管理单元（MMU）管理着地址空间和物理内存的转换，其中的页表（Page table）存储着页（程序地址空间）和页框（物理内存空间）的映射表，使用红黑树实现。
- 一个虚拟地址分成两个部分，一部分存储页面号，一部分存储偏移量。

## 页置换算法

- 最佳（OPT, Optimal replacement algorithm）：所选择的被换出的页面将是最长时间内不再被访问，通常可以保证获得最低的缺页率。是一种理论上的算法，因为无法知道一个页面多长时间不再被访问。
- 最近最久未使用（LRU, Least Recently Used）：虽然无法知道将来要使用的页面情况，但是可以知道过去使用页面的情况。LRU 将最近最久未使用的页面换出。为了实现 LRU，需要在内存中维护一个所有页面的链表。当一个页面被访问时，将这个页面移到链表表头。这样就能保证链表表尾的页面是最近最久未访问的。因为每次访问都需要更新链表，因此这种方式实现的 LRU 代价很高。
- 最近未使用（NRU, Not Recently Used）：每个页面都有两个状态位：R 与 M，当页面被访问时设置页面的 R=1，当页面被修改时设置 M=1。其中 R 位会定时被清零。可以将页面分成以下四类：R=0，

M=0、R=0, M=1、R=1, M=0、R=1, M=1。NRU 优先换出已经被修改的脏页面（R=0, M=1），而不是被频繁使用的干净页面（R=1, M=0）。

- 先进先出（FIFO, First In First Out）：选择换出的页面是最先进入的页面。该算法会将那些经常被访问的页面也被换出，从而使缺页率升高。
- 第二次机会算法：当页面被访问（读或写）时设置该页面的 R 位为 1。需要替换的时候，检查最老页面的 R 位。如果 R 位是 0，那么这个页面既老又没有被使用，可以立刻置换掉；如果是 1，就将 R 位清 0，并把该页面放到链表的尾端，修改它的装入时间使它就像刚装入的一样，然后继续从链表的头部开始搜索。
- 时钟：第二次机会算法需要在链表中移动页面，降低了效率。时钟算法使用环形链表将页面连接起来，再使用一个指针指向最老的页面。

## 进程页表和内核页表

内核页表中的内容为所有进程共享，每个进程都有自己的“进程页表”

操作系统采用分页式存储管理(PAGING)方法，要求(2):

1. 每个进程拥有一张页表，且进程的页表驻留在内存中
2. 每个进程拥有一张页表，但只要执行进程的页表驻留在内存中，其他进程的页表不必驻留在内存中
3. 所有进程共享一张页表，以节约有限的内存空间，但页表必须驻留在内存中
4. 所有进程共享一张页表，只有页表中当前使用的页面必须驻留在内存中，以最大限度地节约有限的内存空间

## 分页和分段的比较

- 对程序员的透明性：分页透明，但是分段需要程序员显式划分每个段。
- 地址空间的维度：分页是一维地址空间，分段是二维的。
- 大小是否可以改变：页的大小不可变，段的大小可以动态改变。
- 出现的原因：分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护。

# 计算机网络

---

## OSI, TCP/IP, 五层协议的体系结构, 以及各层协议

- **OSI 分层（7层）**：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。
- **TCP/IP 分层（4层）**：网络接口层、网际层、运输层、应用层。
- **五层协议（5层）**：物理层、数据链路层、网络层、运输层、应用层。
- 每一层的协议如下：
  - 物理层：RJ45、CLOCK、IEEE802.3（中继器，集线器）
  - 数据链路层：PPP、FR、HDLC、VLAN、MAC（网桥，交换机）
  - 网络层：IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP、（路由器）
  - 传输层：TCP、UDP、SPX
  - 会话层：NFS、SQL、NETBIOS、RPC
  - 表示层：JPEG、MPEG、ASII



- 应用层：FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS
- 每一层的作用如下：
  - 物理层：通过媒介传输比特,确定机械及电气规范（比特Bit）
  - 数据链路层：将比特组装成帧和点到点的传递（帧Frame）
  - 网络层：负责数据包从源到宿的传递和网际互连（包Packet）
  - 传输层：提供端到端的可靠报文传递和错误恢复（段Segment）
  - 会话层：建立、管理和终止会话（会话协议数据单元SPDU）
  - 表示层：对数据进行翻译、加密和压缩（表示协议数据单元PPDU）
  - 应用层：允许访问OSI环境的手段（应用协议数据单元APDU）

## IP地址的分类

- **A类地址**：以0开头，第一个字节范围：1~126（1.0.0.0 - 126.255.255.255）；
- **B类地址**：以10开头，第一个字节范围：128~191（128.0.0.0 - 191.255.255.255）；
- **C类地址**：以110开头，第一个字节范围：192~223（192.0.0.0 - 223.255.255.255）；
- **D类地址**：以1110开头，第一个字节范围：224~239（224.0.0.0 - 239.255.255.255）；（作为多播使用）
- **E类地址**：保留。

其中A、B、C是基本类，D、E类作为多播和保留使用。

- 以下是留用的内部私有地址：

A类 10.0.0.0--10.255.255.255

B类 172.16.0.0--172.31.255.255

C类 192.168.0.0--192.168.255.255

- IP地址与子网掩码相与得到网络号：

ip : 192.168.2.110

&

Submask : 255.255.255.0

= 网络号 : 192.168.2.0

注:主机号，全为0的是网络号（例如：192.168.2.0），主机号全为1的为广播地址（192.168.2.255）

- 一种题型：某公司申请到一个C类IP网段，但要连接6个子公司，最大的一个子公司有26台主机，请问划分分子网后应用什么子网掩码？

答：需划分6个子网，故C类IP地址的默认子网掩码需从主机位借3位作为1来补充子网地址( $2^3 > 6$ )，题中IP地址为C类，所以掩码为11111111.11111111.11111111.11100000，即255.255.255.224。而掩码为255.255.255.224时，子网内最大可有30台计算机( $2^5 - \text{网关} - \text{广播}$ )，可以分割出8个子网，满足条件。所以从前数和从后数的思路都是可行的，本题目刚好两者答案相同。

## ARP是地址解析协议，简单语言解释一下工作原理。

1. 首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址之间的对应关系。
2. 当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP数据包，该数据包包括的内容有：源主机 IP地址，源主机MAC地址，目的主机的IP 地址。
3. 当本网络的所有主机收到该ARP数据包时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的IP和MAC地址写入到ARP列表中，如果已经存在，则覆盖，然后将自己的MAC地址写入ARP响应包中，告诉源主机自己是它要找的MAC地址。
4. 源主机收到ARP响应包后。将目的主机的IP和MAC地址写入ARP列表，并利用此信息发送数据。如果源主机一直没有收到ARP响应数据包，表示ARP查询失败。
5. 广播发送ARP请求，单播发送ARP响应。

## 各种协议的介绍

- ICMP协议：因特网控制报文协议。它是TCP/IP协议族的一个子协议，用于在IP主机、路由器之间传递控制消息。
- TFTP协议：是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。
- HTTP协议：超文本传输协议，是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。
- NAT协议：网络地址转换属接入广域网(WAN)技术，是一种将私有（保留）地址转化为合法IP地址的转换技术，
- DHCP协议：动态主机配置协议，是一种让系统得以连接到网络上，并获取所需要的配置参数手段，使用UDP协议工作。具体用途：给内部网络或网络服务供应商自动分配IP地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段。

## 描述RARP协议

RARP是逆地址解析协议，作用是完成硬件地址到IP地址的映射，主要用于无盘工作站，因为给无盘工作站配置的IP地址不能保存。工作流程：在网络中配置一台RARP服务器，里面保存着IP地址和MAC地址的映射关系，当无盘工作站启动后，就封装一个RARP数据包，里面有其MAC地址，然后广播到网络上去，当服务器收到请求包后，就查找对应的MAC地址的IP地址装入响应报文中发回给请求者。因为需要广播请求报文，因此RARP只能用于具有广播能力的网络。

## TCP和UDP

- 用户数据报协议 UDP（User Datagram Protocol）是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。
- 传输控制协议 TCP（Transmission Control Protocol）是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。
- 比较：TCP/UDP
  - 是否连接：面向连接/无连接。

- 传输可靠性：可靠/不可靠。通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP 尽最大努力交付，即不保证可靠交付。TCP 的逻辑通信信道是全双工的可靠信道；UDP 则是不可靠信道
- 长度限制：无/有。TCP 面向字节流（可能出现黏包问题），实际上是 TCP 把数据看成一连串无结构的字节流；UDP 是面向报文的（不会出现黏包问题）
- 应用场合：传输量大/传输量小
- 速度：慢/快
- 支持广播和多播：否/是。每一条 TCP 连接只能是点到点的；UDP 支持一对一，一对多，多对一和多对多的交互通信
- 拥塞控制：有/无。UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 IP 电话，实时视频会议等）
- TCP 首部开销20字节；UDP 的首部开销小，只有 8 个字节

- 头部

- UDP:

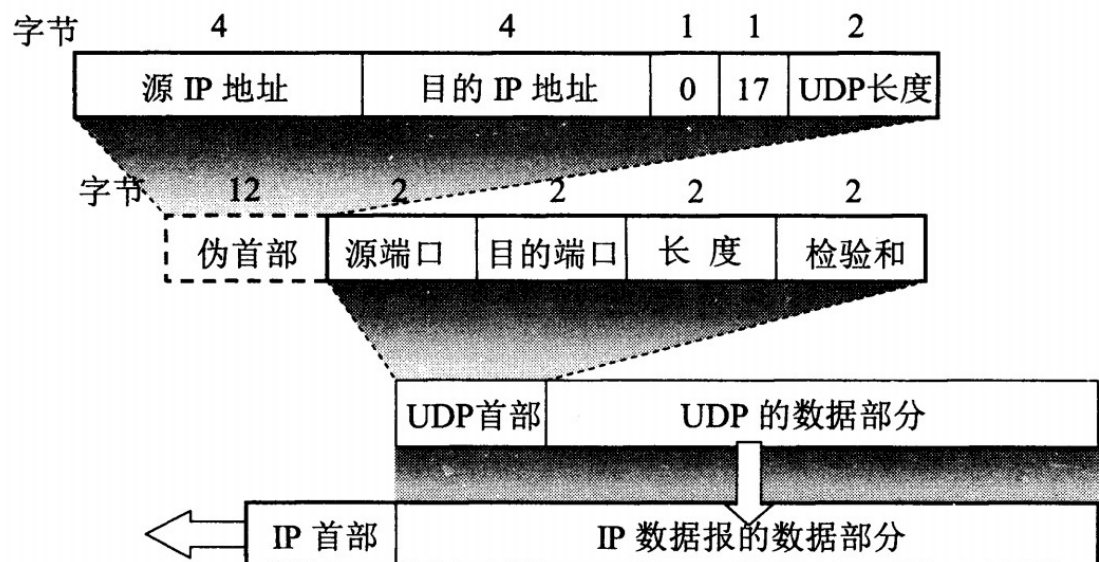


图 5-5 UDP 用户数据报的首部和伪首部

首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。12 字节的伪首部是为了计算检验和临时添加的。

◦ TCP:

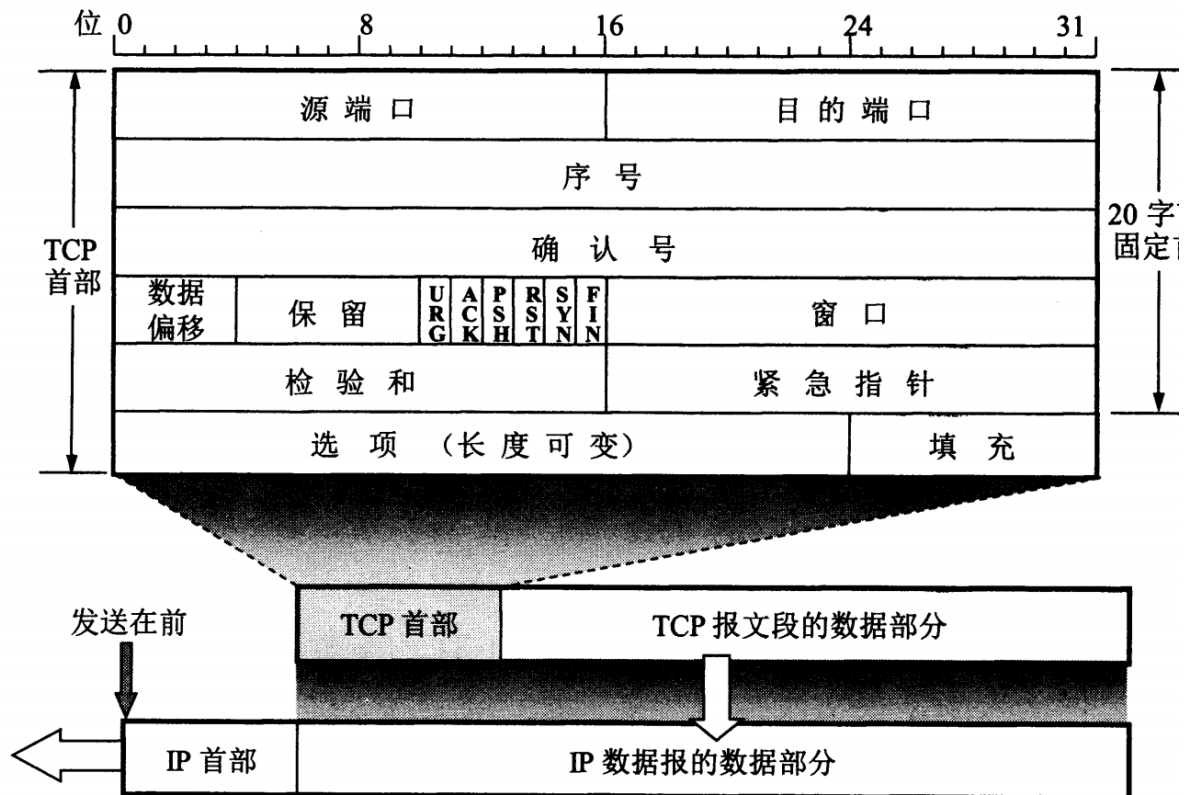


图 5-14 TCP 报文段的首部格式

- **序号：**用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401
- **确认号：**期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。
- **数据偏移：**指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
- **确认 ACK：**当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。
- **同步 SYN：**在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
- **终止 FIN：**用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
- **窗口：**窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的。

## TCP三次握手和四次挥手的全过程（要求可以默写）

- 三次握手：

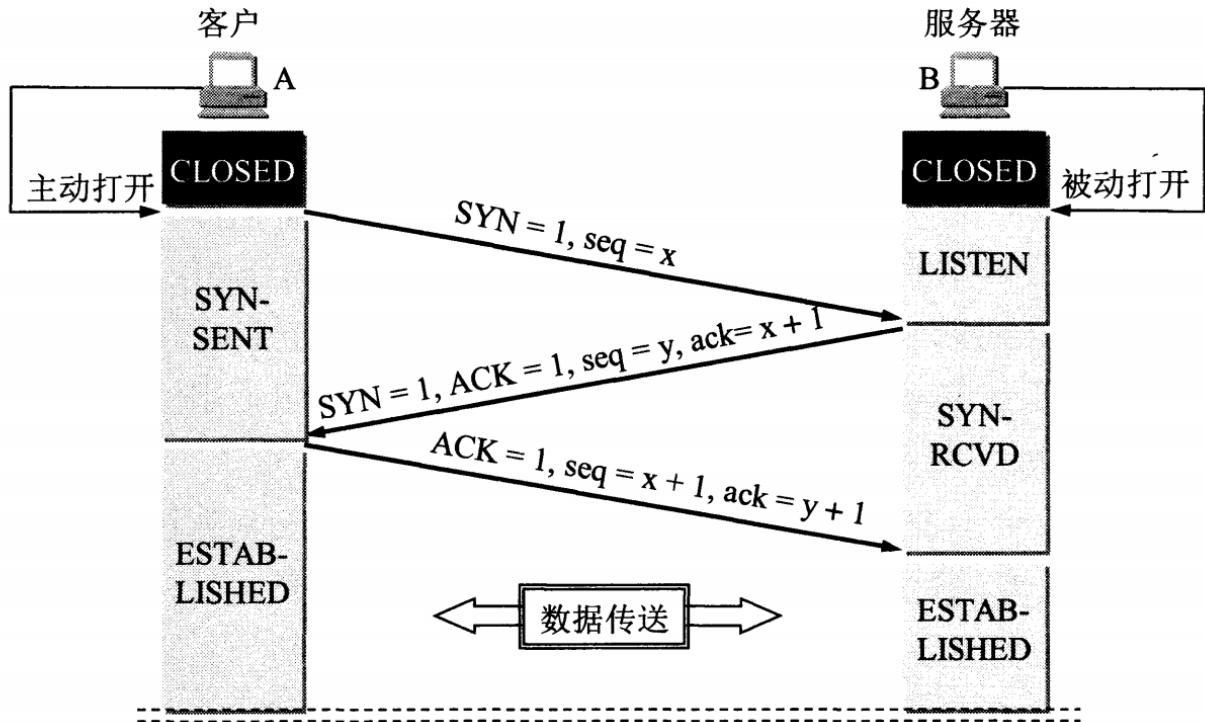


图 5-28 用三报文握手建立 TCP 连接

- 第一次握手：客户端发送syn包( $syn=x$ )到服务器，并进入SYN\_SEND状态，等待服务器确认；
- 第二次握手：服务器收到syn包，必须确认客户的SYN ( $ack=x+1$ )，同时自己也发送一个SYN包 ( $syn=y$ )，即SYN+ACK包，此时服务器进入SYN\_RECV状态；
- 第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK( $ack=y+1$ )，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。“已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”。主要目的防止server端一直等待，浪费资源。

- 四次挥手

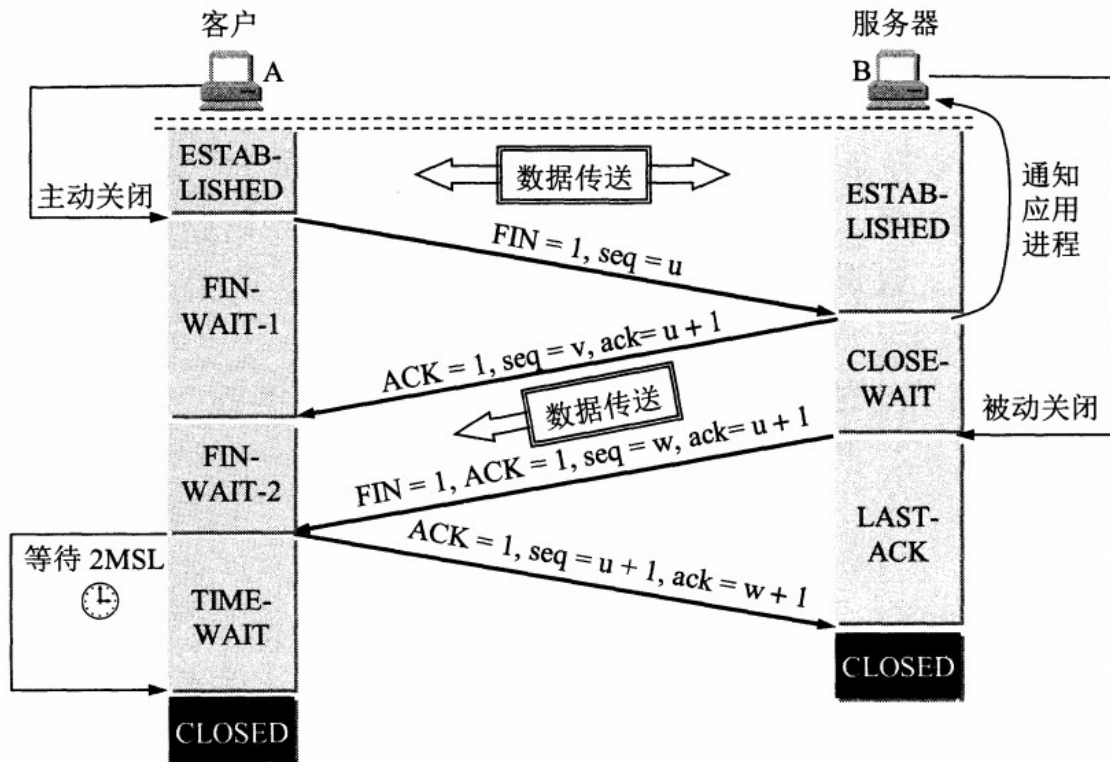


图 5-29 TCP 连接释放的过程

与建立连接的“三次握手”类似，断开一个TCP连接则需要“四次握手”。

- 第一次挥手：主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭方告诉被动关闭方：我已经不会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可以接受数据。
- 第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号）。
- 从被动关闭方到主动关闭方额外的数据传输
- 第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了。
- 第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手。

四次握手的原因：tcp是全双工模式，接收到FIN时意味将没有数据再发来，但是还是可以继续发送数据。

• 细节：

- 每个报文都有seq
- 在tcp连接握手时ACK是和SYN一起发送，断开时ACK却没有和FIN一起发送。原因是因为tcp是全双工模式，接收到FIN时意味将没有数据再发来，但是还是可以继续发送数据。

• 握手的各种状态

- LISTEN: 这个也是非常容易理解的一个状态，表示服务器端的某个SOCKET处于监听状态，可以接受连接了。

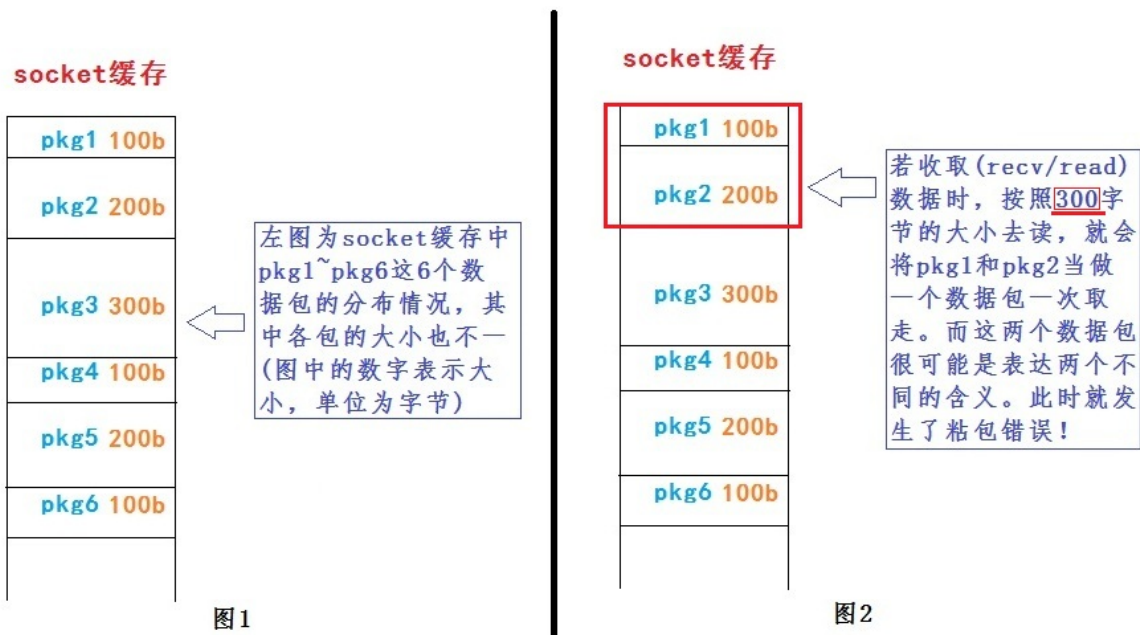
- SYN\_SENT: 当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，因此也随即它会进入到SYN\_SENT状态，并等待服务端的发送三次握手手中的第2个报文。SYN\_SENT状态表示客户端已发送SYN报文。（发送端）
- SYN\_RCVD: 这个状态与SYN\_SENT遥相呼应这个状态表示接受到了SYN报文，在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂，基本上用netstat你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次TCP握手过程中最后一个ACK报文不予发送。因此这种状态时，当收到客户端的ACK报文后，它会进入到ESTABLISHED状态。（服务器端）
- ESTABLISHED: 这个容易理解了，表示连接已经建立了。
- 挥手的各种状态
  - FIN\_WAIT\_1: 这个状态要好好解释一下，其实FIN\_WAIT\_1和FIN\_WAIT\_2状态的真正含义都是表示等待对方的FIN报文。而这两种状态的区别是：FIN\_WAIT\_1状态实际上是当SOCKET在ESTABLISHED状态时，它想主动关闭连接，向对方发送了FIN报文，此时该SOCKET即进入到FIN\_WAIT\_1状态。而当对方回应ACK报文后，则进入到FIN\_WAIT\_2状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应ACK报文，所以FIN\_WAIT\_1状态一般是比较难见到的，而FIN\_WAIT\_2状态还有时常常可以用netstat看到。（主动方）
  - FIN\_WAIT\_2: 上面已经详细解释了这种状态，实际上FIN\_WAIT\_2状态下的SOCKET，表示半连接，也即有一方要求close连接，但另外还告诉对方，我暂时还有点数据需要传送给你(ACK信息)，稍后再关闭连接。（主动方）
  - TIME\_WAIT: 表示收到了对方的FIN报文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果FIN\_WAIT\_1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。（主动方）
  - CLOSING（比较少见）: 这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。
  - CLOSE\_WAIT: 这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方close一个SOCKET后发送FIN报文给自己，你系统毫无疑问地会回应一个ACK报文给对方，此时则进入到CLOSE\_WAIT状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以close这个SOCKET，发送FIN报文给对方，也即关闭连接。所以你在CLOSE\_WAIT状态下，需要完成的事情是等待你去关闭连接。（被动方）
  - LAST\_ACK: 这个状态还是比较好理解的，它是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。（被动方）
  - CLOSED: 表示连接中断。
- 有很多CLOSE\_WAIT的原因：服务器没有继续发FIN包给客户端，即某种情况下对方关闭了socket链接，但是我方忙与读或者写，没有关闭连接。
- 设置TIME\_WAIT的原因：
  - 假设最后一个ACK丢失了，被动关闭一方会重发它的FIN。主动关闭一方必须维持一个有效状态信息（TIMEWAIT状态下维持），以便能够重发ACK。如果主动关闭的socket不维持这种状态而进入到CLOSED状态，那么主动关闭的socket在处于CLOSED状态时，接收到FIN后将会响应一个RST。被动关闭一方接收到RST后会认为出错了。如果TCP协议想要正常完成必要的操作而终止双方的数据

流传输，就必须完全正确的传输四次握手的四个环节，不能有任何的丢失。这就是为什么socket在关闭后，仍然处于TIME\_WAIT状态的第一个原因，因为他要等待以便重发ACK。

- 假设目前连接的通信双方都已经调用了close()，双方同时进入CLOSED的终结状态，而没有走TIME\_WAIT状态。会出现如下问题，现在有一个新的连接被建立起来，使用的IP地址与端口与先前的完全相同，后建立的连接是原先连接的一个完全复用。还假定原先的连接中有数据报残存于网络之中，这样新的连接收到的数据报中有可能是先前连接的数据报。为了防止这一点，TCP不允许新连接复用TIME\_WAIT状态下的socket。处于TIME\_WAIT状态的socket在等待两倍的MSL时间以后（之所以是两倍的MSL，是由于MSL是一个数据报在网络中单向发出到认定丢失的时间，一个数据报有可能在发送途中或是其响应过程中成为残余数据报，确认一个数据报及其响应的丢弃的需要两倍的MSL），将会转变为CLOSED状态。这就意味着，一个成功建立的连接，必然使得先前网络中残余的数据报都丢失了。

## TCP的其他特性

- 可靠传输：TCP 使用超时重传来实现可靠传输：如果一个已经发送的报文段在超时时间内没有收到确认，那么就重传这个报文段。
- 黏包（粘包）：
  - 定义：发送端可以是一K一K地发送数据，而接收端的应用程序可以两K两K地提走数据，当然也有可能一次提走3K或6K数据，或者一次只提走几个字节的数据，也就是说，应用程序所看到的数据是一个整体，或说是一个流（stream），一条消息有多少字节对应用程序是不可见的，因此TCP协议是面向流的协议，这也是容易出现粘包问题的原因。而UDP是面向消息的协议，每个UDP段都是一条消息，应用程序必须以消息为单位提取数据，不能一次提取任意字节的数据，这一点和TCP是很不同的。怎样定义消息呢？可以认为对方一次性write/send的数据为一个消息，需要明白的是当对方send一条信息的时候，无论底层怎样分段分片，TCP协议层会把构成整条消息的数据段排序完成后才呈现在内核缓冲区。例如基于tcp的套接字客户端往服务端上传文件，发送时文件内容是按照一段一段的字节流发送的，在接收方看了，根本不知道该文件的字节流从何处开始，在何处结束。所谓粘包问题主要还是因为接收方不知道消息之间的界限，不知道一次性提取多少字节的数据所造成的。此外，发送方引起的粘包是由TCP协议本身造成的，TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一个TCP段。若连续几次需要send的数据都很少，通常TCP会根据nagel优化算法把这些数据合成一个TCP段后一次发送出去，这样接收方就收到了粘包数据。





◦ 原因:

- 发送端需要等缓冲区满才发送出去，造成粘包（发送数据时间间隔很短，数据了很小，会合到一起，产生粘包）
- 接收方不及时接收缓冲区的包，造成多个包接收（客户端发送了一段数据，服务端只收了一小部分，服务端下次再收的时候还是从缓冲区拿上次遗留的数据，产生粘包）

◦ 解决方法:

- 发送定长包。如果每个消息的大小都是一样的，那么在接收对等方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。
  - 包头加上包体长度。包头是定长的 4 个字节，说明了包体的长度。接收对等方先接收包头长度，依据包头长度来接收包体。
  - 在数据包之间设置边界，如添加特殊符号 `\r\n` 标记。FTP 协议正是这么做的。但问题在于如果数据正文中也含有 `\r\n`，则会误判为消息的边界。
  - 使用更加复杂的应用层协议。
- TCP 滑动窗口：窗口是缓存的一部分，用来暂时存放字节流。发送方和接收方各有一个窗口，接收方通过 TCP 报文段中的窗口字段告诉发送方自己的窗口大小，发送方根据这个值和其它信息设置自己的窗口大小。

发送窗口内的字节都允许被发送，接收窗口内的字节都允许被接收。如果发送窗口左部的字节已经发送并且收到了确认，那么就将发送窗口向右滑动一定距离，直到左部第一个字节不是已发送并且已确认的状态；接收窗口的滑动类似，接收窗口左部字节已经发送确认并交付主机，就向右滑动接收窗口。

接收窗口只会对窗口内最后一个按序到达的字节进行确认，例如接收窗口已经收到的字节为 {31, 34, 35}，其中 {31} 按序到达，而 {34, 35} 就不是，因此只对字节 31 进行确认。发送方得到一个字节的确认之后，就知道这个字节之前的所有字节都已经被接收。

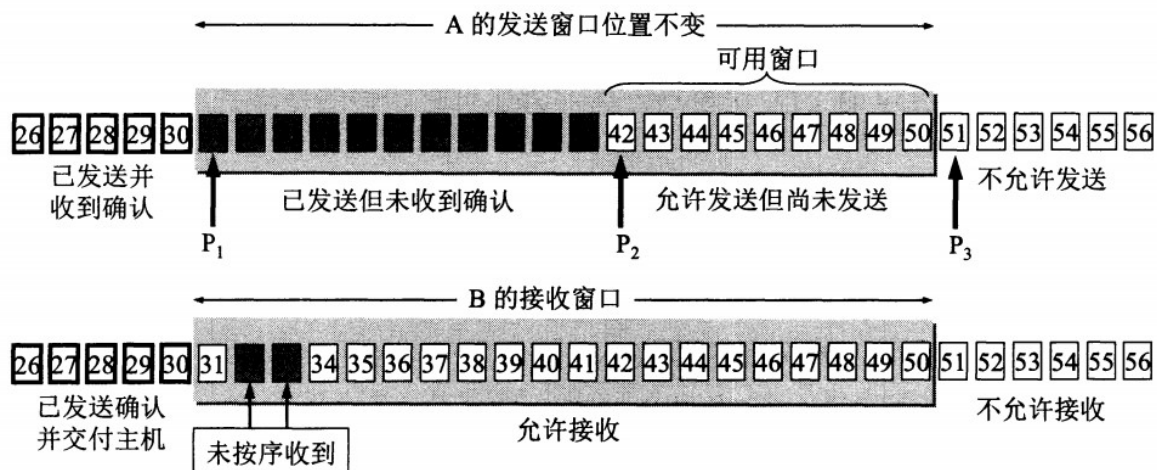


图 5-16 A 发送了 11 个字节的数据

- TCP 流量控制：流量控制（flow control）就是让发送方的发送速率不要太快，要让接收方来得及接收。接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。
- TCP 拥塞控制：

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

TCP 主要通过四个算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。

- 慢开始与拥塞避免：

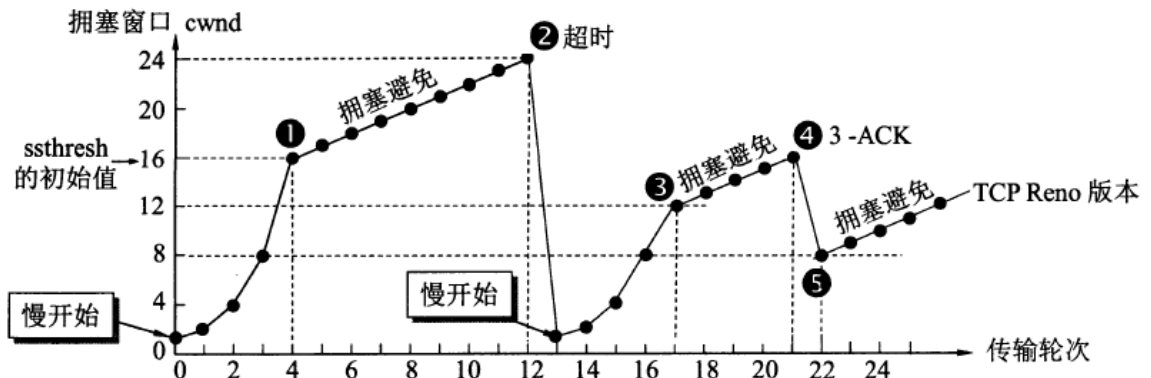


图 5-25 TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

发送的最初执行慢开始，令  $cwnd = 1$ ，发送方只能发送 1 个报文段；当收到确认后，将  $cwnd$  加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将  $cwnd$  加倍，这样会让  $cwnd$  增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限  $ssthresh$ ，当  $cwnd \geq ssthresh$  时，进入拥塞避免，每个轮次只将  $cwnd$  加 1。

如果出现了超时，则令  $ssthresh = cwnd / 2$ ，然后重新执行慢开始。

- 快重传与快恢复：

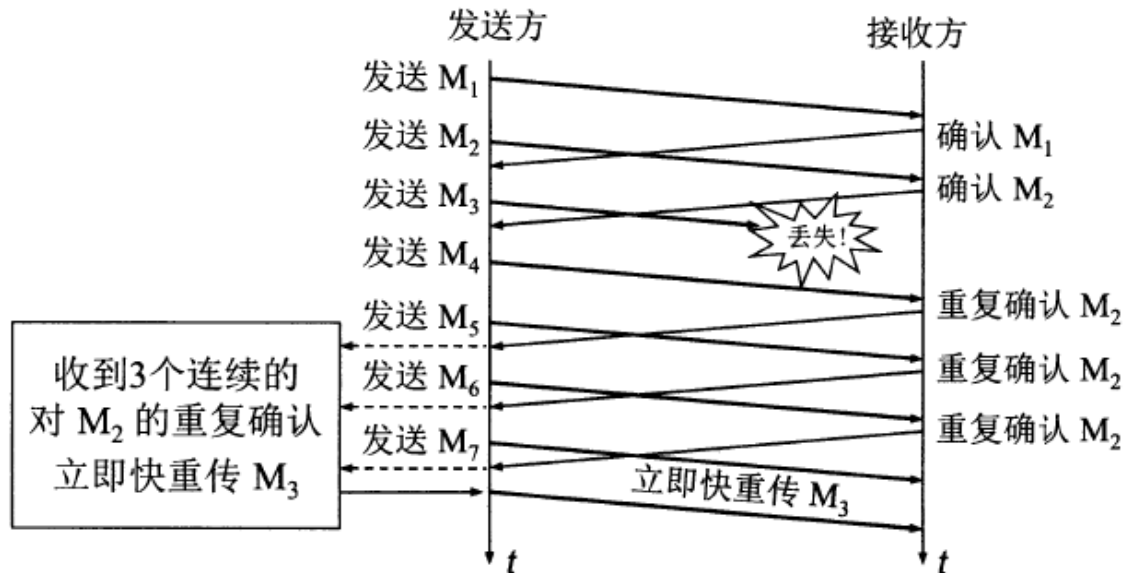


图 5-26 快重传的示意图

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到  $M_1$  和  $M_2$ ，此时收到  $M_4$ ，应当发送对  $M_2$  的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个  $M_2$ ，则  $M_3$  丢失，立即重传  $M_3$ 。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令  $ssthresh = cwnd / 2$ ， $cwnd = ssthresh$ ，注意到此时直接进入拥塞避免(直接令窗口大小等于阈值)。

慢开始和快恢复的快慢指的是  $cwnd$  的设定值，而不是  $cwnd$  的增长速率。慢开始  $cwnd$  设定为 1，而快恢复  $cwnd$  设定为  $ssthresh$ 。

## HTTP

- Http协议：HTTP（Hyper Text Transfer Protocol），超文本传输协议，是一种建立在TCP上的无状态连接，整个基本的工作流程是客户端发送一个HTTP请求，说明客户端想要访问的资源 and 请求的动作，服务端收到请求之后，服务端开始处理请求，并根据请求做出相应的动作访问服务器资源，最后通过发送HTTP响应把结果返回给客户端。
- Http工作在应用层，默认端口号为80
- 特点：
  - 支持客户/服务器模式
  - 简单快速
  - 灵活
  - 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。
  - 无状态：无状态是指协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。即我们给服务器发送 HTTP 请求之后，服务器根据请求，会给我们发送数据过来，但是，发送完，不会记录任何信息。

- HTTP使用统一资源标识符（Uniform Resource Identifiers, URI）来传输数据和建立连接，URI包括URL（Uniform Resource Locator）和URN（Uniform Resource Name）
- HTTP 消息结构：HTTP是基于客户端/服务端（C/S）的架构模型，通过一个可靠的链接来交换信息，是一个无状态的请求/响应协议。一个HTTP"客户端"是一个应用程序（Web浏览器或其他任何客户端），通过连接到服务器达到向服务器发送一个或多个HTTP的请求的目的。一个HTTP"服务器"同样也是一个应用程序（通常是一个Web服务，如Apache Web服务器或IIS服务器等），通过接收客户端的请求并向客户端发送HTTP响应数据。
- 客户端发送一个HTTP请求到服务器的请求消息包括以下格式：请求行（request line）、请求头部（header）、空行和请求数据四个部分组成。请求行中包含请求方式Method、资源路径URL、协议版本Version；
- 八种请求方式
  - GET：获取资源，当前网络请求中，绝大部分使用的是 GET 方法。
  - HEAD：获取报文首部，和 GET 方法类似，但是不返回报文实体主体部分。主要用于确认 URL 的有效性以及资源更新的日期时间等。
  - POST：主要用来传输数据，而 GET 主要用来获取资源。
  - PUT：上传文件，由于自身不带验证机制，任何人都可以上传文件，因此存在安全性问题，一般不使用该方法。
  - PATCH：对资源进行部分修改，PUT 也可以用于修改资源，但是只能完全替代原始资源，PATCH 允许部分修改。
  - DELETE：删除文件，与 PUT 功能相反，并且同样不带验证机制。
  - OPTIONS：查询支持的方法，查询指定的 URL 能够支持的方法。
  - CONNECT：要求在与代理服务器通信时建立隧道，使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加密后经网络隧道传输。
  - TRACE：追踪路径，服务器会将通信路径返回给客户端。发送请求时，在 Max-Forwards 首部字段中填入数值，每经过一个服务器就会减 1，当数值为 0 时就停止传输。通常不会使用 TRACE，并且它容易受到 XST 攻击（Cross-Site Tracing，跨站追踪）。
- GET和POST的比较：
  - GET 用于获取资源，而 POST 用于传输实体主体。
  - GET的参数在URL中，而POST的参数在实体中，如JSON
  - 安全：安全的 HTTP 方法不会改变服务器状态，也就是说它只是可读的。因此GET是安全的，而POST却不是，因为POST的目的是传送实体主体内容，这个内容可能是用户上传的表单数据，上传成功之后，服务器可能把这个数据存储到数据库中，因此状态也就发生了改变。
  - 幂等的HTTP方法，同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的。因此安全的方法同样是幂等的，而POST方法不是，比如增加多行记录。
  - 缓存：请求报文的 HTTP 方法本身是可缓存的，包括 GET 和 HEAD，但是 PUT 和 DELETE 不可缓存，POST 在多数情况下不可缓存的。
- HTTP响应由三部分组成：状态行、响应头、响应正文；
- c#中如何写HTTP请求：

```
// 定义url  
var url = _baseUrl + "/" + method;
```

```
// 新建连接, 选择Method为post
_request = new UnityWebRequest(url, "POST");
// 设置传输的内容
byte[] bodyRaw = Encoding.UTF8.GetBytes(body);
// 初始化上传handler, 设置请求正文
_request.uploadHandler = new UploadHandlerRaw(bodyRaw);
// 初始化下载handler
_request.downloadHandler = new DownloadHandlerBuffer();
// 设置请求头
_request.SetRequestHeader("Content-Type", "application/x-www-form-urlencoded");
// 传输并等待
_request.SendWebRequest();
while (!_request.isDone)
{
    Thread.Sleep(1);
}
// 定义Json类, 解析返回值
XAnalyseJsonBasic jsonAnalysed = JsonUtility.FromJson<XAnalyseJsonBasic>
(_request.downloadHandler.text);
```

## 同步I/O和异步I/O

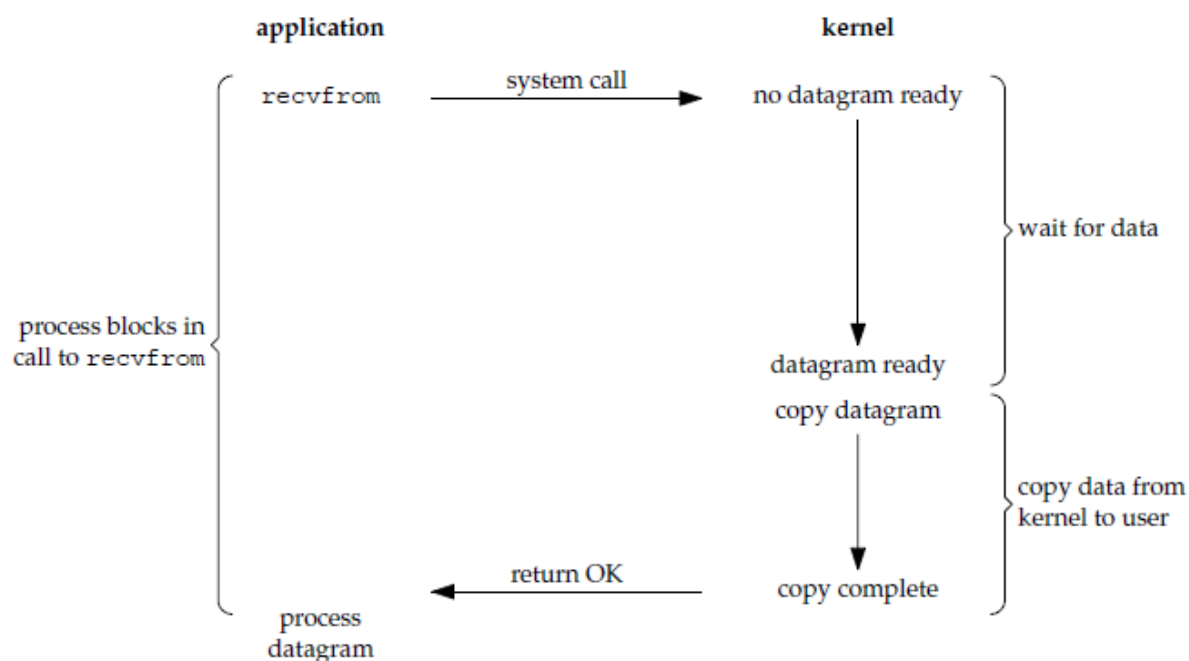
- 首先需要强调的是同步和异步的概念
  - 所谓同步, 就是在发出一个功能调用时, 在没有得到结果之前, 该调用就不返回。也就是必须一件一件事做, 等前一件做完了才能做下一件事。例如普通B/S模式（同步）: 提交请求->等待服务器处理->处理完毕返回 这个期间客户端浏览器不能干任何事
  - 异步的概念和同步相对。当一个异步过程调用发出后, 调用者不能立刻得到结果。实际处理这个调用的部件在完成后, 通过状态、通知和回调来通知调用者。例如 ajax请求（异步）: 请求通过事件触发->服务器处理（这是浏览器仍然可以作其他事情）->处理完毕
- 之后是阻塞和非阻塞
  - 阻塞调用是指调用结果返回之前, 当前线程会被挂起（线程进入非可执行状态, 在这个状态下, cpu不会给线程分配时间片, 即线程暂停运行）。函数只有在得到结果之后才会返回。有人也许会吧阻塞调用和同步调用等同起来, 实际上他是不同的。对于同步调用来说, 很多时候当前线程还是激活的, 只是从逻辑上当前函数没有返回, 它还会抢占cpu去执行其他逻辑, 也会主动检测io是否准备好。
  - 非阻塞和阻塞的概念相对应, 指在不能立刻得到结果之前, 该函数不会阻塞当前线程, 而会立刻返回。
- 阻塞的概念往往伴随着线程。阻塞一般是指: 在调用结果返回之前, 当前线程会被挂起。调用线程只有在得到结果之后才会被唤醒执行后续的操作。非阻塞式的调用指: 在结果没有返回之前, 该调用不会阻塞住当前线程。
- 区别: 阻塞/非阻塞和同步/异步的本质区别在于面向的对象是不同的。阻塞/非阻塞: 进程/线程需要操作的数据如果尚未就绪, 是否妨碍了当前进程/线程的后续操作。同步/异步: 数据如果尚未就绪, 是否需要等待数据结果。
- 五种IO模型:

- 首先需要明确，一个输入操作通常包括两个阶段：

- 等待数据准备好
- 从内核向进程复制数据

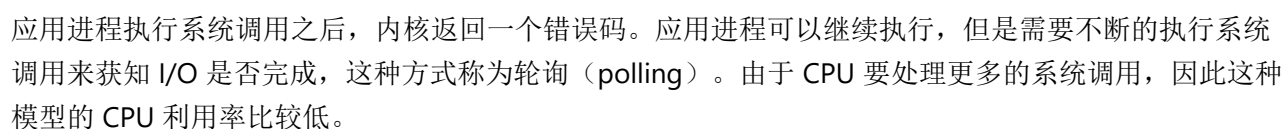
对于一个套接字上的输入操作，第一步通常涉及等待数据从网络中到达。当所等待数据到达时，它被复制到内核中的某个缓冲区。第二步就是把数据从内核缓冲区复制到应用进程缓冲区。

- 阻塞式 I/O:



应用进程被阻塞，直到数据从内核缓冲区复制到应用进程缓冲区中才返回。应该注意到，在阻塞的过程中，其它应用进程还可以执行，因此阻塞不意味着整个操作系统都被阻塞。因为其它应用进程还可以执行，所以不消耗 CPU 时间，这种模型的 CPU 利用率会比较高。

- 非阻塞式 I/O



```
sequenceDiagram
    participant Application
    participant Kernel
    Note over Application: process blocks in call to select, waiting for one of possibly many sockets to become readable
    Application->>Kernel: system call (select)
    Note over Kernel: no datagram ready
    Note over Kernel: wait for data
    Kernel->>Application: return readable
    Application->>Kernel: system call (recvfrom)
    Note over Kernel: datagram ready
    Note over Kernel: copy datagram
    Note over Kernel: copy data from kernel to user
    Kernel->>Application: return OK
    Note over Application: process datagram
```

The diagram illustrates the interaction between an application and a kernel for data reception. It is divided into two main sections: the application side and the kernel side.

**Application Side:**

- First Phase:** The application calls `select`. The process blocks in this call, waiting for one of possibly many sockets to become readable.
- Second Phase:** The application calls `recvfrom`. The process blocks while data is copied into the application buffer.
- Final Step:** The application processes the datagram.

**Kernel Side:**

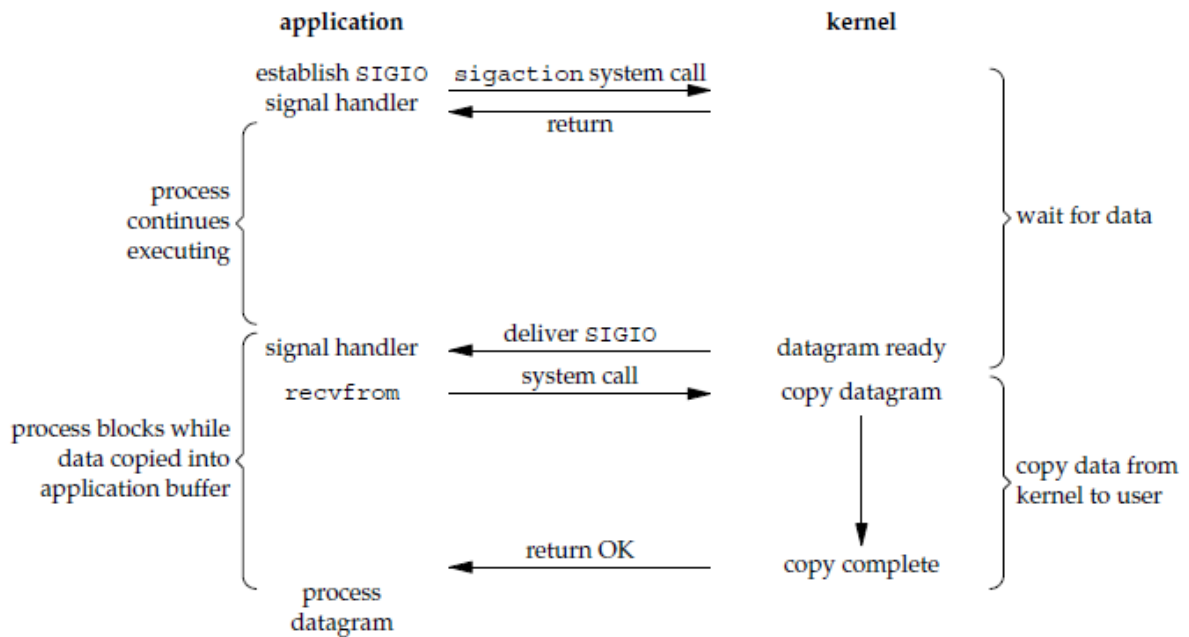
- First Phase:** Upon receiving the `select` system call, the kernel checks for data. If no datagram is ready, it enters a "wait for data" state.
- Second Phase:** Once a datagram is ready, the kernel copies it. This involves copying data from the kernel to the user.
- Final Step:** After the copy is complete, the kernel returns `OK` to the application.

**System Call Transitions:**

- The application initiates the process with a `select` system call.
- After receiving `return readable` from the kernel, the application makes a `recvfrom` system call.
- After receiving `return OK` from the kernel, the application processes the datagram.

如果一个 Web 服务器没有 I/O 复用，那么每一个 Socket 连接都需要创建一个线程去处理。如果同时有几万个连接，那么就需要创建相同数量的线程。相比于多进程和多线程技术，I/O 复用不需要进程线程创建和切换的开销，系统开销更小。

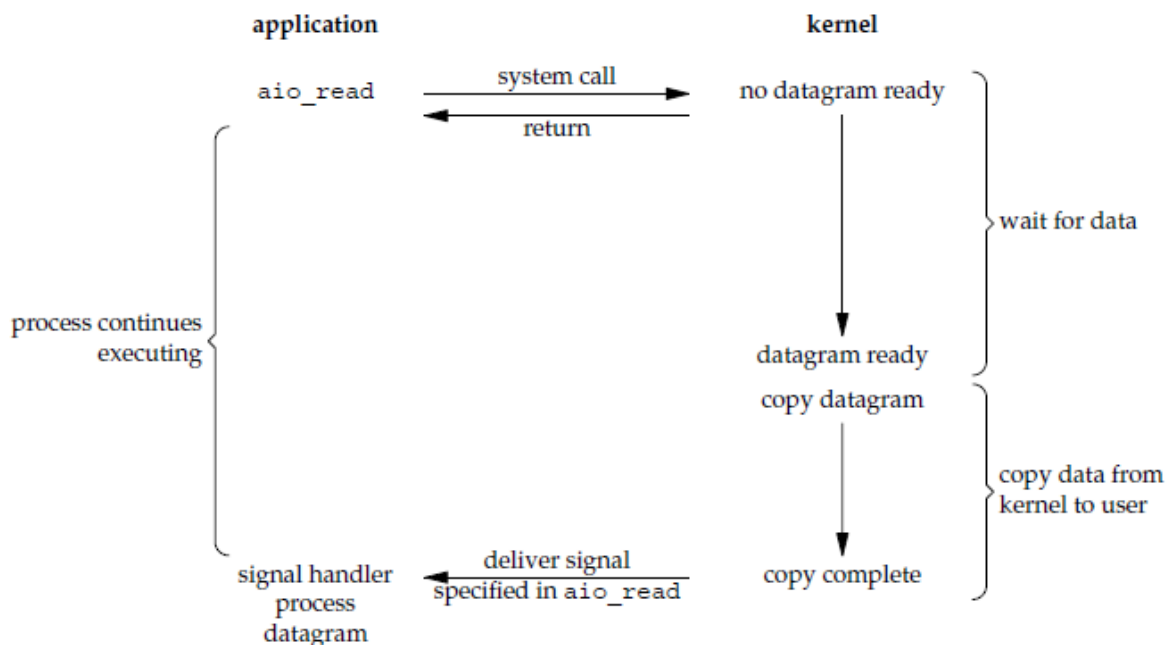
## 63 / 81



应用进程使用 `sigaction` 系统调用，内核立即返回，应用进程可以继续执行，也就是说等待数据阶段应用进程是非阻塞的。内核在数据到达时向应用进程发送 `SIGIO` 信号，应用进程收到之后在信号处理程序中调用 `recvfrom` 将数据从内核复制到应用进程中。

相比于非阻塞式 I/O 的轮询方式，信号驱动 I/O 的 CPU 利用率更高。

- 异步 I/O

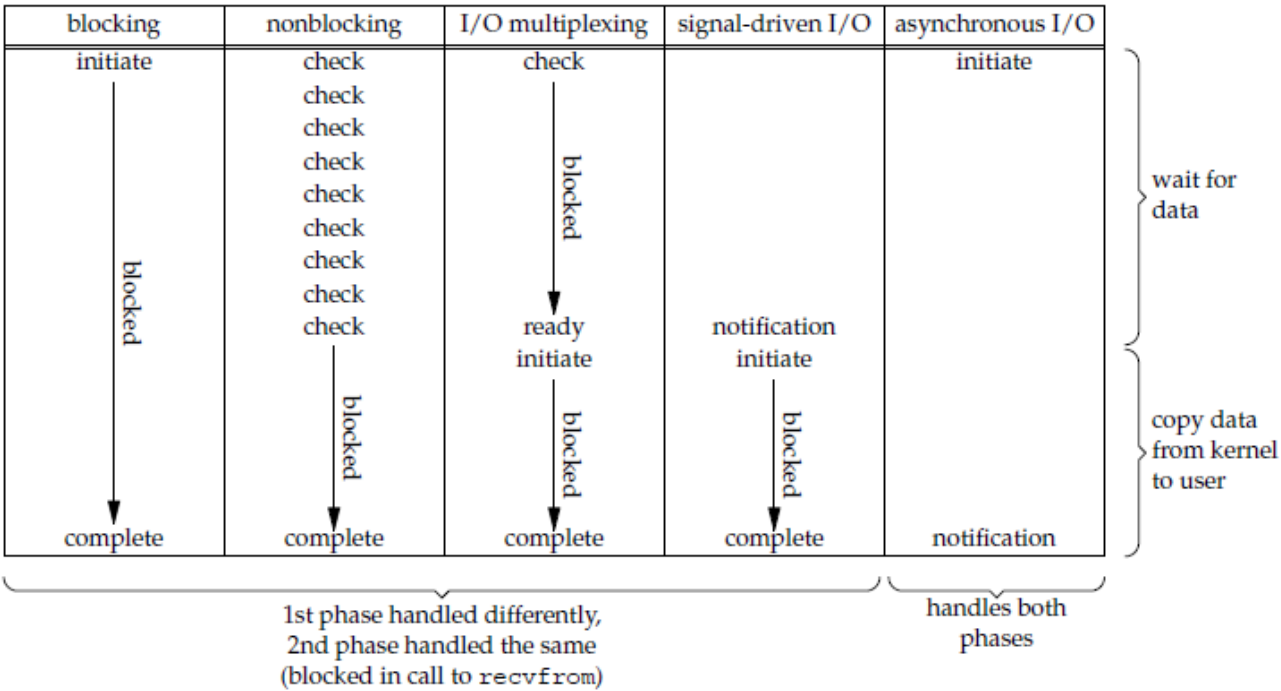


应用进程执行 `aio_read` 系统调用会立即返回，应用进程可以继续执行，不会被阻塞，内核会在所有操作完成之后向应用进程发送信号。

异步 I/O 与信号驱动 I/O 的区别在于，异步 I/O 的信号是通知应用进程 I/O 完成，而信号驱动 I/O 的信号是通知应用进程可以开始 I/O。



- 同步I/O与异步I/O 前四种都是同步I/O，它们的共同点是，在数据被复制到进程缓冲区时（第二阶段），进程会被阻塞。而异步I/O的第二阶段不会阻塞。



数据的序列化与反序列化

序列化是说，将变量和实例这些数据，依照某种约定，转化（通常伴随着压缩）为一种通用的数据格式；转化后的数据，可以用来储存或者传输，以备下次读取使用。其中提到的格式可以是二进制的，也可以是字符串式的。反序列化，就是上述过程的补集：将序列化的数据读入，解析为编程语言可识别的数据结构的过程。

数据序列化的目的是解决跨编程语言、跨操作系统平台的数据传输和储存的问题。

数据库

关系型数据库

- 关系型数据库最典型的数据结构是表，由二维表及其之间的联系所组成的一个数据组织
- 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，可以是文档或者键值对等。存储数据的格式可以是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，使用灵活，应用场景广泛，而关系型数据库则只支持基础类型。

数据库事务(Transaction)

- 事务指的是满足 ACID 特性的一组操作，可以通过 Commit 提交一个事务，也可以使用 Rollback 进行回滚。
- 原子性（Atomicity）：事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。回滚可以用回滚日志来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。
- 一致性（Consistency）：数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对一个数据的读取结果都是相同的,即数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。例如对

银行转帐事务，不管事务成功还是失败，应该保证事务结束后ACCOUNTS表中Tom和Jack的存款总额为2000元。

- 隔离性（Isolation）：一个事务所做的修改在最终提交以前，对其它事务是不可见的。
- 持久性（Durability）：一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。使用重做日志来保证持久性。
- 事务的 ACID 特性概念简单，但不是很好理解，主要是因为这几个特性不是一种平级关系：
  - 只有满足一致性，事务的执行结果才是正确的。
  - 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时只要能满足原子性，就一定能满足一致性。
  - 在并发的情况下，多个事务并行执行，事务不仅要满足原子性，还需要满足隔离性，才能满足一致性。
  - 事务满足持久化是为了能应对数据库崩溃的情况。

## 计算机组成原理

---

### 结构体对齐

- 结构体对齐规则
  - 计算结构体大小不是元素单纯相加；
  - 32位CPU取四个字节比一个字节更高效方便；
  - 若每个元素首地址都是4的整数倍，取数据元素就能更高效方便；
  - 每个特定平台上编译器都有自己的默认的对齐系数（对齐模数），可通过编译命令来改变这一系数 `#pragma pack(n);` // `n=1,2,4,8,16`, 其中 `n` 就是你想要指定的系数；
  - 一般情况下32位默认4字节对齐。
- 原则1：结构体中元素是按照定义顺序一个一个放到内存中去的，但并不是紧密排列的。从结构体存储的首地址开始，每一个元素放置到内存中时，它都会认为内存是以它自己的大小来划分的，因此元素放置的位置一定会在自己宽度的整数倍上开始（以结构体变量首地址为0计算）

0	a (char)
1	
2	
3	
4	b (int)
5	
6	
7	
8	c (double)
9	
10	
11	
12	
13	
14	
15	

图 1 结构体 S1 存储结构示意图

- 原则2：在经过第一原则分析后，检查计算出的存储单元是否为所有元素中最宽的元素的长度的整数倍，是，则结束；若不是，则补齐为它的整数倍。

0	a (char)
1	
2	
3	
4	
5	
6	
7	
8	b (double)
9	
10	
11	
12	
13	
14	
15	
16	c (int)
17	
18	
19	
20	补齐到 8 的 整数倍
21	
22	
23	

图 2 结构体 S2 存储结构示意图

- 原则3：指针本身所占的存储空间是4个字节就行了，而不必看它是指向什么类型的指针

## 大小端

## 各种编码（补码、原码）的定义和转换

# 面向对象

---

三大特征：封装、继承、多态

## 封装

把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。关键字：public, protected, private。不写默认为 private。

- public 成员：可以被任意实体访问
- protected 成员：只允许被子类及本类的成员函数访问
- private 成员：只允许被本类的成员函数、友元类或友元函数访问

## 继承

基类（父类）—> 派生类（子类）

- 公有继承、保护继承、私有继承三种继承的区别主要在于父类成员在派生类中的可见性：

	public	protected	private
公有继承	public	protected	不可见
私有继承	private	private	不可见
保护继承	protected	protected	不可见

## 多态

- 多态，就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。简单来说，我们可以将多态定义为消息以多种形式显示的能力。多态是以封装和继承为基础的。
- C++ 多态分类及实现：
  - 重载多态（Ad-hoc Polymorphism，编译期）：函数重载、运算符重载
  - 参数多态性（Parametric Polymorphism，编译期）：类模板、函数模板
  - 子类型多态（Subtype Polymorphism，运行期）：虚函数
  - 强制多态（Coercion Polymorphism，编译期/运行期）：基本类型转换、自定义类型转换
  - 因此，c++中多态大致分为两种，运行时多态（动态多态）和编译时多态（静态多态）。动态多态最常见的用法就是声明基类的指针，利用该指针指向任意一个子类对象，调用相应的虚函数，可以根据指向的子类的不同而调用不同的方法。静态多态往往通过函数重载和模版（泛型编程）来实现。
- 重载多态（静态多态，编译期/早绑定）：函数重载

```
class A
{
public:
    void do(int a);
    void do(int a, int b);
};
```

- 子类型多态（动态多态，运行期/晚绑定）：虚函数：用 **virtual** 修饰成员函数，使其成为虚函数
  - 注意：
    - 普通函数（非类成员函数）不能是虚函数
    - 静态函数（**static**）不能是虚函数
    - 构造函数不能是虚函数（因为在调用构造函数时，虚表指针并没有在对象的内存空间中，必须要构造函数调用完成后才会形成虚表指针）
    - 内联函数不能是表现多态性时的虚函数，
  - 动态多态使用：

```
class Shape // 形状类
{
public:
    virtual double calcArea()
    {
        ...
    }
    virtual ~Shape();
};
class Circle : public Shape // 圆形类
{
public:
    virtual double calcArea();
    ...
};
class Rect : public Shape // 矩形类
{
public:
    virtual double calcArea();
    ...
};
int main()
{
    Shape * shape1 = new Circle(4.0);
    Shape * shape2 = new Rect(5.0, 6.0);
    shape1->calcArea(); // 调用圆形类里面的方法
    shape2->calcArea(); // 调用矩形类里面的方法
    delete shape1;
    shape1 = nullptr;
    delete shape2;
    shape2 = nullptr;
```

```
    return 0;
}
```

- 虚析构造函数：虚析构造函数是为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象。

```
class Shape
{
public:
    Shape();                // 构造函数不能是虚函数
    virtual double calcArea();
    virtual ~Shape();       // 虚析构造函数
};
class Circle : public Shape // 圆形类
{
public:
    virtual double calcArea();
    ...
};
int main()
{
    Shape * shape1 = new Circle(4.0);
    shape1->calcArea();
    delete shape1; // 因为Shape有虚析构造函数，所以delete释放内存时，先调用
                  // 子类析构函数，再调用基类析构函数，防止内存泄漏。
    shape1 = NULL;
    return 0;
}
```

- 纯虚函数：纯虚函数是一种特殊的虚函数，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。

```
virtual int A() = 0;
```

- 虚函数和纯虚函数的区别:
  - 类里如果声明了虚函数，这个函数是实现的，哪怕是空实现，它的作用就是为了让这个函数在它的子类里面可以被覆盖，这样的话，编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。
  - 虚函数在子类里面也可以不重载的；但纯虚函数必须在子类去实现。
  - 虚函数的类用于“实作继承”，继承接口的同时也继承了父类的实现。当然大家也可以完成自己的实现。纯虚函数关注的是接口的统一性，实现由子类完成。
  - 带纯虚函数的类叫抽象类，这种类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。抽象类被继承后，子类可以继续是抽象类，也可以是普通类。
  - 抽象类：含有纯虚函数的类。接口类：仅含有纯虚函数的抽象类
  - 虚基类是虚继承中的基类。

- 虚函数指针和虚函数表：
  - 虚函数指针：在含有虚函数类的对象中，指向虚函数表，在运行时确定。
  - 虚函数表：在程序只读数据段（`.rodata section`，见：目标文件存储结构），存放虚函数指针，如果派生类实现了基类的某个虚函数，则在虚表中覆盖原本基类的那个虚函数指针，在编译时根据类的声明创建。
  - 关于虚函数表的覆盖，[这篇文章讲得很好](#)
- 虚继承
  - 虚继承用于解决多继承条件下的菱形继承问题（浪费存储空间、存在二义性）。
  - 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（占用一个指针的存储空间，4字节）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）；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。
  - 实际上，`vbptr` 指的是虚基类表指针（`virtual base table pointer`），该指针指向了一个虚基类表（`virtual table`），虚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。
- 虚继承和虚函数
  - 相同之处：都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）
  - 不同之处：
 

虚继承：虚基类依旧存在继承类中，只占用存储空间。虚基类表存储的是虚基类相对直接继承类的偏移

虚函数：虚函数不占用存储空间，虚函数表存储的是虚函数地址

## 数学

---

去祖龙面试时第一次被问到了数学题

- 给定 $n$ 个苹果和 $m$ 个篮子，要求将 $n$ 个苹果随机分到 $m$ 个篮子中，并且每个篮子中的苹果数量不超过 $2/m$ 个。
 

理解起来是：将每个篮子视为 $m/2$ （下取整）个小格，所以每个一共有 $n*m/2$ 个小格，从这些小格中取 $n$ 个放置苹果即可满足要求。
- 蓄水池算法：就是一种均匀采样 $k$ 个的算法，比如从一副牌中随机选择 $n$ 张。该算法也可以适用于从一个字符流中进行采样，最后保留 $n$ 个字符，而并不知道这个流什么时候结束，且须保证每个字符被采样到的几率相同。
  - 先选取数据流中的前 $k$ 个元素，保存在集合 $A$ 中；
  - 从第 $j$ （ $k + 1 \leq j \leq n$ ）个元素开始，每次先以概率 $p = k/j$ 选择是否让第 $j$ 个元素留下。若 $j$ 被选中，则从 $A$ 中随机选择一个元素并用该元素 $j$ 替换它；否则直接淘汰该元素；
  - 重复步骤2直到结束，最后集合 $A$ 中剩下的就是保证随机抽取的 $k$ 个元素。

- 同时这里也复习一下洗牌算法
- 错排公式:

递推：现在我们将 $n$ 封信装入 $n$ 个信封，考察当我们装第 $n$ 封信是的情况：

- 第一种：前面的 $n-1$ 封信已经满足错排，这时我们从其中选择一封信，跟第 $n$ 封信交换，则结果依然满足错排，所以这种情况有 $(n-1) * D(n-1)$
- 第二种：前面的 $n-1$ 封信中，有一封信是装对的，其余 $n-2$ 封信错排，这时我们同样可以将这封装对的信与第 $n$ 封信交换，结果同样是错排，因此这种情况为 $(n-1)*D(n-2)$

因此最终的递推公式为：

$$D(n) = (n-1) [D(n-2) + D(n-1)]$$

特别的 $D(1) = 0, D(2) = 1$

- 洗牌算法：
  - 问题：从52张牌中抽取13张，要求抽到的牌是完全随机和等概率的；
  - 最直接的想法是，有一个能够生成1-52的随机函数，我们连续生成多次 $i$ ，每生成一次 $i$ ，就取对应的第 $i$ 张牌，如果这张牌已经被取走，则重新生成随机数。这种方法有个问题是如果我们取51张牌，越到后面，我们重新生成随机数的概率就越大，导致效率很低。
  - 一个很好的改进是，我们可以第一次生成1-52，取走一张牌后，牌堆中还有51张牌，这时我们随机生成1-51，这样再到牌堆中取，可以保证每次都能取成功。其实这跟之后的Fisher-Yates Shuffle洗牌算法就十分接近了。
  - 转化：如果我们能将这52张牌全部洗乱，则我们取其中任意13张，都是符合问题要求的结果；
  - Fisher-Yates Shuffle 算法：
    1. 初始化原始数组和新数组，原始数组长度为 $n$ (已知)；
    2. 从还没处理的数组（假如还剩 $k$ 个）中，随机产生一个 $[0, k)$ 之间的数字 $p$ （假设数组从0开始）；
    3. 从剩下的 $k$ 个数中把第 $p$ 个数取出；
    4. 重复步骤2和3直到数字全部取完；
    5. 从步骤3取出的数字序列便是一个打乱了的数列。

一个元素 $m$ 被放入第 $i$ 个位置的概率 $P =$  前 $i-1$ 个位置选择元素时没有选中 $m$ 的概率 \* 第 $i$ 个位置选中 $m$ 的概率，即：

$$P = \frac{n-1}{n} \times \frac{n-2}{n-1} \times \dots \times \frac{n-i+1}{n-i+2} \times \frac{1}{n-i+1}$$

$$= \frac{1}{n}$$

- Knuth-Durstenfeld Shuffle 算法：



1. 建立一个数组大小为  $n$  的数组 `arr`，分别存放 1 到  $n$  的数值；
2. 生成一个从 0 到  $n - 1$  的随机数  $x$ ；
3. 输出 `arr` 下标为  $x$  的数值，即为第一个随机数；
4. 将 `arr` 的尾元素和下标为  $x$  的元素互换；
5. 同2，生成一个从 0 到  $n - 2$  的随机数  $x$ ；
6. 输出 `arr` 下标为  $x$  的数值，为第二个随机数；
7. 将 `arr` 的倒数第二个元素和下标为  $x$  的元素互换；
8. ...

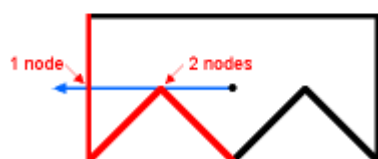
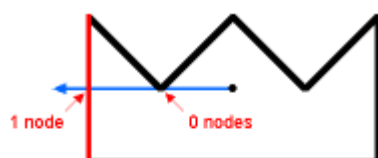
如上，直到输出  $m$  个数为止 总结一下就是每次对第  $i$  个数进行操作，生成一个  $1-i$  的随机数，然后将第  $i$  个数跟随机数位置的牌进行交换即可。之后  $i=i-1$

## 点乘与叉乘的区别

- 点乘，也叫数量积。结果是一个向量在另一个向量方向上投影的长度，是一个标量。
- 叉乘，也叫向量积。结果是一个和已有两个向量都垂直的向量。
- 一般点乘用来判断两个向量是否垂直，也可以用来计算一个向量在某个方向上的投影长度，就像定义一样。
- 叉乘更多的是判断某个平面的方向。叉乘的结果是一个与参与运算的两个向量  $a$  和  $c$  都垂直的一个向量，其模长为  $a$  和  $b$  模长乘它们夹角的正弦值。二维平面中，叉乘等于由向量  $a$  和向量  $b$  构成的平行四边形的面积。当然，叉乘也可以用来判断一个点在一条直线的哪一侧，主要是通过夹角是否超过  $180$  度来进行判断。
- 向量平行比较好判断，直接看两个向量的坐标是否成比例或者看叉乘结果是否为  $0$  也可以

## 如何判断一个点在多边形内部

- 射线法：从这个点向外部射出一条射线，与求出所有与多边形的交点，如果交点个数为奇数个，则在内部，否则在外部。但有一些特殊情况，就是平行和交到顶点时的情况。
  - 与射线平行的边可以不考虑
  - 与顶点相交时，判断这个顶点是不是其所在边纵坐标最大的点，如果是，则计数一次，否则不计数。也就是说，一个顶点最少关联两条边，那么它的计数结果可能为  $0$ 、 $1$  或  $2$ 。下面是例子：



## 如何判断一个点在三角形内部

- 叉乘法：除了上面说的射线法外，对三角形内部的判断还可以用叉乘法。叉乘法的本质是计算一个点在一条直线的哪一侧，所以如果一个点在三条按照顺序连起来的边的同侧，则说明其在这个三角形内部，否则在外部。

形式化描述就是，对于一个三角形ABC和点P，先计算向量PA、PB、PC，然后计算它们之间的叉乘，计算时要按顺序， $PA \times PB$ 、 $PB \times PC$ 、 $PC \times PA$ ，如果三个结果同正或同负，则说明P在ABC内部。

- 也可以使用面积法，求三个点P和任意两顶点组成的三角形面积之和是否与ABC组成的三角形面积相等。

## 游戏相关

### A\*寻路:有点类似迪杰斯特拉算法

1. 把起始格添加到开启列表。

2. 重复如下工作：

1. 寻找开启列表中F值最低的格子。我们称它为当前格。

2. 把它切换到关闭列表。

3. 对相邻的8格中的每一个格进行如下判断：

如果它不可通过或者已经在关闭列表中，略过它。反之如下。

如果它不在开启列表中，把它添加进去。把当前格作为这一格的父节点。记录这一格的  $F (F=G+H)$  ,G（表示沿路径从起点到当前点的移动耗费）和H值（可以是当前格到目的格之间水平和垂直的方格的数量总和）。

如果它已经在开启列表中，用G值为参考检查新的路径是否更好。更低的G值意味着更好的路径。如果是这样，就把这一格的父节点改成当前格，并且重新计算这一格的G和F值。如果你保持你的开启列表按F值排序，改变之后你可能需要重新对开启列表排序。

4. 停止，当你把目标格添加进了关闭列表，这时候路径被找到，或者没有找到目标格，开启列表已经空了。这时候，路径不存在。

3. 保存路径。从目标格开始，沿着每一格的父节点移动直到回到起始格。这就是你的路径。

### 游戏中的AI表示

- if-else 条件分支
- 状态机
- 行为树
- 强化学习

### ECS

### RVO、HRVO、RVO2

### 帧同步和状态同步

## PBR技术

## 程序化生成植被技术

## 面经

---

- 自我介绍
- 死锁的原因、必要条件
- 多态，虚函数的目的，多态实现是依靠什么：虚函数表
- UDP TCP
- 数据库的备份是如何实现的
- 黑盒白盒测试
- 红黑树
- 进程和线程
- APP是用多进程安全还是用多线程安全
- 进程通信和线程通信
- windows和linux调度进程咋回事
- 关系型数据库有哪些，知道NOSQL吗？
- 快速排序是什么过程？其思想是什么？
- C++11中你认为比较惊艳的设计
- lamda表达式常用吗？
- 闭包是什么？
- C++新增的unorderedmap底层是用什么实现的？查找元素的时间复杂度是？map呢？
- vector动态扩容的机制
- 当String内容小于16字节时，是放在堆中还是栈中，比较长的时候又放在哪里
- C++语言多态的实现是依靠什么
- 计算机网络几层，TCP/IP是在哪一层，他下面还有哪些层
- HTTP是什么，他在哪一层
- SOCKET是什么，经常使用吗？
- 编程题：strcpy的实现
- linux基本命令

- LeetCode第86题
- OOP三大特性
- 二叉树，八叉树
- 进程与线程
- 高并发
- 你认为自己有能力胜任此工作的5点优势
- 说一说dynamic\_cast
- 堆和栈有什么区别
- 数据结构上的堆和栈有什么区别、底层结构是什么
- 快排的时间复杂度、最坏情况呢
- 算法题：找无序数组中第k个数（一开始说用堆实现、后来我又想着用快排的partition实现）
- 智能指针
- 循环引用是什么
- 说一说epoll，为什么epoll用双向链表
- 说一说select
- epoll和select各自应用场景
- 说一说reactor、proactor，两者有什么区别
- 定时器怎么做的（时间堆）、时间复杂度是多少
- 还有其他定时器吗（时间轮）、时间复杂度
- LeetCode第1题
- 判断一个字符串是否为另一个字符串子串
- 从字符串S变到T，插入消耗2、删除消耗2、替换消耗3、求最小消耗
- 两个栈实现一个队列（实现push、pop、count三个函数）
- LeetCode第72题
- 讲讲http
- get和post区别
- 同步异步
- 除了select，poll，epoll，还了解哪些I/O复用吗

- TCP断开连接，讲讲Close\_Wait
- 讲讲乐观锁，悲观锁
- 讲讲数据库，索引
- 讲讲深浅拷贝
- 两个有序数组找相同元素，我说双指针，无序数组呢，我说我用dict做，空间换时间
- 一堆需求，每个需求都有需要完成的时间和deadline，求最多能完成几个
- $n*n$ 格子，从左下走到右上。
- 说说Time\_wait
- 说说volatile
- 说说数据库四大特性
- STL了解吗，说说常用的。然后他说set，hashmap呢？
- 现在有一个很大很大的地图，上面有很多角色（怪物，NPC等），每个角色视为一个质点，有一个坐标（float型）。要怎么设计底层的存储结果。
- 字节跳动提前批面经

笔试包含三种题型：选择、简答和编程，简单记录一下：

- 选择
  1. 下列哪些算法是基于比较的排序算法：桶排序、基数排序、冒泡排序、希尔排序（冒泡排序和希尔排序）
  2. 子网掩码:255.255.255.0，最多接入多少个主机（答案是254，即2的8次方减2，减去的2是全0的网络地址和全1的广播地址）
  3. hash\_map中能否存null值和null键（可以，但hash\_table不可以）
- 简答
  1. 网络传输层协议在游戏中的应用（[腾讯游戏学院](#)，总结一下就是如果是由客户端间歇性的发起无状态的查询，并且偶尔发生延迟是可以容忍，那么使用HTTP/HTTPS吧。如果客户端和服务端都可以独立发包，但是偶尔发生延迟可以容忍（比如：在线的纸牌游戏，许多MMO类的游戏），那么使用TCP长连接吧。如果客户端和服务端都可以独立发包，而且无法忍受延迟（比如：大多数的多人动作类游戏，一些MMO类游戏），那么使用UDP吧。这些也应该考虑在内：你的MMO客户端也许首先使用HTTP去获取上一次的更新内容，然后使用UDP跟游戏服务器进行连接。）
  2. 数组、链表、vector的区别（主要是Vector:Vector在程序员角度上是可变长的，但本质上还是定长的，如果一开始分配的空间不够的话，会重新重新分配一个更大的空间，然后进行元素拷贝。）
  3. 同步io与异步io的区别（所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做，等前一件做完了才能做下一件事。同步，就是我调用一个功能，该功能没有结束前，我死等结果。异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。异步，就是我调用一个功能，不需要知道该功能结果，该功能有结果后通知我（回调通知）。）
  4. 设计一个monster的AI

- 编程

1. 输出链表中终点距离为k的节点值（水题）
  2. Leetcode 135 Candy
  3. 马里奥跳板。大意是，给定一个数组，[3,0,2,1]，和一个起始位置p，数组的值表示最大跳跃的距离，问从起始位置能够跳到终点，即数组长度加1的地方。（简单的BFS）
3. 7.5日十一点半接到短信和邮件，通知视频面试，但可供选择的周六和周日两个时间已经全部灰了。第二天再看，可供选择的天数变多了，所以约了周二下午3点。
4. 7.8日晚十点，电话通知修改面试时间，改到了周四上午十一点。
5. 最后也没过，写个面经供大家参考吧：

1. 中午十一点开始一面：时长约40分钟（代码题给了20分钟）

1. `const` 和 `static` 的区别，修饰不同变量或函数时的区别，变量的生命周期？看一道代码题，说说第三次运行`func()`输出的内容。

```
//  
// Created by YCIrving on 2019-07-15.  
//  
  
#include <stdio.h>  
  
void func() {  
    static int x = 1; // 在对func的三次调用中,x只进行一次初始化  
    x = x + 1;  
    printf("%d\n", x); // 输出x的值  
}  
  
int main(int argc, char * const argv[]) {  
    func(); // 输出2  
    func(); // 输出3  
    func(); // 输出4  
    return 0;  
}
```

2. 渲染管线了解吗？主要说了渲染流程，谈了谈顶点shader和片段shader
  3. 说一种你熟悉的设计模式？目前只接触到了MVC
  4. 基本STL的底层实现及复杂度？包括`vector`、`map`、`list`
  5. 构造函数和析构函数能否是虚函数？这道题没答好，复习的时候只注意到析构函数可以是虚函数，所以猜测构造函数也可以是虚函数，下来看了答案，只有析构函数可以是虚函数。
  6. 聊了聊项目，问了一下项目中地图编辑器有没有想过撤销操作怎么实现？树形结构怎么解析？
  7. 代码题：十分简单的dp，类似于上楼梯，原题目是给定一个n，从零开始，每次操作只能加6或者加8，输出到达n的最小操作次数。
2. 面试结束半小时接到电话，下午三点二面：

1. 说说C++中的`union`（联合体）？没复习到/(ToT)/~~~

2. 说说如何理解`*(void (*)())0()`的含义吧？我.....（后来面试官给了思路，说里面涉及函数的声明和调用、类型转换、函数指针）

```
void(*)()      // a pointer-to-function type, taking unspecified
parameters
               // and returning nothing.
(void(*)())0    // a null pointer of that pointer-to-function
type
*(void(*)())0   // dereference that pointer
*(void(*)())0(); // and call it with no parameters
```

3. 说说C++中，子类继承父类时，子类实例的内存布局？复习时只看了虚函数表的覆盖，只说了这部分
4. 看你项目用到了Unity，说说c#中装箱和拆箱？一脸懵逼:)
5. 那说说C++和C#是类型安全的语言吗？反问了什么叫类型安全，面试官说就是一个概念，我说不知道。面试官友好地做了解答，可以简单理解为能否直接操作内存，C++中有指针，所以不是类型安全的。
6. 那说说unity中的协程(coroutine)吧？我说自己项目之前打算用，但是写法比较复杂，所以用了`while(true)`，然后不断检查状态的方式实现同步
7. 那说说Unity中的`yield return`吧，跟你上面的`while(true)`有什么区别？我说`yield return`没有按照我预期的顺序执行代码，所以也没用。面试官继续友好地给出了答案，说`yield return`类似于语法糖，是逐帧进行轮询的，而`while (true)`是按照时间片来轮询的，所以`yield return`效率上会更好一些。（5和6感觉是自己回答最不好的两题，自己下来反思了一下，回答从侧面反映了自己遇到问题会找不同的方法解决，但没有深挖某种方法，比如效率更高的coroutine，只求能运行，这种态度很不好。）
8. Lua熟悉吗？我说项目正在用，只了解基本的八种数据类型。面试官说.....那算了。
9. 看看你逻辑吧？平面上有一个三角形，已知三个顶点的坐标信息，现在给你另外一个点的坐标，怎么判断这个点是否在三角形内部？看我画了半天，面试官提示说点乘叉乘说一下计算方法和几何意义，我说完之后还是不会。面试官说没关系，就是一些技巧方面的判断。
10. 说说大顶堆、小顶堆的结构有什么要求？如何删除一个堆顶元素？
11. 聊了聊动规的本质，我说是状态的转移，通过避免访问已经访问过的状态来提高效率。然后面试官又以最长递增子序列和切木材的例题让我讲一下转移方程（切木材问题是说不同长度的木材有不同的价值，现在有一条长度为n的木材，问怎么切价值最大。我开始说像恰好装满的完全背包问题，面试官说有区别，让我写方程，我写了 $dp(n) = \max(dp(n-l_i) + v_i)$ ，面试官还说不对，说 $dp(n)$ 只跟 $dp(n-1)$ 有关，最后我还是觉得有问题，但时间关系只能先接受面试官的意见，然后说我下去再思考一下）
12. 最后让写一道代码题，是字符串解码。题目是给定一个字符串，目标是将写在#和%之间的字符串，重复#号前面的数字次(数字只会是个位数)，比如下面这个`he3#llo%world`，解码后是`hellolllolloworld`。比较难的点是存在嵌套，比如`he3#l12#o%wo%rld`，答案是`helloowolloowollooworld`，（原题的样例比这个好看一些，但我实在想不起来了）。类似括号匹配，但还涉及到字符串的复写啥的，规定时间没写出来。面试官说除了使用基本的栈以外，还需要一些数据结构的支持，存储字符串起止点之类的信息。

3. 二面结束后，一个小时没有接到通知，遂加了Hr微信，确认了确实GG。整个面试“轻松愉快”，面试官友好热情，温文尔雅，给五星好评，秋招还会再来。

4. 玩笑归玩笑，总结一下自己这次面试失败学到的教训：

1. 自己只找游戏开发岗，是底线，所以这次面试的目的就是找工作，并不是积累经验，没过有点不开心。
2. 一面答错了一些问题，主要原因是临时抱佛脚，知识没有深入理解和形成体系。
3. 二面难度不太适应，基本没有完全符合面试官预期的答案，跟我自己看到的二面面经也有差距（说好的tcp udp呢啊喂！）。上来几个问题问懵了之后就开始慌了，以后心态需要调整好，面试没结束之前永远不要放弃，这也是自己抗压能力的一种体现。
4. 最后反问环节的问题一定要准备一下，可以问一下游戏相关的，不打算再问新员工培训的问题了，不知道会不会让对方觉得我是到了公司才开始学习的。
5. 自己貌似深受“读书无用论”的茶毒，虽然没有表达过，但潜意识里还是认为学校学的东西在工作中用到的概率很少，但现在我连扇自己十个大耳刮子道歉，经典和基础的知识是以后拓展技术栈的基础，十分重要，奉劝类似的同学早点摒弃这种想法，打牢基础，至少也能让你在面试中脱颖而出。
6. 现实很残酷，没必要玻璃心，没过就是没过，重振旗鼓然后再去学习吧~

- 腾讯光子技术中心

1. 渲染管线
2. c++堆内存和栈内存的区别
3. 虚函数表指针在什么位置（对象头部有4个字节，存储虚函数表的位置）

- 网易雷火

总结一下面试问到的问题，主要是一面的内容，因为二面一直在问简历上的项目，数模IBM这些很久远的东西，都有点记不清了，好在是自己实打实做过的东西，所以还是能回忆起来一部分内容的。Hr面就是闲聊，没啥好记录的。

- dynamic\_cast和static\_cast的区别
- c++ new() 失败了会怎么办
- 说几种Unity中的MonoBehaviour
- 渲染管线分别对应Unity中的部分哪些
- UGUI和IMGUI的区别
- 虚拟内存
- 智能指针
- 回去写写多线程程序

- 疯狂游戏提前批

笔试考的大部分是一些基本的内容，有几道题印象比较深刻：

1. 说出一个序列，以第一个数为轴，执行一次快排之后的样子；
2. 给定一个ip地址，要求划分4个子网，问子网掩码和各个子网的ip范围
3. 向量点乘和叉乘的区别
4. 位运算：

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

int main()
{
```



```
int a = 998;
int i = 0;
while(a)
{
    i ++;
    a = a&(a-1);
}
cout<<i;
return 0;
}
```

就是求这个数中1的个数。

5. 算法题是一道动规，没做出来。题目是给定n个数字，求它们之间的大小关系一共有多少种。

面试：

1. 说说你知道的排序算法
2. 对一个基本有序的数组排序，用哪种算法比较合适，时间复杂度呢（插入排序比较好）
3. 有n个文件，需要动态返回最近使用的k个文件，文件按文件名索引，时间用时间戳表示。