1. git添加/删除远程仓库及github相关：

```
git remote rm origin
git remote add orinin https://github.com/wanggaolang/Blog

echo "# note" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:wanggaolang/note.git
git push -u origin master
---

当github与本地master不一致（如github上多了个readme）时，
若让本地与远程相同：git pull
若让远程与本地相同：git push -f
```

2.内存操作的小技巧

*(int *)ptr的意思是从ptr这个地址开始向上（因为是小端存储）取四个字节出来看成int，注意编译器优化会使两个相邻变量上下字节间发生变化，可以加上`volatile`。大部分机器都是小端存储，变量的首地址是最低的一个字节的地址，取变量时向上取，存储时向低地址存（因为是栈）。

3. sqrt是开平方，pow(x,n)是N次方

4. 重载<<运算符示例

```
ostream & operator<<( ostream & os,const Vector2D & c) //二维向量
{
    os << "x: "<<c.x <<" y: "<<c.y;
    return os;
}
```

5. new和malloc的区别

1. new可以自动计算所需要大小；malloc则必须要由我们计算字节数。
2. new操作符内存分配成功时，返回的是对象类型的指针；malloc内存分配成功则是返回void * ，需要通过强制类型转换将void*指针转换成我们需要的类型。

3. new内存分配失败时，会抛出bac_alloc异常；malloc分配内存失败时返回NULL。

4. malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。

5. 使用new操作符来分配对象内存时会经历三个步骤：

    - 第一步：调用operator new 函数（对于数组是operator new[]）分配一块足够大的，原始的，未命名的内存空间以便存储特定类型的对象。
    - 第二步：编译器运行相应的构造函数以构造对象，并为其传入初值。
    - 第三部：对象构造完成后，返回一个指向该对象的指针。

    使用delete操作符来释放对象内存时会经历两个步骤：

    - 第一步：调用对象的析构函数。
    - 第二步：编译器调用operator delete(或operator delete[])函数释放内存空间。

    总之来说，new/delete会调用对象的构造函数/析构函数以完成对象的构造/析构；而malloc 只管分配内存，并不能对所得的内存进行初始化

    6.使用malloc分配的内存后，如果在使用过程中发现内存不足，可以使用realloc函数进行内存重新分配实现内存的扩充。

```
void *realloc(void *ptr, size_t size)
    realloc先判断当前的指针所指内存是否有足够的连续空间，如果有，原地扩大可分配的内存地址，并且返回原来的地址指针；如果空间不够，先按照新指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来的内存区域。
    --ptr    指针指向一个要重新分配内存的内存块，该内存块之前是通过调用 malloc、calloc 或 realloc 进行分配内存的。如果为空指针，则会分配一个新的内存块，且函数返回一个指向它的指针。
    --size   内存块的新的大小，以字节为单位。如果大小为 0，且 ptr 指向一个已存在的内存块，则 ptr 所指向的内存块会被释放，并返回一个空指针。
    void *calloc(size_t nitems, size_t size)
    也是申请内存，nitems为元素个数，size为元素大小。与malloc的区别是这个会初始化为0。
```

6. 常见c语言函数

```
    void *memcpy(void *dest, const void *src, size_t n)
从 src 复制 n 个字符到 dest，不会先清空dest。
    void *memset(void *str, int c, size_t n)
复制字符 c（一个无符号字符）到参数 str 所指向的字符串的前 n 个字符。
    char *strcat(char *dest, const char *src)
把 src 所指向的字符串追加到 dest 所指向的字符串的结尾。
    char *strcpy(char *dest, const char *src)
把 src 所指向的字符串复制到 dest，会先清空dest。
```

7.红黑树和AVL树

红黑树不追求"完全平衡"，即不像AVL那样要求节点的 `|balFact| <= 1`，它只要求部分达到平衡，但是提出了为节点增加颜色，红黑是用非严格的平衡来换取增删节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。

就插入节点导致树失衡的情况，RB-Tree最多两次树旋转来实现复衡rebalance，旋转的量级是O(1)，而AVL的插入和删除节点导致失衡，AVL需要维护从被删除/插入节点到根节点root这条路径上所有节点的平衡，旋转的量级为O(logN)，而RB-Tree最多只需要旋转3次实现复衡，只需O(1)，所以说RB-Tree删除节点的rebalance的效率更高，开销更小！在查找时候AVL更快，但快的有限。

8.同步异步套接字

使用套接字进行数据处理有两种基本模式：同步和异步。

同步模式：同步模式的特点是在通过Socket进行连接、接收、发送数据时，客户机和服务器在接收到对方响应前会出于阻塞状态，即一直等到收到对方请求进才继续执行下面的语句。可见，同步模式只适用于数据处理不太多的场合。当程序执行的任务很多时，长时间的等待可能会让用户无法忍受。

异步模式：异步模式的特点是在通过Socket进行连接、接收、发送操作时，客户机或服务器不会处于阻塞方式，而是利用callback机制进行连接、接收、发送处理，这样就可以在调用发送或接收的方法后直接返回，并继续执行下面的程序。可见，异步套接字特别适用于进行大量数据处理的场合。

使用同步套接字进行编程比较简单，而异步套接字编程则比较复杂

9.GMP库

用于大数运算的c/c++库，在linux下完美支持，windows需要用mingw和msys进行编译，或者用gmp的windows版本mpir，原生支持vs上编译

10.clion快捷键设置

Setting -- Keymap

光标到上一个光标：搜索back

光标到下一个光标：搜索forward

11.进程，线程，协程

进程：计算机程序运行的实体。每个进程都有自己的独立内存空间，上下文切换开销比较大（栈，寄存器，虚拟内存，文件句柄等），相对稳定安全

线程：进程的一个实体，是cpu调度和分派的基本单位。线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。

协程：

12.volatile

volatile关键词影响编译器编译的结果，用volatile声明的变量表示该变量随时可能发生变化，与该变量有关的运算，不再编译优化，以免出错。

13.小知识（一）

1. ***ptr--** 会先减再运行*
2. 类外定义成员函数不能加上默认参数，如：`Test fun(int a = 1)`会报错，同样static声明的成员在外部定义时候，必须省去static。同时，static成员变量只有跟了const才可以在类里面的初始化列表中进行初始化，其余的都要在类的外部初始化。
3. string.find()和map.find()以及set.find()如果找不到目标，则结果为x.end()

14. c++ string的实习

```
#include <cstddef>
#include <iosfwd>
#include <iostream>
#include <string.h>
using namespace std;
class String {
private:
    /* data */
    char *data;    //字符串
    size_t length; //长度

public:
    String(const char *str = nullptr); //默认构造函数
    String(const String &str);         //拷贝构造函数
    friend istream &operator>>(istream &is, String &str);
    friend ostream &operator<<(ostream &os, String &str);

    String operator+(const String &str) const; //重载+
    String &operator=(const String &str);      //重载=
    String &operator+=(const String &str);     //重载+=
    bool operator==(const String &str) const;  //重载==
    char &operator[](int n) const;             //重载[]

    size_t size() const;       //获取长度
    const char *c_str() const; //获取C字符串

    ~String();
};

String::String(const char *str) { //通用构造函数
    if (!str) {
        length = 0;
```

```
        data = new char[1];
        *data = '\0';
    } else {
        length = strlen(str);
        data = new char[length + 1];
        strcpy(data, str);
    }
}
String::String(const String &str) { //拷贝构造函数
    length = str.size();
    data = new char[length + 1];
    const char *temp = str.c_str();
    strcpy(data, temp);
}

String::~String() {
    delete[] data;
    length = 0;
}

String String::operator+(const String &str) const //重载+
{
    String newString;
    newString.length = length + str.size();
    newString.data = new char[newString.length + 1];
    strcpy(newString.data, data);
    strcat(newString.data, str.data);
    return newString;
}

String &String::operator=(const String &str) //重载+
{
    if (this == &str) {
        return *this;
    }
    delete[] data;
    length = str.size();
    data = new char[length];
    strcpy(data, str.c_str());
    return *this;
}

String &String::operator+=(const String &str) //重载+
{
    length += str.size();
    char *newData = new char[length + 1];
    strcpy(newData, data);
    strcat(newData, str.c_str());
    delete[] data;
    data = newData;
    return *this;
}
inline bool String::operator==(const String &str) const //重载==
```

```
{
    if (length != str.size())
        return false;
    return strcmp(data, str.data) ? false : true;
}

inline char &String::operator[](int n) const //重载[]
{
    if (n >= length) {
        return data[length - 1]; //错误处理
    }
    return data[n];
}

inline size_t String::size() const //获取长度
{
    return length;
}
inline auto String::c_str( ) const ->const char * //获取C字符串
{
    return data;
}

istream &operator>>(istream &is, String &str) //输入
{
    char tem[1000]; //简单的申请一块内存
    is >> tem;
    str.length = strlen(tem);
    str.data = new char[str.length + 1];
    strcpy(str.data, tem);
    return is;
}

ostream &operator<<(ostream &os, String &str) //输出
{
    os << str.data;
    return os;
}
int main()
{
    String test("abc");
    cout<<test<<endl;
}
```

15.面向对象三大特性：封装、继承和多态

16.socket

假如b进程是异常终止的，发送FIN包是OS代劳的，b进程已经不复存在，**当机器再次收到该socket的消息时，会回应RST（因为拥有该socket的进程已经终止）**。a进程对收到RST的socket调用write时，操作系统会给a进程发送SIGPIPE，默认处理动作是终止进程。即：

> It is okay to write to a socket that has received a FIN, but it is an error to write to a socket that has received an RST

17.浮点数大小 //EQ

float：32位 1位符号位，8位指数位，23位尾数

double：64位 1位符号位，11位指数位，52位尾数

18.TCP长连接

长连接：client向server发起连接，server接受client连接，双方建立连接。Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

首先说一下TCP/IP详解上讲到的TCP保活功能，保活功能主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将应远等待客户端的数据，保活功能就是试图在服务器端检测到这种半开放的连接。

如果一个给定的连接在**两小时**内没有任何的动作，则服务器就向客户发一个探测报文段，客户主机必须处于以下4个状态之一：

1. 客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。
2. 客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在**75秒**后超时。服务器总共发送**10个**这样的探测 ，每个间隔**75秒**。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。
3. 客户主机崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。
4. 客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探查的响应。

从上面可以看出，TCP保活功能主要为探测长连接的存活状况，不过这里存在一个问题，存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。

在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

在应用层则可以用**心跳包**来进行保持长连接