粘包和半包

什么情况下会出现粘包和半包的问题呢？

TCP会出现粘包和半包的问题，UDP不会。

为什么TCP会出现粘包和半包的问题呢？

因为TCP是基于链接的可靠的流协议。而流消息是没有明确的边界的，所以无法判断哪一段流属于一个消息。

什么是粘包？

粘包：发送两条消息的时候，接收方一次性读取的两条消息，这种情况就叫粘包。（比如一条消息是ABC，一条消息是DEF，接收方读取到的却是ABCD）

什么是半包？

半包：发送一条消息的时候，接收方接收到的是两条消息，这种情况就叫半包。（比如一条消息是ABCD，接收方接收到的是一条ABC，一条D）

什么情况下会出现粘包？

一种情况是发送的消息的大小小于套接字缓冲区的大小。

一种情况是接收方没有及时接收。

什么情况下会出现半包？

一种情况是发送的消息的大小大于套接字缓冲区的大小。

一种情况是发送的消息的大小大于最大传输单元（MTU），必须进行拆包。

为什么要有缓冲区？

如果不使用缓冲区的话，每次写入操作CPU都会直接写到磁盘上，速度是很慢的。那如果先将写入的内容放在缓冲区，达到一定的大小之后，再一次性的写入磁盘上，这样的速度是会比较快的。总而言之，就是写入缓冲区的速度会比写入磁盘的速度快。

如何解决粘包和半包的问题？

有三种方法：

一种是：固定缓冲区的大小。服务端和客户端都使用相同的缓冲区大小，如果发送的数据不足缓冲区的大小的话，就使用空字符弥补到缓冲区的大小。（缺点是：当发送的数据较小的时候，会使用空字符进行弥补，增加了网络传输的负担。）

一种是：封装请求协议。将请求的数据封装成两部分，数据的大小+数据正文。读取的时候，先读取数据的大小，然后根据大小读取数据正文部分。（缺点是：编码成本较高。）

一种是：以特殊字符结尾，按行读取。

总而言之，上述的三种方法都是为了能够知道流的边界。

shared\_ptr智能指针

什么是shared\_ptr智能指针？

通过增加引用计数来共享同一块堆内存，当引用计数为0时，会自动释放该空间。

shared\_ptr智能指针的创建？

1. 空shared\_ptr智能指针的引用计数为0。
2. 可以使用 make\_shared进行初始化。
3. 使用拷贝构造函数和移动构造函数（左值用来赋值给别人初始化，就是拷贝构造函数）
4. 普通的指针不能为多个shared\_ptr智能指针初始化，会发生异常。
5. 在初始化shared\_ptr智能指针的时候，可以自定义指针的释放规则。（shared\_ptr默认的释放不支持释放申请的动态数组，所以释放申请的动态数组的时候需要自定义释放规则）

#include <iostream>

#include <memory>

using namespace std;

void deleteInt(int \*p)

{

    delete []p;

}

int main()

{

    std::shared\_ptr<int> p;    //空智能指针，引用计数为0

    std::shared\_ptr<int> q(nullptr);    //空智能指针，引用计数为0

    std::shared\_ptr<int> a(new int(10));

    std::cout<<\*a<<std::endl;

    std::shared\_ptr<int> b = std::make\_shared<int>(10); //使用make\_shared进行初始化

    std::cout<<\*b<<std::endl;

    std::shared\_ptr<int> c(a); //a是左值，所以是拷贝构造函数

    std::cout<<\*c<<std::endl;

    std::shared\_ptr<int> d = std::move(a); //移动构造函数，将a的堆内存给了d，此时a是空智能指针

    //std::cout<<\*d<<\*a<<std::endl;

    int \*e = new int(20);    //普通的指针

    std::shared\_ptr<int> f(e);

    std::cout<<\*f<<std::endl;

    //std::shared\_ptr<int> g(e);

    //std::cout<<\*g<<std::endl;

    //自定义释放规则1:使用std::default\_delete

    std::shared\_ptr<int> h(new int[10], std::default\_delete<int[]>());

    //自定义释放规则2:使用自定义的函数

    std::shared\_ptr<int> i(new int[10], deleteInt);

    //自定义释放规则3:使用Lambda表达式

    std::shared\_ptr<int> j(new int[10], [](int \*k){delete []k;});

    return 0;

}

shared\_ptr智能指针的一些成员方法

1. bool()
2. unique()
3. use\_count()
4. get()
5. reset()
6. swap()

#include <iostream>

#include <memory>

using namespace std;

int main()

{

    std::shared\_ptr<int> p1(new int(10));

    //bool用来判断shared\_ptr是否是空智能指针，空指针返回0，非空指针返回1

    std::cout<<"p1是空指针吗："<<bool(p1)<<std::endl;

    std::shared\_ptr<int> p2 = std::move(p1);

    std::cout<<"p2是空指针吗："<<bool(p2)<<std::endl;

    std::cout<<"p1是空指针吗："<<bool(p1)<<std::endl;

    //unique用来判断p2指向的堆内存是否只有p2指向它而已，是返回1，不是返回0

    std::cout<<"p2指向的堆内存，是否只有p2指向它而已："<<p2.unique()<<std::endl;

    std::shared\_ptr<int> p3(p2);

    std::cout<<"p2指向的堆内存，是否只有p2指向它而已："<<p2.unique()<<std::endl;

    //use\_count用来返回当前有多少个指针指向一样的堆空间

    std::cout<<"当前有多少个指针指向和p2指向一样的堆空间："<<p2.use\_count()<<std::endl;

    int \*p4 = new int(20);

    std::cout<<"普通指针p4的地址："<<p4<<std::endl;

    std::shared\_ptr<int> p5(p4);

    //get用来返回智能指针指向普通指针的地址

    std::cout<<"获取p5包含的普通指针："<<p5.get()<<std::endl;

    std::shared\_ptr<int> p6(new int(30));

    std::cout<<"p6是空指针吗："<<bool(p6)<<std::endl;

    //reset当没有实参的时候，将当前对象置为空指针，当前对象所指的堆内存的引用计数减1

    p6.reset();

    std::cout<<"p6是空指针吗："<<bool(p6)<<std::endl;

    //reset当有实参的时候，当前对象获得实参的所有使用权，并且所指的堆内存的引用计数为1

    p6.reset(new int(40));

    std::cout<<"p6是空指针吗："<<bool(p6)<<std::endl;

    std::cout<<"p5是多少："<<\*p5<<"p6是多少："<<\*p6<<std::endl;

    std::shared\_ptr<int> p7(p5);

    std::cout<<"当前有多少个指针指向和p5指向一样的堆空间："<<p5.use\_count()<<"p7是多少："<<\*p7<<std::endl;

    //swap用来交换两个相同类型的shared\_ptr智能指针的内容，其它指向同一堆内存的智能指针的值也会跟着交换

    swap(\*p5, \*p6);

    std::cout<<"p5是多少："<<\*p5<<"p6是多少："<<\*p6<<std::endl;

    std::cout<<"当前有多少个指针指向和p5指向一样的堆空间："<<p5.use\_count()<<"p7是多少："<<\*p7<<std::endl;

    return 0;

}

weak\_ptr智能指针