

# 常识（从先前的笔记中节选出来的一些笔记）

2025年5月21日 16:43

》》》这里我会放置一些比较基础的概念，值得反复回顾的概念。  
这些笔记都来自我的笔记中。放在这个分区中，是为了能够高效率的回顾这些常见知识。

## 》》》什么是 QT 中的 signals 和 slots ?

### signals

signals 关键字是 Qt 中特有的，用于声明信号。  
信号是 Qt 的对象间通信机制之一，signals 通常与槽（slots）配合使用。信号用于在对象之间发送通知，信号和槽的连接由 Qt 的 QObject::connect() 函数处理。  
例如，当某个事件发生时，发出一个信号，其他对象可以响应该信号。

```
class MyClass : public QObject {
    Q_OBJECT // 必须包含此宏才能使用信号和槽

public:
    MyClass() : QObject() {}

signals:
    void valueChanged(int newValue); // 信号声明

public slots:
    void onValueChanged(int newValue) {
        qDebug() << "Value changed to" << newValue;
    }
};
```

在这个示例中，valueChanged 是一个信号，表示值已经变化。信号的发送通常通过 emit 关键字进行，如：

```
emit valueChanged(42);
```

### slots

slots 也是在类中定义的，不同于 signals 的是，slots 可以放在类的 public、protected 或 private 部分，具体取决于你想如何控制访问权限。

public slots:	如果槽是公共的，那么外部代码可以直接调用这些槽函数。这在很多情况下是需要的，因为槽函数可能是信号的响应函数。
private slots:	如果槽是私有的，那么它们只能在类内部被调用，外部代码无法直接调用。通常用于仅在内部响应某些信号的场景。
protected slots:	保护槽可以在类的派生类中访问。

```
class MyClass : public QObject {
    Q_OBJECT

public:
    MyClass() {}

signals:
    void valueChanged(int newValue);

public slots:
    void onValueChanged(int newValue) {
        qDebug() << "Value changed to" << newValue;
    }

private slots:
    void internalSlot() {
        // 仅在内部调用的槽
    }
};
```

### Signals 的作用域（是否存在修饰符？）

在 Qt 中，signals 不支持像 slots 那样使用 public或 private 进行前向修饰，因为信号的作用是：让类的外部对象能够触发它们，并与某个对象进行通信。  
因此 signals 在类中默认是 public 的，如果信号被声明为私有或受保护的，外部对象就无法连接到它，这违背了信号与槽机制的设计目的。

```
class MyClass : public QObject {
    Q_OBJECT

public:
    MyClass() {}

signals: // 这里的信号默认是 public 的，不需要显式指定
    void valueChanged(int newValue); // 信号是 public 的，外部可以连接和触发

private:
    void privateMethod() {}
};
```

在这个示例中，valueChanged 信号默认是 public 的，你不需要显示地使用 public signals。Qt 也不允许将其声明为 private。

slots 和 signals 的区别：

signals	通常是 public 的，用于与外部对象通信。信号不会有访问限制，因此能够被外部对象通过 connect() 连接和触发。
slots:	可以是 public、protected 或 private，这取决于设计者是否希望在外部分访问，或只在类内部调用。

》》》关于 QT 中的信号和槽机制的启用。QT 中 signals slots 函数的要求。QT 中的 connect() 的解释。  
》》》111111  
》》》Q\_OBJECT 的定义及其作用

定义	<pre>153  /* qmake ignore Q_OBJECT */ 154  #define Q_OBJECT \ 155  public: \ 156      QT_WARNING_PUSH \ 157      Q_OBJECT_NO_OVERRIDE_WARNING \ 158      static const QMetaObject staticMetaObject; \ 159      virtual const QMetaObject *metaObject() const; \ 160      virtual void *qt_metacast(const char *); \ 161      virtual int qt_metacall(QMetaObject::Call, int, void **); \ 162      QT_TR_FUNCTIONS \ 163  private: \ 164      Q_OBJECT_NO_ATTRIBUTES_WARNING \ 165      Q_DECL_HIDDEN_STATIC_METACALL static void qt_static_metacall(QObject *, QMetaObject::Call, int, void **); \ 166      QT_WARNING_POP \ 167      struct QPrivateSignal {}; \ 168      QT_ANNOTATE_CLASS(qt_qobject, "") 169</pre>
作用	<ul style="list-style-type: none"><li>支持信号与槽的机制。</li><li>支持运行时元信息、反射。</li><li>支持动态属性和事件处理。</li><li>允许 moc 自动生成代码，支持自动化的信号和槽连接。</li></ul>
解释	<p>1. 信号和槽机制是否可用：</p> <ul style="list-style-type: none"><li>定义了 Q_OBJECT： 你可以使用 Qt 的信号和槽机制。（信号可以被发射，槽可以被调用，且可以通过 QObject::connect() 方法将信号和槽连接在一起。） 信号和槽机制支持跨线程调用。（Qt 会自动处理线程间的信号和槽连接，确保在正确的线程中调用槽。）</li><li>没有定义 Q_OBJECT： 信号和槽机制无法使用。即使你在类中定义了 signals 和 public slots，Qt 也不会为它们生成相关的代码。 (这意味着你无法通过 connect() 来连接信号和槽，或者使用 emit 来发射信号。) 信号和槽无法跨线程使用，或者可能无法在不同的线程中正确工作。</li></ul> <p>2. moc 生成的代码：</p> <ul style="list-style-type: none"><li>定义了 Q_OBJECT： Qt 的元对象编译器 (moc) 会自动为你生成与信号、槽和元信息相关的代码，并使其能够正确地参与 Qt 的信号和槽机制。 在运行时，Qt 可以动态地查询类的元信息 and 处理信号槽的连接。（例如，信号的实现和槽的连接都由 moc 生成，metaObject() 方法可以用于动态查询类的元信息。）</li><li>没有定义 Q_OBJECT： moc 不会生成这些代码，你的类不会有与信号和槽相关的支持，也无法查询元信息。 (你无法在运行时通过反射访问类的信号、槽或属性，也不能使用 Qt 提供的信号和槽连接机制。)</li></ul> <p>3. 动态属性和反射功能：</p> <ul style="list-style-type: none"><li>定义了 Q_OBJECT： 你可以使用 Qt 的动态属性系统，比如通过 setProperty() 和 property() 来设置和获取对象的属性。此外，可以使用 QMetaObject 来查询类的元信息，如类名、信号、槽等。</li><li>没有定义 Q_OBJECT： 你的类将没有这些功能，因为 moc 生成的代码包含了必要的元信息和属性处理机制。</li></ul>

没有 Q\_OBJECT 时的后果:

如果你在类中没有定义 Q\_OBJECT, 并尝试使用信号和槽, 程序将编译失败。你可能看到类似以下的错误:

“signal is not a member of class” 或 “no matching function for call to connect” 这些错误会提示你在类中没有启用信号和槽的功能。

》》！！！！》》将一个类定义为: 继承自 QObject 类 和 在类中声明宏定义

Q\_OBJECT, 这两个步骤必须搭配使用吗? 是否可以单独使用? 如果可以单独使用, 两者有什么不同?

图示:

**是否必须搭配使用?** 是的, 这两者必须搭配使用, 缺一不可。如果只使用其中之一, 类的信号与槽机制或其他 Qt 功能将无法正常工作。

**两者的功能:** 继承 QObject 类 -> 让类成为 Qt 对象, 具备 Qt 的基础功能。  
声明 Q\_OBJECT 宏 -> 启用元对象系统, 使得类能够支持信号与槽、反射机制等高级特性。

》》！！！！》》Qt 中的信号函数一般在什么时候发出信号? 怎样发出信号? 槽函数通过怎样的手段得知信号已经被发出? 并接受该信号?

signal

如何发送信号:	在 Qt 中, 信号函数本身并不直接发出信号, 信号的发出是通过调用 emit 关键字实现的。
何时发送信号:	我们必须在合适的地方使用 emit 来发出信号。(它通常是在特定条件满足时, 由用户选择使用 emit() 发出信号。)
示例:	<pre>void MyClass::someFunction() {     If(...)     {         // .....         emit mySignal(); // 发出信号     } }</pre>

slot

如何接受信号:	你需要通过 QObject::connect() 函数将信号与槽关联起来, 连接后, 信号触发时会调用对应的槽函数。
示例:	<pre>MyClass obj; connect(&amp;obj, &amp;MyClass::someFunction, &amp;someObject, &amp;SomeClass::someSlot);</pre>

》》》2222222222

》》》Qt 中槽函数和信号函数的参数需要匹配, 以下是其规则:

1. 参数类型匹配	信号和槽的参数类型必须匹配, 即信号发射时传递的参数类型必须与槽函数中定义参数类型一致。 (括引用传递: 如果信号发出的是某种类型的引用 (如 QString&), 那么槽函数也必须接受同样类型的引用。)
2. 参数个数匹配	信号和槽的参数个数必须匹配。即信号定义参数数量和槽函数定义参数数量必须相等。
3. 默认参数	如果槽函数的参数有默认值, 信号发射时可以不传递这些参数, 但如果槽函数没有默认值, 则必须在连接时传递对应的参数。
4. 如果信号发出的参数与槽的接收参数不同:	Qt 会尝试进行类型转换。例如, 如果信号发出的参数类型是 int, 但槽函数的参数是 double, Qt 会自动进行转换, 但这仅在 Qt 支持的类型转换之间有效。 (如果类型不兼容, Qt 会报错, 且信号与槽无法连接。)

》》》3333333333

》》》QObject::connect 函数的参数, 示意, 重载, 使用方法?

1. QObject::connect 函数的基本语法:	<b>函数签名:</b> QObject::connect(sender, signal, receiver, slot); <ul style="list-style-type: none"><li>• sender: 发射信号的对象 (通常是一个继承自 QObject 的类的实例)。</li><li>• signal: 信号的名称。需要使用 Qt 的信号和槽机制进行声明, 通常是 SIGNAL() 宏包裹的信号名称。</li><li>• receiver: 接收信号的对象 (通常是一个继承自 QObject 的类的实例)。</li><li>• slot: 槽的名称, 通常是 SLOT() 宏包裹的槽函数名称。</li></ul>						
2. QObject::connect 函数的重载: Qt 中的 QObject::connect 有多个重载版本, 主要根据信号和槽的参数传递方式有所不同。以下是一些常见的重载版本:	<table><tr><td>2.1 基本版本 (旧版信号和槽机制):</td><td>这种连接方式是 Qt 经典的信号和槽机制 (老版本)。</td></tr><tr><td><b>函数签名:</b></td><td>QObject::connect(sender, SIGNAL(signal_name(ArgType)), receiver, SLOT(slot_name(ArgType)));</td></tr><tr><td><b>函数参数:</b></td><td><ul style="list-style-type: none"><li>• sender: 信号发送方 (QObject)。</li><li>• signal_name: 信号名称, 必须是 SIGNAL() 宏的形式。</li><li>• receiver: 信号接收方 (QObject)。</li><li>• slot_name: 槽函数名称, 必须是 SLOT() 宏的形式。</li></ul></td></tr></table>	2.1 基本版本 (旧版信号和槽机制):	这种连接方式是 Qt 经典的信号和槽机制 (老版本)。	<b>函数签名:</b>	QObject::connect(sender, SIGNAL(signal_name(ArgType)), receiver, SLOT(slot_name(ArgType)));	<b>函数参数:</b>	<ul style="list-style-type: none"><li>• sender: 信号发送方 (QObject)。</li><li>• signal_name: 信号名称, 必须是 SIGNAL() 宏的形式。</li><li>• receiver: 信号接收方 (QObject)。</li><li>• slot_name: 槽函数名称, 必须是 SLOT() 宏的形式。</li></ul>
2.1 基本版本 (旧版信号和槽机制):	这种连接方式是 Qt 经典的信号和槽机制 (老版本)。						
<b>函数签名:</b>	QObject::connect(sender, SIGNAL(signal_name(ArgType)), receiver, SLOT(slot_name(ArgType)));						
<b>函数参数:</b>	<ul style="list-style-type: none"><li>• sender: 信号发送方 (QObject)。</li><li>• signal_name: 信号名称, 必须是 SIGNAL() 宏的形式。</li><li>• receiver: 信号接收方 (QObject)。</li><li>• slot_name: 槽函数名称, 必须是 SLOT() 宏的形式。</li></ul>						

2.2 <u>新版本（基于函数指针的连接）</u> ：	Qt 5 引入了基于函数指针的新版本信号和槽连接，这种方式比传统的宏方式更安全，类型检查更严格，避免了运行时错误。 在新版本中，信号和槽的连接是类型安全的，编译时会检查信号和槽参数是否匹配。
函数签名：	QObject::connect(sender, &Sender::signal_name, receiver, &Receiver::slot_name);
函数参数：	<ul style="list-style-type: none"><li>• sender：信号发送方 <u>(QObject)</u>。</li><li>• signal_name：信号名称，<u>是成员函数指针</u>。</li><li>• receiver：信号接收方 <u>(QObject)</u>。</li><li>• slot_name：槽函数名称，<u>是成员函数指针</u>。</li></ul>
2.3 <u>重载版本（支持 lambda 函数）</u> ：	这种方式允许使用 lambda 函数来作为槽，具有更高的灵活性和简洁性。
函数签名：	QObject::connect(sender, &Sender::signal_name, [=](ArgType arg){ // lambda 表达式中处理信号 });
函数参数：	<ol style="list-style-type: none"><li>1. sender：指向信号发送者对象的指针。它是发出信号的 QObject 类的实例。</li><li>2. &amp;Sender::signal_name（信号） 这是发送者类（Sender）中定义的信号。信号是发送者对象通过某些事件或状态变化发出的通知，通常是在 Sender 类内部通过 signals 关键字定义的。例如，可能是 clicked、valueChanged 等信号。</li><li>3. [=](ArgType arg) (lambda 表达式)：作为槽函数来处理信号。当信号发出时，lambda 表达式会被调用。</li></ol>
2.4 <u>连接类型（传递方式）</u> ：	Qt 还提供了几种 connect 的重载形式来指定信号和槽的调用方式：
函数签名：	QObject::connect(sender, &Sender::signal_name, receiver, &Receiver::slot_name, <u>Qt::ConnectionType type</u> );
函数参数：	<ul style="list-style-type: none"><li>• Qt::AutoConnection（默认）：根据信号和槽的执行线程自动选择连接方式。</li><li>• Qt::DirectConnection：直接在发出信号的线程中调用槽。</li><li>• Qt::QueuedConnection：将槽调用放入接收者线程的事件队列中。</li><li>• Qt::BlockingQueuedConnection：与 QueuedConnection 类似，但会阻塞直到槽函数执行完成。</li></ul>

》》》》什么是正则表达式？正则表达式的规则是什么？

》》》》正则表达式

定义：

正则表达式（Regular Expression，简称 regex 或 regexp）是一种用于匹配字符串的模式。它由一些字符组成，利用这些字符可以定义复杂的匹配规则。并可以用来检查、查找、替换或操作字符串中的文本。

正则表达式的规则表：

普通字符：	字母、数字和标点符号等普通字符代表它们自己，如 a、1、# 等。	
特殊字符：	..：	匹配除换行符以外的任何单个字符。
	^：	匹配输入字符串的开始位置。
	\$：	匹配输入字符串的结束位置。
	*	匹配前一个字符零次或多次。
	+	匹配前一个字符一次或多次。
	?	匹配前一个字符零次或一次。
	[]：	字符集，匹配括号内的任何字符，例如 [abc] 匹配 a、b 或 c。
	：	表示“或”操作符，例如 a b 匹配 a 或 b。
转义字符：	注意： 1 -> "." 是一个有意义的特殊字符，如果需要匹配"."，则需要对齐进行转移。 2 -> "**", "+", "?" 指的是匹配前一个字符一次或者多次，而不是前面所有字符一次或多次。 (比如((\w+)(\.\.))?，这里的 "?" 仅作用于(\.)，即仅作用于"." 二者的匹配。)	
	\：用来转义特殊字符，使其失去特殊含义，或者用于表示一些特殊字符，如：	
	\d：	匹配一个数字，等价于 [0-9]。
	\w：	匹配一个字母、数字或下划线，等价于 [A-Za-z0-9_]。
量词：	{n}：	匹配前一个字符恰好出现 n 次，例如 a{3} 匹配 aaa。

	<table><tr><td>{n}:</td><td>匹配前一个字符至少出现 n 次，例如 a{2,} 匹配 aa、aaa、aaaa 等。</td></tr><tr><td>{n,m}:</td><td>匹配前一个字符出现 n 到 m 次，例如 a{2,4} 匹配 aa、aaa 或 aaaa。</td></tr></table>	{n}:	匹配前一个字符至少出现 n 次，例如 a{2,} 匹配 aa、aaa、aaaa 等。	{n,m}:	匹配前一个字符出现 n 到 m 次，例如 a{2,4} 匹配 aa、aaa 或 aaaa。
{n}:	匹配前一个字符至少出现 n 次，例如 a{2,} 匹配 aa、aaa、aaaa 等。				
{n,m}:	匹配前一个字符出现 n 到 m 次，例如 a{2,4} 匹配 aa、aaa 或 aaaa。				
分组与捕获:	<table><tr><td>():</td><td>用于分组，可以将多个字符组合成一个单元，进行整体匹配。分组还可以用于捕获匹配的内容，例如 (abc) 匹配 abc，并且可以获取匹配到的字符串。</td></tr></table>	():	用于分组，可以将多个字符组合成一个单元，进行整体匹配。分组还可以用于捕获匹配的内容，例如 (abc) 匹配 abc，并且可以获取匹配到的字符串。		
():	用于分组，可以将多个字符组合成一个单元，进行整体匹配。分组还可以用于捕获匹配的内容，例如 (abc) 匹配 abc，并且可以获取匹配到的字符串。				

比如用于匹配邮箱的正则表达式: ((\w+)(\.|\_)?(\w+))@(\w+)(\.|\_)(\w+))+)



》》对于末尾的理解

((\w+)(\.|\_)?(\w+))@(\w+)(\.|\_)(\w+))+)

这里的 “+” 指的是对 (\w+)) 需要匹配多次，比如对于这个邮箱：  
user@mail.example.co.uk，我们需要对 (\w+)) 匹配多次，因为我们会多次获取 example, co, uk

》》》》什么是全双工？半双工？单工？（关于设备能否进行传播和接收、传播或接收能否同时发生的问题）

1. 全双工 (Full-Duplex)	全双工通信指的是在同一时间内，通信双方可以同时进行双向数据传输。也就是说，双方可以在同一时间既发送又接收数据。
特点:	<ul style="list-style-type: none"><li>同时发送和接收数据。</li><li>双向通信不互相干扰。</li><li>需要独立的信号通道来实现同时的发送和接收。</li></ul>
例子:	<ul style="list-style-type: none"><li>电话通信：在电话通话过程中，双方可以同时说话和听到对方的声音。</li><li>现代计算机网络：例如以太网，在全双工模式下，数据可以同时发送和接收方向上传输。</li><li>无线通信（如Wi-Fi）：现代的无线设备也支持全双工通信。</li></ul>

2. 半双工 (Half-Duplex)	半双工通信指的是通信双方在同一时间内只能单向传输数据。在一个时间段内，数据只能在一个方向上传输，另一方只能接收数据，无法同时发送。 要实现双向通信，设备需要切换方向。
特点:	<ul style="list-style-type: none"><li>在任意时刻只能发送或接收数据，不能同时进行。</li><li>数据流是单向的，传输方向可以改变，但不能同时改变。</li></ul>
例子:	<ul style="list-style-type: none"><li>对讲机：对讲机是典型的半双工通信设备，一个用户按下按钮讲话时，另一方只能接收，直到用户松开按钮才能接收或发送。</li><li>无线电通信：在许多无线电系统中，广播和接收这两个操作只能是交替进行的。</li></ul>

3. 单工 (Simplex)	单工通信是一种通信模式，在这种模式下，数据只能沿着一个方向传输，即只能从发送方到接收方，不允许反向传输。 可以将其视为全双工和半双工的“极端”情况。
特点:	<ul style="list-style-type: none"><li>数据只能单向传输，接收方无法发送任何反馈。</li><li>通常只用于一些简单的数据传输场景。</li></ul>
例子:	<ul style="list-style-type: none"><li>电视广播：电视台只会将信号广播到所有观众，观众无法将</li></ul>

	信号发送回广播站。
	◦收音机：收音机只能接收广播电台的信号，无法反向传输数据。

》》》》什么是短连接，什么是长连接？

1. 短连接（Short Connection）

特点	<ul style="list-style-type: none"><li>•每次通信都建立新连接，完成后立即关闭。</li><li>•适用于 请求-响应模式（如 HTTP/1.0）。</li><li>•每次请求都需要 三次握手（建立连接）和四次挥手（关闭连接）。</li></ul>
工作流程	<ol style="list-style-type: none"><li>1.客户端 发起 TCP 连接（三次握手）。</li><li>2.客户端 发送请求，服务器 返回响应。</li><li>3.连接立即关闭（四次挥手）。</li><li>4.下次请求时，重新建立连接。</li></ol>
优点	<ul style="list-style-type: none"><li>•实现简单，服务器不需要维护连接状态。</li><li>•适合低频请求（如网页浏览、传统 API 调用）。</li></ul>
缺点	<ul style="list-style-type: none"><li>•频繁建立/关闭连接，性能开销大（三次握手、四次挥手耗时）。</li><li>•高并发时服务器压力大（每个请求都要新建连接）。</li></ul>
典型应用	<ul style="list-style-type: none"><li>•HTTP/1.0（默认短连接）。</li><li>•简单的 REST API 请求。</li></ul>

2. 长连接（Long Connection）

特点	<ul style="list-style-type: none"><li>•一次连接，多次通信，完成后不会立即关闭。</li><li>•适用于 持续交互（如 WebSocket、数据库连接）。</li><li>•减少握手开销，提高性能。</li></ul>
工作流程	<ol style="list-style-type: none"><li>1.客户端 发起 TCP 连接（三次握手）。</li><li>2.客户端 和 服务器 可以 多次交换数据。</li><li>3.连接保持，直到超时或主动关闭。</li><li>4.下次请求 复用同一个连接。</li></ol>
优点	<ul style="list-style-type: none"><li>•减少 TCP 握手/挥手开销，提高效率。</li><li>•适合高频请求（如实时通信、游戏、数据库访问）。</li><li>•降低服务器负载（减少连接数）。</li></ul>
缺点	<ul style="list-style-type: none"><li>•服务器需要维护连接状态（可能占用更多内存）。</li><li>•需要心跳机制（防止连接被误杀）。</li></ul>
典型应用	<ul style="list-style-type: none"><li>•HTTP/1.1（Keep-Alive）（默认复用连接）。</li><li>•WebSocket（全双工长连接）。</li><li>•MySQL/Redis 数据库连接池。</li><li>•实时通信（如聊天、直播）。</li></ul>

》》》》HTTP，UDP，TCP等等协议的区别？包括应用层面和形式上的不同：包括协议的格式等等，这些格式的设置导致了什么，有什么好处？

1. 协议层级与功能定位

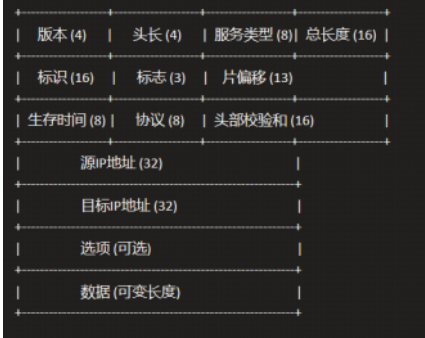
根据 TCP/IP 四层模型，这些协议分布在不同的层级，承担不同的职责：


层级	协议	核心功能
应用层	HTTP	定义应用程序间的数据交互规则（如网页请求/响应格式）。
传输层	TCP / UDP	提供端到端的数据传输服务（可靠性、流量控制、复用等）。
网络层	IP	实现主机到主机的数据包路由和寻址（基于IP地址）。
链路层	以太网 / Wi-Fi	负责物理介质上的数据传输（如MAC地址寻址、帧封装）。

2. 协议格式对比

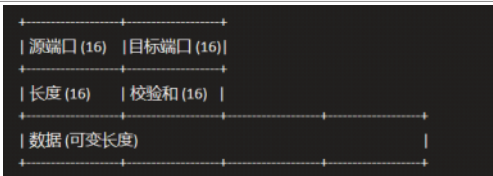
(1) IP 协议（网络层）	(2) TCP 协议（传输层）
----------------	-----------------



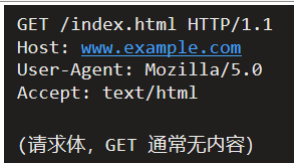
格式:	
关键字段:	<ul style="list-style-type: none"><li>• 协议字段: 标识上层协议 (如 6 表示 TCP, 17 表示 UDP)。</li><li>• 生存时间 (TTL): 防止数据包无限循环, 每经过一个路由器减 1, 归零则丢弃。</li><li>• 片偏移: 支持大数据包分片传输, 由接收端重组。</li></ul>
设计优势:	<ul style="list-style-type: none"><li>• 分片重组: 允许大数据包适应不同网络的最大传输单元 (MTU)。</li><li>• 全局寻址: 通过 IP 地址实现跨网络的端到端路由。</li></ul>

格式:	
关键字段:	<ul style="list-style-type: none"><li>• 序列号/确认号: 实现可靠传输 (通过确认机制和重传)。</li><li>• 控制位: 如 SYN (建立连接)、ACK (确认)、FIN (终止连接)。</li><li>• 窗口大小: 流量控制, 避免发送方淹没接收方。</li></ul>
设计优势:	<ul style="list-style-type: none"><li>• 可靠性: 通过三次握手、数据确认、超时重传确保数据完整有序。</li><li>• 拥塞控制: 动态调整发送速率, 避免网络拥塞 (如慢启动、拥塞避免)。</li></ul>
	<ul style="list-style-type: none"><li>• <b>传输层协议:</b> TCP 是传输层协议, 用于确保在两个端点之间可靠的数据传输。</li><li>• <b>面向连接:</b> TCP 是面向连接的协议, 通信前需要先建立连接 (三次握手), 通信结束后需要关闭连接 (四次挥手)。</li><li>• <b>可靠性:</b> TCP 保证数据的可靠性, 包括数据的顺序、完整性和无丢失传输。使用了流量控制、拥塞控制和重传机制。</li></ul>

(3) UDP 协议 (传输层)

格式:	
关键字段:	<ul style="list-style-type: none"><li>• 无序列号/确认号: 不保证数据可靠到达。</li><li>• 校验和可选: 部分场景可关闭以提升性能。</li></ul>
设计优势:	<ul style="list-style-type: none"><li>• 低延迟: 无连接、无握手, 适合实时应用 (如视频通话、游戏)。</li><li>• 轻量级: 头部开销小 (仅 8 字节), 传输效率高。</li></ul>
1	<ul style="list-style-type: none"><li>• <b>传输层协议:</b> UDP 是一个面向数据报的协议, 属于传输层协议。</li><li>• <b>无连接:</b> UDP 是无连接的, 不需要建立连接即可发送数据。</li><li>• <b>不可靠:</b> UDP 不保证数据包的送达、顺序和完整性, 数据包可能丢失、乱序或重复。</li></ul>

(4) HTTP 协议 (应用层)

格式 (HTTP/1.1 请求示例):	
关键特点:	<ul style="list-style-type: none"><li>• 文本协议: 人类可读, 但 HTTP/2 后改为二进制帧以提高效率。</li><li>• 无状态: 每次请求独立, 依赖 Cookie/Session 维持状态。</li><li>• 方法语义: GET (获取资源)、POST (提交数据)、PUT (更新资源) 等。</li></ul>
设计优势:	<ul style="list-style-type: none"><li>• 灵活性: 支持多种内容类型 (JSON、HTML、图片等)。</li><li>• 可扩展性: 通过头部字段 (如 Content-Type、Authorization) 增强功能。</li></ul>
1	<p><b>应用层协议:</b> HTTP 属于应用层协议, 是用于客户端 (通常是浏览器) 和服务器之间交换数据的协议。</p> <p><b>无连接:</b> HTTP 协议是无连接的, 客户端和服务端在请求响应过程中不保持连接, 每次请求都需要建立新的连接。</p> <p><b>请求-响应模式:</b> HTTP 通常采用请求-响应模式, 客户端发送请求, 服务器返回响应。</p> <p><b>面向文本的协议:</b> HTTP 消息通常是基于文本的, 包括请求头、请求体 (可选), 响应头、响应体 (可选)。</p>

3. 应用场景对比

协议	典型应用场景	选择原因
HTTP	网页浏览 (HTTPS)、REST API、文件下载	标准化、易调试、兼容性强。
TCP	文件传输 (FTP)、电子邮件 (SMTP)、远程登录 (SSH)	需要可靠传输和有序交付。
UDP	实时音视频 (Zoom、VoIP)、DNS查询、在线游戏 (低延迟)	容忍少量丢包, 追求传输速度。

IP	所有基于 IP 的网络通信（如 TCP/UDP/ICMP）	提供全局寻址和路由，是互联网
----	-------------------------------	----------------

》》》HTTP 请求的格式：

定义：	HTTP 请求的格式由 请求行（Request Line）、请求头（Headers）、空行（CRLF）和请求体（Body）四部分组成。
基本格式：	<请求行> <请求头> <空行> // 通过回车换行（CRLF，即`\\r\\n`）分隔头和体 <请求体> // 可选（如 POST 请求携带数据）
示例（GET 请求） 通常用于从服务器中获取数据	GET /api/user?id=123 HTTP/1.1  Host: example.com User-Agent: Mozilla/5.0 Accept: application/json Connection: keep-alive
示例（POST请求） 通常用于向服务器发送数据	POST /api/login HTTP/1.1  Host: example.com Content-Type: application/json Content-Length: 45  { "username": "alice", "password": "123456" }

细则：（对照上述请求的示例理解）

- 》》》处理 HTTP 请求时，必须进行的操作包括：
- 》》》处理 HTTP 请求时，必须进行的操作包括：
1. 设置 HTTP 响应状态码。
  2. 设置响应头部（如 Content-Type, Content-Length 等）。



- 3. 生成响应体（实际返回的内容）。
- 4. 处理错误和异常，返回适当的错误码和信息。
- 5. （视情况）处理 CORS。
- 6. （视情况）设置 Cookie。
- 7. 设置缓存控制。
- 8. （视情况）确保安全，如使用 HTTPS 和设置相关的安全头部。
- 9. 返回正确格式的内容。

》》》》POST 请求和 GET 请求

GET GET GET

GET请求：	GET请求是用来从服务器获取数据的请求。它是一个无副作用的请求，即不会修改服务器上的资源。 GET请求通常用于请求数据或者获取某些信息，并且请求参数通常会通过URL传递。
操作：	<ul style="list-style-type: none"><li>获取资源：通常用于请求页面内容、查询数据库中的数据、获取图片等静态资源。</li><li>传递参数：GET请求将请求的参数附加在URL后面，通常以?开头，多个参数之间用&amp;连接。例如：https://example.com/api?name=John&amp;id=123.</li></ul>
GET请求的特点：	<ol style="list-style-type: none"><li>参数通过URL传递，请求体为空。</li><li>请求内容可以被缓存。</li><li>浏览器可以书签GET请求，也就是说，GET请求的URL可以被保存并稍后再次访问。</li><li>请求参数有限制：由于GET请求的参数是附加在URL后的，所以URL长度有限制，通常为2048个字符左右。</li><li>无副作用：GET请求通常是只读取数据，不修改服务器上的任何资源。</li></ol>
客户端发起GET请求后，服务器一般的处理：	<ul style="list-style-type: none"><li>服务器接收到GET请求后，会解析URL中附带的参数。</li><li>然后根据请求的URL和参数，查找相应的资源或执行对应的查询操作。</li><li>最终，服务器将资源返回给客户端（如网页、图片、JSON数据等）。</li></ul>
处理流程简要：	<ol style="list-style-type: none"><li>客户端发送GET请求。</li><li>服务器解析请求并获取资源。</li><li>服务器返回请求的资源或响应数据。</li></ol>

POST POST POST

POST请求：	POST请求用于向服务器发送数据，通常用于提交表单数据或上传文件等操作。 与GET不同，POST请求会将请求数据包含在请求体中，而不是URL中。
操作：	<ul style="list-style-type: none"><li>提交数据：例如，提交用户的表单数据、登录请求、注册信息等。</li><li>修改服务器资源：POST请求会对服务器上的资源进行操作，可能会修改数据、存储数据、上传文件等。</li></ul>
POST请求的特点：	<ol style="list-style-type: none"><li>参数通过请求体传递，不会显示在URL中，传输更为安全。</li><li>请求内容不会被缓存。</li><li>无长度限制：POST请求的数据量没有像GET请求那样的URL长度限制，适用于大数据的传输。</li><li>可能引起副作用：POST请求一般用于向服务器提交数据，可能会修改服务器上的资源（例如，创建新记录、更新数据等）。</li></ol>
客户端发起POST请求后，服务器一般的处理：	<ul style="list-style-type: none"><li>服务器接收到POST请求后，会解析请求体中的数据（例如，表单数据、JSON数据等）。</li><li>根据数据内容，服务器可能会修改数据库或执行其他操作。</li><li>服务器根据操作的结果，返回一个响应（例如，操作成功的确认信息、错误消息或跳转指令等）。</li></ul>
处理流程简要：	<ol style="list-style-type: none"><li>客户端发送POST请求，携带数据。</li><li>服务器解析请求体中的数据，并执行相应的操作（如数据库插入、更新等）。</li><li>服务器返回处理结果（如成功、失败、错误消息等）。</li></ol>

》》》》既然有请求，那么服务器处理了请求之后，会做出响应

》》》》HTTP 响应的格式：

HTTP响应是服务器向客户端（如浏览器）发送的消息，表示请求的处理结果。它通常包含以

下内容：

- 1. 状态行：指示请求的处理结果和状态码。
- 2. 响应头：包含有关响应的信息，比如内容类型、缓存控制等。
- 3. 响应体：包含实际的响应数据，如HTML页面、图片、JSON、XML等。

响应的组成部分

状态行（Status Line）	<ul style="list-style-type: none"><li>包含HTTP版本、状态码和状态消息。</li><li>例如：HTTP/1.1 200 OK<ul style="list-style-type: none"><li>HTTP/1.1：表示使用的HTTP协议版本。</li><li>200：表示请求成功（状态码）。</li><li>OK：状态码的描述信息。</li></ul></li></ul>
响应头（Response Headers）	<ul style="list-style-type: none"><li>含有描述响应内容的元信息。</li><li>例如：<ul style="list-style-type: none"><li>Content-Type: text/html 表示响应体是HTML内容。</li><li>Content-Length: 1234 表示响应体的大小是1234字节。</li><li>Cache-Control: no-cache 指示客户端不要缓存响应。</li><li>Set-Cookie: sessionId=abc123 用于设置浏览器的cookie。</li></ul></li></ul>
响应体（Response Body）	<ul style="list-style-type: none"><li>响应体包含了服务器返回的具体数据内容，如网页HTML、图片、视频、JSON数据等。</li><li>对于GET请求，响应体通常是请求的资源（如网页、图片、JSON数据等）。</li><li>对于POST请求，响应体可能是服务器操作结果的反馈，或者确认消息等。</li></ul>

常见的 HTTP 响应格式

HTML	如果客户端请求网页（如GET /index.html） ， 响应体通常是HTML格式的页面。
示例:	HTTP/1.1 200 OK Content-Type: text/html; charset=UTF-8 Content-Length: 215  <html> <head> <title>Example</title> </head> <body> <h1>Welcome to the site!</h1> </body> </html>

JSON	常用于API响应，特别是AJAX请求或RESTful API接口，响应体通常是JSON格式的数据。
示例:	HTTP/1.1 200 OK Content-Type: application/json Content-Length: 52  { "message": "Form submitted successfully", "status": "OK" }

XML	在某些情况下，特别是SOAP Web服务中，响应体可能是XML格式。
示例:	HTTP/1.1 200 OK Content-Type: application/xml Content-Length: 92  <response> <status>success</status> <message>Data processed successfully</message> </response>

图片/二进制文件	如果请求的是图片或其他二进制文件，响应体会包含这些文件的数据。
示例:	HTTP/1.1 200 OK Content-Type: image/jpeg Content-Length: 2048  [二进制图像数据]

纯文本	有时响应体可能是纯文本内容。
示例:	HTTP/1.1 200 OK Content-Type: text/plain Content-Length: 15  Hello, world!

	<body><h1>Hello, world!</h1></body></html>
--	--

POST请求的响应:	请求: POST /submit-form HTTP/1.1 (包含表单数据)
响应:	HTTP/1.1 200 OK Content-Type: application/json Content-Length: 52  {"message": "Form submitted successfully", "status": "OK"}

》》》其他响应

1. FTP (File Transfer Protocol) 响应

响应格式:	FTP响应通常包含状态码和附带信息，通常是三位数字的状态码，后跟一条信息。
用途:	用于文件传输。
示例:	◦ 200 OK: 表示命令成功执行。 ◦ 550 Requested action not taken: 表示无法访问请求的文件。
完整示例:	220 FTP server ready. 230 User logged in, proceed. 550 File not found.

2. SMTP (Simple Mail Transfer Protocol) 响应

用途:	用于邮件发送。
响应格式:	SMTP响应通常是三位数字代码，并附带一个简短的描述信息。
常见状态码:	◦ 220: 服务就绪。 ◦ 250: 请求完成。 ◦ 550: 拒绝邮件发送。
示例:	220 smtp.example.com ESMTP Exim 4.94.2 Wed, 20 May 2025 10:23:45 +0000 250 OK 550 Requested action not taken: mailbox unavailable

3. IMAP (Internet Message Access Protocol) 响应

用途:	用于电子邮件的接收与管理。
响应格式:	IMAP的响应格式也是以状态码为主，后跟响应信息，通常使用一组标签或标识符。
示例:	◦ OK: 表示命令成功执行。 ◦ NO: 表示命令失败。 ◦ BAD: 表示请求无效。
示例:	* 3 FETCH (FLAGS (\Seen) UID 1234) OK FETCH completed. * BYE IMAP4rev1 server terminating connection

4. DNS (Domain Name System) 响应

用途:	用于域名解析。
响应格式:	DNS响应通常由DNS服务器返回，它包含查询的结果，如IP地址或其他资源记录。
常见响应:	◦ 查询类型是A记录时，返回IP地址。 ◦ 查询类型是MX记录时，返回邮件交换服务器的域名。
示例:	;; ANSWER SECTION: example.com. 3600 IN A 93.184.216.34

》》》std::once\_flag 和 std::call\_once

std::once_flag:	std::once_flag 是 C++ 标准库中的同步工具，可以确保在多线程环境中某段代码（比如初始化函数） <u>只会执行一次</u> 。 std::once_flag 是一个标志对象，标记某个操作是否已经执行过，通常用于某个共享资源的初始化。
用法:	std::once_flag 通常与 std::call_once() 一起使用。你声明一个 std::once_flag 变量，然后在多线程代码中使用 std::call_once() 来确保某个代码块只执行一次。
代码示例:	std::once_flag flag;

std::call_once():	std::call_once() 是一个函数，确保传入的某个函数或代码块在多线程环境下只执行一次。 无论多少线程调用它，所传入的函数或代码块只会被执行一次，其它线程会等到第一次执行完成。
用法:	std::call_once() 接受一个 std::once_flag 对象和一个可调对象作为参数。 即便是多个线程同时调用该函数，所传入的代码（或函数）也只会被执行一次，后续的线程会跳过这一执行。
示例:	<pre>#include &lt;iostream&gt; #include &lt;thread&gt; #include &lt;mutex&gt;  std::once_flag flag;  void initialize() {     std::cout &lt;&lt; "Initializing... Only once." &lt;&lt; std::endl; }  void thread_func() {     std::call_once(flag, initialize); // 确保initialize()只执行一次 }  int main() {     std::thread t1(thread_func);     std::thread t2(thread_func);     std::thread t3(thread_func);      t1.join();     t2.join();     t3.join();     return 0; }</pre>

工作原理:	<ul style="list-style-type: none"><li>• <b>std::once_flag</b> 是一个特殊的对象，用来标记某个操作是否已经执行过。它通常是全局或静态的，在第一次调用时会被初始化。</li><li>• <b>std::call_once()</b> 会检查 std::once_flag 是否已经标记过，如果没有，它就会执行传入的函数；如果已经执行过，则其他线程会跳过这个执行。这样确保了函数只执行一次。</li></ul>
-------	---

用处:	
线程安全的初始化:	std::once_flag 和 std::call_once() 常用于保证某些初始化操作只执行一次，特别是在多线程环境下。例如，线程安全地初始化单例模式、全局资源或库加载等。
避免重复工作:	当多个线程尝试同时执行相同的初始化时，使用 std::call_once() 可以确保只执行一次初始化，避免了重复工作或多次初始化带来的潜在问题。
避免线程冲突:	std::call_once() 和 std::once_flag 可以确保代码块只被执行一次，避免了多个线程同时执行某些敏感操作的竞态条件。

》》》enable\_shared\_from\_this什么类? 在哪里定义?

定义:	enable_shared_from_this 是一个 C++ 标准库 中的模板类，它位于 <memory> 头文件中。
-----	--

	<pre>template &lt;class _Ty&gt; class enable_shared_from_this { // provide member functions that create shared_ptr to this public:     using _Esft_type = enable_shared_from_this;      _NODISCARD shared_ptr&lt;_Ty&gt; shared_from_this() {         return shared_ptr&lt;_Ty&gt;(_Wptr);     } };</pre>
作用:	enable_shared_from_this 是一个辅助类模板, 用于使得一个类的对象能够从 shared_ptr 获取指向自己的 shared_ptr 实例。
思考:	<p>通常, 只有对象是由一个 shared_ptr 管理时, 你才可以对其使用 enable_shared_from_this。</p> <p><b>你会想, 我们明明可以使用 this, 而 this 指针也能获得指向该对象的指针, 为什么需要使用 shared_from_this 来获取呢?</b></p> <p>答案是:</p> <p>假设你的对象由智能指针管理 (特指 shared_ptr, 因为共享指针 shared_ptr 会自动计数, 如果该对象被引用时, 引用计数会自动 +1), 当我们想要使用对象时, 如果不小心在之前将该智能指针类型的对象引用清零 (即没有代码调用/引用该对象时, 智能指针中的引用计数会递减, 直到归零, 如果系统检测到计数为0, 则会结束该智能指针的生命周期), 那么在之后的使用中, 这个指针类型的对象便是无效的。</p> <p>为了避免这种情况发生, 我们在调用之前提前使用 auto self = shared_from_this(); 这段代码, 不仅仅获取指针 (该指针存储的内容和 this 指针完全相同), 同时还为智能指针添加一个引用计数, 这确保我们在使用对象指针的过程中, 操作始终有效。</p>
详细描述:	<p>1. 类模板定义: enable_shared_from_this 是一个模板类, 接受一个类型参数 T, 通常是你想要使其能够获取 shared_ptr 的类。</p> <p>例如, 假设有一个类 MyClass, 你希望它能从 shared_ptr&lt;MyClass&gt; 中获取指向它自己的 shared_ptr, 则 MyClass 类可以继承自 enable_shared_from_this&lt;MyClass&gt;。</p> <p>2. 实现方式: enable_shared_from_this 提供了一个成员函数 shared_from_this(), 允许从当前对象中获取一个指向自己的 shared_ptr。</p> <p>3. 但是, shared_from_this() 只能在对象已经由一个 shared_ptr 管理时调用, 否则会抛出异常 (std::bad_weak_ptr)。</p>
示例:	<pre>#include &lt;iostream&gt; #include &lt;memory&gt;  class MyClass : public std::enable_shared_from_this&lt;MyClass&gt; { public:     void print() {         std::cout &lt;&lt; "Hello, I am MyClass!" &lt;&lt; std::endl;     }      void example() {         // 获取指向当前对象的shared_ptr         std::shared_ptr&lt;MyClass&gt; ptr = shared_from_this();         ptr-&gt;print(); // 使用shared_ptr调用成员函数     } };  int main() {     std::shared_ptr&lt;MyClass&gt; ptr = std::make_shared&lt;MyClass&gt;();     ptr-&gt;example(); // 正常调用 (ptr 会调用 MyClass的成员函数, 该成员函数使用了 shared_from_this() 实现其详细操作)     return 0; }</pre>

》》》》我终于明白为什么需要使用 shared\_from\_this 了 (请看实例: m\_Socket 是一个已经被定义的变量)

这样的代码有一种危险:	
你想直接在 Lambda 中捕获 m_Socket 和 m_Deadline, 但这两个变量是 HttpConnection 类的成员变量 (也就是你当前对象的成员)。	
如果你直接捕获这些成员变量, 它们会在 Lambda 执行时被销毁, 导致你在异步操作过程中访问到悬空对象 (悬空引用), 即这些对象已经不再存在或者被销毁了。	
解决方法是使用 shared_from_this():	
它返回一个 shared_ptr, 确保当前对象在每一次的 Lambda 回调中仍然有效, 不会被销毁。	<pre>auto self = shared_from_this(); boost::beast::http::async_write(XXX, XXX, [m_Socket, m_Deadline](boost::beast::error_code ec, std::size_t size) {     m_Socket.shutdown(XXX, ec); });  auto self = shared_from_this(); boost::beast::http::async_write(XXX, XXX, [self](boost::beast::error_code ec, std::size_t size) {     self-&gt;m_Socket.shutdown(XXX, ec); });</pre>

》》》》 unsigned int 和 unsigned short 的区别? ?

unsigned short:	通常是一个 16 位（2 字节）的无符号整数，值的范围是 0 到 65535。
unsigned int:	通常是一个 32 位（4 字节）的无符号整数，值的范围是 0 到 4294967295。

》》》》 什么是左值，什么是右值?

左值 (Lvalue) :

特点:	<ul style="list-style-type: none"><li>可以被赋值。</li><li>在程序中有明确的内存地址。即 --&gt; 左值是可寻址的（例如，变量名就是一个左值）。</li></ul>
定义:	左值是指可以出现在赋值符号 “=” 左边的表达式。简单来说，左值是有持久位置的对象，可以通过其地址进行访问。
示例:	<pre>int x = 10;    // x 是左值 x = 20;        // x 作为左值出现在赋值左边</pre>

右值 (Rvalue) :

定义:	右值是指不能出现在赋值符号 “=” 左边的表达式，通常是临时对象或即将销毁的对象。
特点:	<ul style="list-style-type: none"><li>右值是没有持久地址的。</li><li>右值通常表示临时值，不能对其进行修改（不能取地址）。</li><li>右值通常是表达式的结果，如常量或函数返回的临时值。</li></ul>
示例:	<pre>int x = 10; int y = x + 5; // x + 5 是右值</pre>

左值引用:

左值引用:	int& 是左值引用类型，表示对一个左值的引用。
示例:	<pre>int a = 5; int&amp; b = a; // b 是 a 的左值引用</pre>

右值引用:

右值引用:	int&& 是右值引用类型，表示对一个右值的引用。右值引用允许我们移动资源而不是复制它们。
示例:	<pre>int&amp;&amp; c = 10; // c 是右值引用，绑定到右值 10</pre>

》》》》 std::move ?



std::move 可以将左值转为右值吗?

是的，std::move 可以将左值转为右值。

std::move 本身并不会移动任何数据，它的作用只是将一个左值转换为右值引用，从而允许将这个对象的资源转移（移动）给另一个对象。这表明我们想要“移动”这个对象的资源，而不是拷贝它。

示例:	<pre>int x = 10; int&amp;&amp; y = std::move(x); // std::move 将左值 x 转为右值引用 y</pre>
-----	--

为什么需要 std::move?

在 C++ 中，移动语义允许资源（如动态内存或文件句柄）从一个对象转移到另一个对象，而不是进行昂贵的拷贝操作。std::move 是实现这一机制的工具，告诉编译器这个对象不再需要，且其资源可以安全地转移。

示例：	<pre>HttpConnection::HttpConnection(boost::asio::ip::tcp::socket socket)     : m_Socket(std::move(socket))    // 使用 std::move, 将 socket 的资源转移到 m_Socket { }  boost::asio::ip::tcp::socket 是一个非拷贝类型（禁止拷贝构造和拷贝赋值），也就是说它不允许被拷贝。 如果你尝试直接将 socket 赋值给 m_Socket 而不使用 std::move, 编译器会报错，因为它无法进行拷贝。  <b>通过 std::move 转移资源：</b> std::move(socket) 将 socket 转换成右值引用，允许编译器将 socket 的内部资源（如缓冲区、网络连接等）转移给 m_Socket, 而不是进行拷贝。 这样可以避免不必要的资源复制，并且能够正确地初始化 m_Socket。</pre>
-----	---

》》》复制与移动

复制（Copying）：

定义：	复制是将一个对象的内容拷贝到另一个对象中。通常，复制会创建对象的一个副本，并且源对象和目标对象各自拥有自己的资源。
过程：	当进行复制时，原始对象的内容（如内存、数据等）会被逐个拷贝到新的内存位置。复制操作会导致开销，特别是当对象包含动态分配的资源（如指针、文件句柄等）时。
示例：	<pre>std::vector&lt;int&gt; vec1 = {1, 2, 3}; std::vector&lt;int&gt; vec2 = vec1;    // 复制, vec2 拷贝了 vec1 的所有元素</pre>

移动（Moving / 资源转移）：

定义：	移动是将一个对象的资源（如内存、指针等）从一个对象转移到另一个对象，而不是复制这些资源。移动后的原对象通常不再拥有这些资源，移动操作不会对这些资源进行复制，而是直接“转移”它们的所有权。
具体操作：	在移动操作中，原对象持有某些资源（例如动态分配的内存、文件句柄等），而目标对象需要接管这些资源。移动操作通常通过直接传递资源的地址或指针来实现这一转移。简而言之，原对象的指针（内存地址）会直接赋给目标对象。 例如，如果一个对象内有一个指向动态分配内存的指针，移动操作会将该指针直接转移到新对象，而原对象的指针会被置为 nullptr 或清空，从而避免资源的重复释放。
过程：	移动操作的关键是资源的所有权转移，原对象会被标记为“空”或“无效”状态，而目标对象获得资源的所有权。移动通常比复制更高效，因为它避免了不必要的资源分配和复制。
示例：	<pre>std::vector&lt;int&gt; vec1 = {1, 2, 3}; std::vector&lt;int&gt; vec2 = std::move(vec1); // 移动, vec2 获得 vec1 的资源, vec1 变为空状态</pre>

移动和复制的关系：

复制：	拷贝资源，两个对象各自拥有独立的资源副本，可能需要额外的内存分配。
移动：	转移资源的所有权，一个对象不再拥有资源，另一个对象获得所有权，通常不需要额外的内存分配或数据拷贝。

举例：（移动和复制的关系：）

用我的话来讲，复制就是：	我有一套房子，如果你想使用我的房子做一些事情，则需要你自己去买一套一模一样的房子，并可由你自己进行装潢。
而移动则是：	我有一套房子，如果你想使用我的房子做一些事情，我直接把房产证给你，在拥有者这一栏划掉我的名字，并写上你的名字，转移所有权，然后房间任你处置。



》》》什么是 static\_cast? 为什么需要使用 static\_cast?

什么是 static\_cast ?

static\_cast 是 C++ 中的一种类型转换操作符，用于在类型之间进行显式的转换。它在编译时进行类型检查，能够安全地将一种类型转换为另一种类型，只要这种转换是合法的。与其他类型转换操作符（如 reinterpret\_cast 或 dynamic\_cast）不同，static\_cast 只适用于类型之间具有明确关系（如继承关系或基本类型转换）的转换。

为什么需要 static\_cast<unsigned short>(8080)?

- 1. 类型转换：在 C++ 中，8080 是一个常量整数（int 类型）。然而，unsigned short 是一种较小的整数类型，通常为 16 位，表示范围从 0 到 65535（通常是 2^16 - 1）。  
当我们将一个值赋给 unsigned short 类型的变量时，必须确保它符合该类型的范围。  

- 2. 显式转换：使用 static\_cast<unsigned short>(8080) 将 8080 显式地转换为 unsigned short 类型。  
虽然 8080 在 int 类型范围内，但是为了确保类型的正确性和避免潜在的隐式转换错误，显式地使用 static\_cast 可以帮助我们：
  - 明确指定我们想要的类型转换。
  - 使代码更具可读性和可维护性，特别是在类型转换可能影响程序行为时。  
如果 8080 的值超过了 unsigned short 的上限（65535），赋值可能会导致数据丢失或溢出。使用 static\_cast 可以明确指定转换类型，并且在编译时提醒你进行这种转换。
  - 即使 8080 在 unsigned short 的有效范围内，使用 static\_cast 也是一种良好的编程实践，确保代码清晰、显式。

》》提示

```
unsigned short port = unsigned short(8080);
```

这样写效果也是一样的。

》》》EXIT\_FAILURE

意义：	EXIT_FAILURE 是一个宏常量，用于表示程序执行失败的退出状态。它通常在程序执行异常时返回，表示程序未正常完成，错误退出。 而 EXIT_SUCCESS 与其相对，表示程序成功退出。
定义：	EXIT_FAILURE 在标准库头文件 <cstdlib> 中定义。  #define EXIT_FAILURE 1 <ul style="list-style-type: none"><li>EXIT_FAILURE 通常被定义为 1（某些系统中可能是其他值），代表程序异常退出。</li><li>EXIT_SUCCESS 是另一个常量，通常定义为 0，表示程序成功退出。</li></ul>

