4 应用层自定义协议与序列化

4-1 应用层

我们程序员写的一个个解决我们实际问题, 满足我们日常需求的网络程序, 都是在应用层.

再谈 "协议"

协议是一种 "约定". socket api 的接口, 在读写数据时, 都是按 "字符串" 的方式来发送接收的. 如果我们要传输一些 "结构化的数据" 怎么办呢?

> 其实, 协议就是双方约定好的结构化的数据

网络版计算器

例如, 我们需要实现一个服务器版的加法器. 我们需要客户端把要计算的两个加数发过去, 然后由服务器进行计算, 最后再把结果返回给客户端.

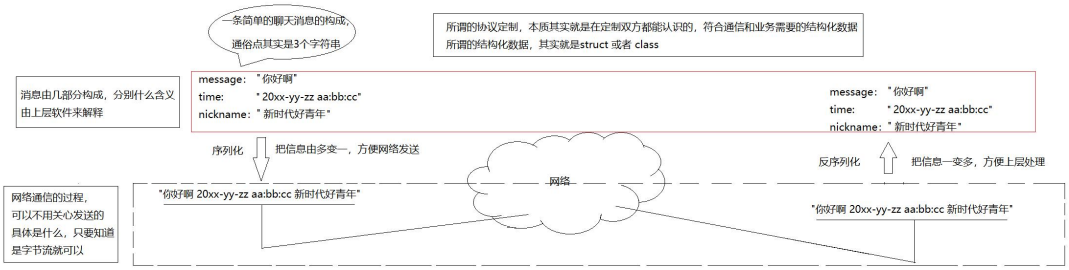
约定方案一:

> - 客户端发送一个形如"1+1"的字符串;
> - 这个字符串中有两个操作数, 都是整形;
> - 两个数字之间会有一个字符是运算符, 运算符只能是 + ;
> - 数字和运算符之间没有空格;
> - ...

约定方案二:

> - 定义结构体来表示我们需要交互的信息;
> - 发送数据时将这个结构体按照一个规则转换成字符串, 接收到数据的时候再按照相同的规则把字符串转化回结构体;
> - 这个过程叫做 "序列化" 和 "反序列化"

序列化 和 反序列化

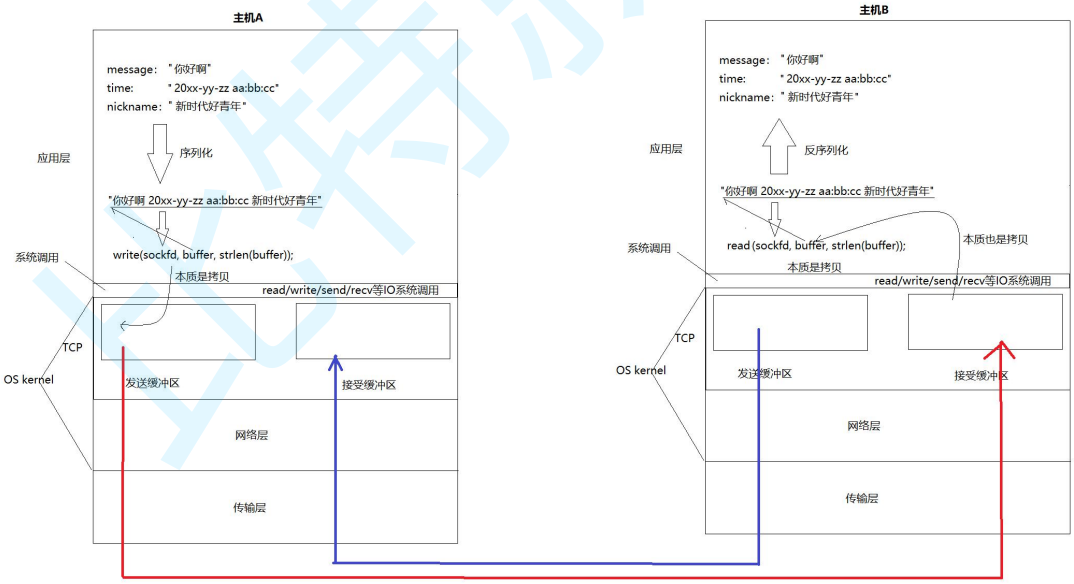


无论我们采用方案一, 还是方案二, 还是其他的方案, 只要保证, 一端发送时构造的数据, 在另一端能够正确的进行解析, 就是 ok 的. 这种约定, 就是 **应用层协议**

但是，为了让我们深刻理解协议，我们打算自定义实现一下协议的过程。

- 我们采用方案 2，我们也要体现协议定制的细节
- 我们要引入序列化和反序列化，只不过我们课堂直接采用现成的方案 -- jsoncpp 库
- 我们要对 socket 进行字节流的读取处理

4-2 重新理解 read、write、recv、send 和 tcp 为什么支持全双工



所以:

- 在任何一台主机上，TCP 连接既有发送缓冲区，又有接受缓冲区，所以，在内核中，可以在发消息的同时，也可以收消息，即全双工
- 这就是为什么一个 tcp sockfd 读写都是它的原因

- 实际数据什么时候发，发多少，出错了怎么办，由 TCP 控制，所以 TCP 叫做传输控制协议

4-3 开始实现

代码结构

```
C++
Calculate.hpp  Makefile      Socket.hpp        TcpServer.hpp
Daemon.hpp     Protocol.hpp  TcpClientMain.cc  TcpServerMain.cc
// 简单起见，可以直接采用自定义线程
// 为了减少课堂时间的浪费，也建议不用用户输入，直接 client<<->>server 通信，这样可以省去编写没有干货的代码
```

Socket 封装

socket.hpp

```
C++
#pragma once

#include <iostream>
#include <string>
#include <cstring>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

#define Convert(addrptr) ((struct sockaddr *)addrptr)

namespace Net_Work
{
    const static int defaultsockfd = -1;
    const int backlog = 5;

    enum
    {
        SocketError = 1,
        BindError,
        ListenError,
```

```
    };

    // 封装一个基类，Socket 接口类
    // 设计模式：模版方法类
    class Socket
    {
    public:
        virtual ~Socket() {}
        virtual void CreateSocketOrDie() = 0;
        virtual void BindSocketOrDie(uint16_t port) = 0;
        virtual void ListenSocketOrDie(int backlog) = 0;
        virtual Socket *AcceptConnection(std::string *peerip,
uint16_t *peerport) = 0;
        virtual bool ConnectServer(std::string &serverip, uint16_t
serverport) = 0;
        virtual int GetSockFd() = 0;
        virtual void SetSockFd(int sockfd) = 0;
        virtual void CloseSocket() = 0;
        virtual bool Recv(std::string *buffer, int size) = 0;
        virtual void Send(std::string &send_str) = 0;
        // TODO
    public:
        void BuildListenSocketMethod(uint16_t port, int backlog)
        {
            CreateSocketOrDie();
            BindSocketOrDie(port);
            ListenSocketOrDie(backlog);
        }
        bool BuildConnectSocketMethod(std::string &serverip,
uint16_t serverport)
        {
            CreateSocketOrDie();
            return ConnectServer(serverip, serverport);
        }
        void BuildNormalSocketMethod(int sockfd)
        {
            SetSockFd(sockfd);
        }
    };

    class TcpSocket : public Socket
    {
    public:
        TcpSocket(int sockfd = defaultsockfd) : _sockfd(sockfd)
```

```
        {
        }
        ~TcpSocket()
        {
        }
        void CreateSocketOrDie() override
        {
            _sockfd = ::socket(AF_INET, SOCK_STREAM, 0);
            if (_sockfd < 0)
                exit(SocketError);
        }
        void BindSocketOrDie(uint16_t port) override
        {
            struct sockaddr_in local;
            memset(&local, 0, sizeof(local));
            local.sin_family = AF_INET;
            local.sin_addr.s_addr = INADDR_ANY;
            local.sin_port = htons(port);

            int n = ::bind(_sockfd, Convert(&local),
sizeof(local));
            if (n < 0)
                exit(BindError);
        }
        void ListenSocketOrDie(int backlog) override
        {
            int n = ::listen(_sockfd, backlog);
            if (n < 0)
                exit(ListenError);
        }
        Socket *AcceptConnection(std::string *peerip, uint16_t
*peerport) override
        {
            struct sockaddr_in peer;
            socklen_t len = sizeof(peer);
            int newsockfd = ::accept(_sockfd, Convert(&peer),
&len);
            if (newsockfd < 0)
                return nullptr;
            *peerport = ntohs(peer.sin_port);
            *peerip = inet_ntoa(peer.sin_addr);
            Socket *s = new TcpSocket(newsockfd);
            return s;
        }
```

```
        bool ConnectServer(std::string &serverip, uint16_t
serverport) override
        {
            struct sockaddr_in server;
            memset(&server, 0, sizeof(server));
            server.sin_family = AF_INET;
            server.sin_addr.s_addr = inet_addr(serverip.c_str());
            server.sin_port = htons(serverport);

            int n = ::connect(_sockfd, Convert(&server),
sizeof(server));
            if (n == 0)
                return true;
            else
                return false;
        }
        int GetSockFd() override
        {
            return _sockfd;
        }
        void SetSockFd(int sockfd) override
        {
            _sockfd = sockfd;
        }
        void CloseSocket() override
        {
            if (_sockfd > defaultsockfd)
                ::close(_sockfd);
        }
        bool Recv(std::string *buffer, int size) override
        {
            char inbuffer[size];
            ssize_t n = recv(_sockfd, inbuffer, size-1, 0);
            if(n > 0)
            {
                inbuffer[n] = 0;
                *buffer += inbuffer; // 故意拼接的
                return true;
            }
            else if(n == 0) return false;
            else return false;
        }
        void Send(std::string &send_str) override
        {
```

```
            // 多路转接我们在统一说
            send(_sockfd, send_str.c_str(), send_str.size(), 0);
        }
    private:
        int _sockfd;
    };

}
```

定制协议

基本结构

- 定制基本的结构化字段，这个就是协议

```
C++

class Request
{
private:
    // _data_x _oper _data_y
    // 报文的自描述字段
    // "len\r\nx op y\r\n" : \r\n 不属于报文的一部分，约定
    // 很多工作都是在做字符串处理！
    int _data_x; // 第一个参数
    int _data_y; // 第二个参数
    char _oper;  // + - * / %
};

class Response
{
private:
    // "len\r\n_result _code\r\n"
    int _result; // 运算结果
    int _code;   // 运算状态
};
```

protocol.hpp

```
C++
#pragma once

#include <iostream>
```

```
#include <memory>
#include <jsoncpp/json/json.h>

namespace Protocol
{
    // 问题
    // 1. 结构化数据的序列和反序列化
    // 2. 还要解决用户区分报文边界 --- 数据包粘报问题

    // 讲法
    // 1. 自定义协议
    // 2. 成熟方案序列和反序列化

    // 总结：
    // 我们今天定义了几组协议呢？？我们可以同时存在多个协议吗？？？可以
    // "protocol_code\r\nlen\r\nx op y\r\n" : \r\n 不属于报文的一部分，约定

    const std::string ProtSep = " ";
    const std::string LineBreakSep = "\r\n";

    // "len\r\nx op y\r\n" : \r\n 不属于报文的一部分，约定
    std::string Encode(const std::string &message)
    {
        std::string len = std::to_string(message.size());
        std::string package = len + LineBreakSep + message + LineBreakSep;
        return package;
    }
    // "len\nx op y\n" : \n 不属于报文的一部分，约定
    // 我无法保证 package 就是一个独立的完整的报文
    // "l
    // "len
    // "len\r\n
    // "len\r\nx
    // "len\r\nx op
    // "len\r\nx op y
    // "len\r\nx op y\r\n"
    // "len\r\nx op y\r\n""len
    // "len\r\nx op y\r\n""len\n
    // "len\r\nx op
    // "len\r\nx op y\r\n""len\nx op y\r\n"
    // "len\r\nresult code\r\n""len\nresult code\r\n"
```

```
    bool Decode(std::string &package, std::string *message)
    {
        // 除了解包，我还想判断报文的完整性，能否正确处理具有"边界"的报
文
        auto pos = package.find(LineBreakSep);
        if (pos == std::string::npos)
            return false;
        std::string lens = package.substr(0, pos);
        int messagelen = std::stoi(lens);
        int total = lens.size() + messagelen + 2 *
LineBreakSep.size();
        if (package.size() < total)
            return false;
        // 至少 package 内部一定有一个完整的报文了！
        *message = package.substr(pos + LineBreakSep.size(),
messagelen);
        package.erase(0, total);
        return true;
    }

    class Request
    {
    public:
        Request() : _data_x(0), _data_y(0), _oper(0)
        {
        }
        Request(int x, int y, char op) : _data_x(x), _data_y(y),
_oper(op)
        {
        }
        void Debug()
        {
            std::cout << "_data_x: " << _data_x << std::endl;
            std::cout << "_data_y: " << _data_y << std::endl;
            std::cout << "_oper: " << _oper << std::endl;
        }
        void Inc()
        {
            _data_x++;
            _data_y++;
        }
        // 结构化数据->字符串
        bool Serialize(std::string *out)
```

```
        {
            Json::Value root;
            root["datax"] = _data_x;
            root["datay"] = _data_y;
            root["oper"] = _oper;
            Json::FastWriter writer;
            *out = writer.write(root);
            return true;
        }
        bool Deserialize(std::string &in) // "x op y" []
        {
            Json::Value root;
            Json::Reader reader;
            bool res = reader.parse(in, root);
            if(res)
            {
                _data_x = root["datax"].asInt();
                _data_y = root["datay"].asInt();
                _oper = root["oper"].asInt();
            }
            return res;
        }
        int GetX() { return _data_x; }
        int GetY() { return _data_y; }
        char GetOper() { return _oper; }

    private:
        // _data_x _oper _data_y
        // 报文的自描述字段
        // "len\r\nx op y\r\n" : \r\n 不属于报文的一部分，约定
        // 很多工作都是在做字符串处理!
        int _data_x; // 第一个参数
        int _data_y; // 第二个参数
        char _oper;  // + - * / %
    };

    class Response
    {
    public:
        Response() : _result(0), _code(0)
        {
        }
        Response(int result, int code) : _result(result),
_code(code)
```

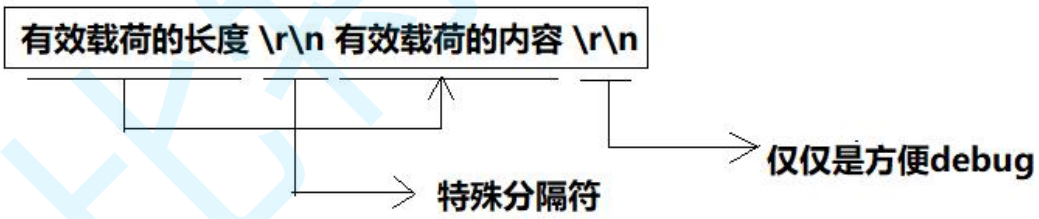
```
        {
        }
        bool Serialize(std::string *out)
        {
            Json::Value root;
            root["result"] = _result;
            root["code"] = _code;
            Json::FastWriter writer;
            *out = writer.write(root);
            return true;
        }
        bool Deserialize(std::string &in) // "_result _code" [)
        {
            Json::Value root;
            Json::Reader reader;
            bool res = reader.parse(in, root);
            if(res)
            {
                _result = root["result"].asInt();
                _code = root["code"].asInt();
            }
            return res;
        }
        void SetResult(int res) { _result = res; }
        void SetCode(int code) { _code = code; }
        int GetResult() { return _result; }
        int GetCode() { return _code; }

    private:
        // "len\r\n_result _code\r\n"
        int _result; // 运算结果
        int _code;   // 运算状态
    };

    // 简单的工厂模式，建造类设计模式
    class Factory
    {
    public:
        std::shared_ptr<Request> BuildRequest()
        {
            std::shared_ptr<Request> req =
std::make_shared<Request>();
            return req;
        }
```

```
        std::shared_ptr<Request> BuildRequest(int x, int y, char
op)
        {
            std::shared_ptr<Request> req =
std::make_shared<Request>(x, y, op);
            return req;
        }
        std::shared_ptr<Response> BuildResponse()
        {
            std::shared_ptr<Response> resp =
std::make_shared<Response>();
            return resp;
        }
        std::shared_ptr<Response> BuildResponse(int result, int
code)
        {
            std::shared_ptr<Response> req =
std::make_shared<Response>(result, code);
            return req;
        }
    };
}
```

期望的报文格式



4-4 关于流式数据的处理

- 你如何保证你每次读取就能读完请求缓冲区的所有内容?
- 你怎么保证读取完毕或者读取没有完毕的时候，读到的就是一个完整的请求呢?
- 处理 TCP 缓冲区中的数据，一定要保证正确处理请求

```
C++
    const std::string ProtSep = " ";
```

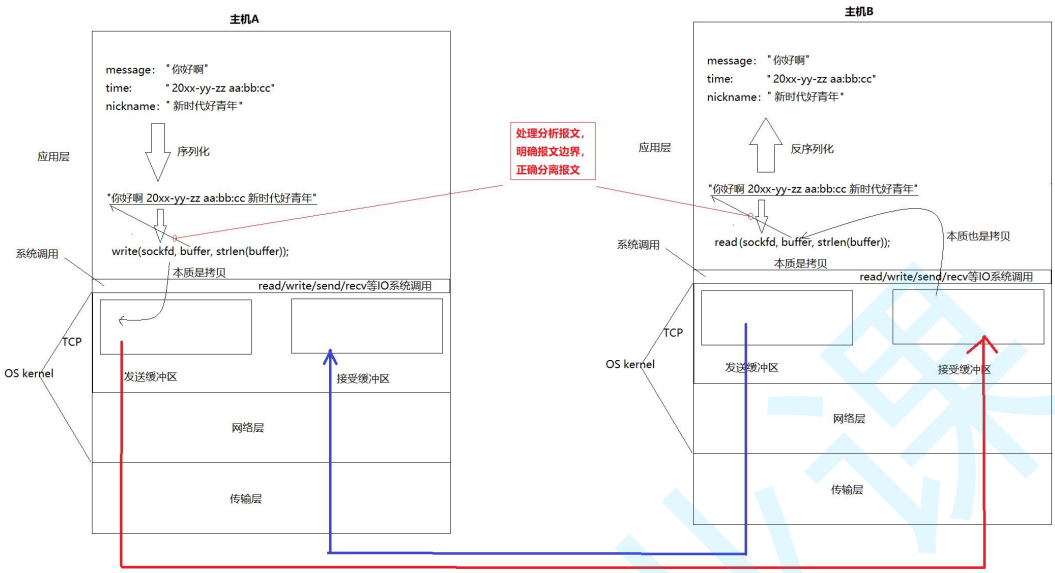
```
    const std::string LineBreakSep = "\n";

    // "len\nx op y\n" : \n 不属于报文的一部分，约定
    std::string Encode(const std::string &message)
    {
        std::string len = std::to_string(message.size());
        std::string package = len + LineBreakSep + message +
LineBreakSep;
        return package;
    }
    // "len\nx op y\n" : \n 不属于报文的一部分，约定
    // 我无法保证 package 就是一个独立的完整的报文
    // "l
    // "len
    // "len\n
    // "len\nx
    // "len\nx op
    // "len\nx op y
    // "len\nx op y\n"
    // "len\nx op y\n""len
    // "len\nx op y\n""len\n
    // "len\nx op
    // "len\nx op y\n""len\nx op y\n"
    // "len\nresult code\n""len\nresult code\n"

    bool Decode(std::string &package, std::string *message)
    {
        // 除了解包，我还想判断报文的完整性，能否正确处理具有"边界"的报
文
        auto pos = package.find(LineBreakSep);
        if (pos == std::string::npos)
            return false;
        std::string lens = package.substr(0, pos);
        int messagelen = std::stoi(lens);
        int total = lens.size() + messagelen + 2 *
LineBreakSep.size();
        if (package.size() < total)
            return false;
        // 至少 package 内部一定有一个完整的报文了！
        *message = package.substr(pos + LineBreakSep.size(),
messagelen);
        package.erase(0, total);
        return true;
```

```
    }
```

所以，完整的处理过程应该是：



附录

Jsoncpp

Jsoncpp 是一个用于处理 JSON 数据的 C++ 库。它提供了将 JSON 数据序列化为字符串以及从字符串反序列化为 C++ 数据结构的功能。Jsoncpp 是开源的，广泛用于各种需要处理 JSON 数据的 C++ 项目中。

特性

1. 简单易用：Jsoncpp 提供了直观的 API，使得处理 JSON 数据变得简单。
2. 高性能：Jsoncpp 的性能经过优化，能够高效地处理大量 JSON 数据。
3. 全面支持：支持 JSON 标准中的所有数据类型，包括对象、数组、字符串、数字、布尔值和 null。
4. 错误处理：在解析 JSON 数据时，Jsoncpp 提供了详细的错误信息和位置，方便开发者调试。

当使用 Jsoncpp 库进行 JSON 的序列化和反序列化时，确实存在不同的做法和工具类可供选择。以下是对 Jsoncpp 中序列化和反序列化操作的详细介绍：

安装

```
C++
ubuntu：sudo apt-get install libjsoncpp-dev
Centos: sudo yum install jsoncpp-devel
```

序列化

序列化指的是将数据结构或对象转换为一种格式，以便在网络上传输或存储到文件中。Jsoncpp 提供了多种方式进行序列化：

1. **使用** `Json::Value` 的 `toStyledString` 方法：
   - 优点：将 `Json::Value` 对象直接转换为格式化的 JSON 字符串。
   - 示例：

```
C++
#include <iostream>
#include <string>
#include <jsoncpp/json/json.h>

int main()
{
    Json::Value root;
    root["name"] = "joe";
    root["sex"] = "男";
    std::string s = root.toStyledString();

    std::cout << s << std::endl;

    return 0;
}

$ ./test.exe
{
   "name" : "joe",
   "sex" : "男"
}
```

2. **使用** `Json::StreamWriter`：
   - 优点：提供了更多的定制选项，如缩进、换行符等。
   - 示例：

```
C++
```

```
#include <iostream>
#include <string>
#include <sstream>
#include <memory>
#include <jsoncpp/json/json.h>

int main()
{
    Json::Value root;
    root["name"] = "joe";
    root["sex"] = "男";
    Json::StreamWriterBuilder wbuilder; // StreamWriter 的
工厂
    std::unique_ptr<Json::StreamWriter>
writer(wbuilder.newStreamWriter());
    std::stringstream ss;
    writer->write(root, &ss);
    std::cout << ss.str() << std::endl;
    return 0;
}

$ ./test.exe
{
        "name" : "joe",
        "sex" : "男"
}
```

3. **使用** `Json::FastWriter`:
   - 优点：比 `StyledWriter` 更快，因为它不添加额外的空格和换行符。
   - 示例:

```
C++
#include <iostream>
#include <string>
#include <sstream>
#include <memory>
#include <jsoncpp/json/json.h>

int main()
{
    Json::Value root;
    root["name"] = "joe";
    root["sex"] = "男";
```

```
    Json::FastWriter writer;
    std::string s = writer.write(root);
    std::cout << s << std::endl;
    return 0;
}

$ ./test.exe
{"name":"joe","sex":"男"}


#include <iostream>
#include <string>
#include <sstream>
#include <memory>
#include <jsoncpp/json/json.h>

int main()
{
    Json::Value root;
    root["name"] = "joe";
    root["sex"] = "男";
    // Json::FastWriter writer;
    Json::StyledWriter writer;
    std::string s = writer.write(root);
    std::cout << s << std::endl;
    return 0;
}

$ ./test.exe
{
   "name" : "joe",
   "sex" : "男"
}
```

反序列化

反序列化指的是将序列化后的数据重新转换为原来的数据结构或对象。Jsoncpp 提供了以下方法进行反序列化：

1. **使用** `Json::Reader`：
    - 优点：提供详细的错误信息和位置，方便调试。
    - 示例：

```
C++
#include <iostream>
#include <string>
#include <jsoncpp/json/json.h>

int main() {
    // JSON 字符串
    std::string json_string = "{\"name\":\"张三\",
\"age\":30, \"city\":\"北京\"}";

    // 解析 JSON 字符串
    Json::Reader reader;
    Json::Value root;

    // 从字符串中读取 JSON 数据
    bool parsingSuccessful = reader.parse(json_string,
root);

    if (!parsingSuccessful) {
        // 解析失败，输出错误信息
        std::cout << "Failed to parse JSON: " <<
reader.getFormattedErrorMessages() << std::endl;
        return 1;
    }

    // 访问 JSON 数据
    std::string name = root["name"].asString();
    int age = root["age"].asInt();
    std::string city = root["city"].asString();

    // 输出结果
    std::cout << "Name: " << name << std::endl;
    std::cout << "Age: " << age << std::endl;
    std::cout << "City: " << city << std::endl;

    return 0;
}

$ ./test.exe
Name: 张三
Age: 30
City: 北京
```

2. **使用** `Json::CharReader` 的派生类(不推荐了，上面的足够了)：
    - 在某些情况下，你可能需要更精细地控制解析过程，可以直接使用 `Json::CharReader` 的派生类。
    - 但通常情况下，使用 `Json::parseFromStream` 或 `Json::Reader` 的 `parse` 方法就足够了。

总结

- `toStyledString`、`StreamWriter` 和 `FastWriter` 提供了不同的序列化选项，你可以根据具体需求选择使用。
- `Json::Reader` 和 `parseFromStream` 函数是 Jsoncpp 中主要的反序列化工具，它们提供了强大的错误处理机制。
- 在进行序列化和反序列化时，请确保处理所有可能的错误情况，并验证输入和输出的有效性。

Json::Value

`Json::Value` 是 Jsoncpp 库中的一个重要类，用于表示和操作 JSON 数据结构。以下是一些常用的 `Json::Value` 操作列表：

1. 构造函数

- `Json::Value()`：默认构造函数，创建一个空的 `Json::Value` 对象。
- `Json::Value(ValueType type, bool allocated = false)`：根据给定的 `ValueType`（如 `nullValue`, `intValue`, `stringValue` 等）创建一个 `Json::Value` 对象。

2. 访问元素

- `Json::Value& operator[](const char* key)`：通过键（字符串）访问对象中的元素。如果键不存在，则创建一个新的元素。
- `Json::Value& operator[](const std::string& key)`：同上，但使用 `std::string` 类型的键。
- `Json::Value& operator[](ArrayIndex index)`：通过索引访问数组中的元素。如果索引超出范围，则创建一个新的元素。

- `Json::Value& at(const char* key)`：通过键访问对象中的元素，如果键不存在则抛出异常。
- `Json::Value& at(const std::string& key)`：同上，但使用 `std::string` 类型的键。

3. 类型检查

- `bool isNull()`：检查值是否为 null。
- `bool isBool()`：检查值是否为布尔类型。
- `bool isInt()`：检查值是否为整数类型。
- `bool isInt64()`：检查值是否为 64 位整数类型。
- `bool isUInt()`：检查值是否为无符号整数类型。
- `bool isUInt64()`：检查值是否为 64 位无符号整数类型。
- `bool isIntegral()`：检查值是否为整数或可转换为整数的浮点数。
- `bool isDouble()`：检查值是否为双精度浮点数。
- `bool isNumeric()`：检查值是否为数字（整数或浮点数）。
- `bool isString()`：检查值是否为字符串。
- `bool isArray()`：检查值是否为数组。
- `bool isObject()`：检查值是否为对象（即键值对的集合）。

4. 赋值和类型转换

- `Json::Value& operator=(bool value)`：将布尔值赋给 `Json::Value` 对象。
- `Json::Value& operator=(int value)`：将整数赋给 `Json::Value` 对象。
- `Json::Value& operator=(unsigned int value)`：将无符号整数赋给 `Json::Value` 对象。
- `Json::Value& operator=(Int64 value)`：将 64 位整数赋给 `Json::Value` 对象。
- `Json::Value& operator=(UInt64 value)`：将 64 位无符号整数赋给 `Json::Value` 对象。
- `Json::Value& operator=(double value)`：将双精度浮点数赋给 `Json::Value` 对象。
- `Json::Value& operator=(const char* value)`：将 C 字符串赋给 `Json::Value` 对象。

- `Json::Value& operator=(const std::string& value)`：将 `std::string` 赋给 `Json::Value` 对象。
- `bool asBool()`：将值转换为布尔类型（如果可能）。
- `int asInt()`：将值转换为整数类型（如果可能）。
- `Int64 asInt64()`：将值转换为 64 位整数类型（如果可能）。
- `unsigned int asUInt()`：将值转换为无符号整数类型（如果可能）。
- `UInt64 asUInt64()`：将值转换为 64 位无符号整数类型（如果可能）。
- `double asDouble()`：将值转换为双精度浮点数类型（如果可能）。
- `std::string asString()`：将值转换为字符串类型（如果可能）。

5. 数组和对象操作

- `size_t size()`：返回数组或对象中的元素数量。
- `bool empty()`：检查数组或对象是否为空。
- `void resize(ArrayIndex newSize)`：调整数组的大小。
- `void clear()`：删除数组或对象中的所有元素。
- `void append(const Json::Value& value)`：在数组末尾添加一个新元素。
- `Json::Value& operator[](const char* key, const Json::Value& defaultValue = Json::nullValue)`：在对象中插入或访问一个元素，如果键不存在则使用默认值。
- `Json::Value& operator[](const std::string& key, const Json::Value& defaultValue = Json::nullValue)`：同上，但使用 `std::string` 类型的