

天气预报（布局/样式表/Http通信/Json解析）

本章会实现一个相对完整的天气预报项目

该项目就是通过 `HTTP` 接口访问 `HTTP` 服务器，获取天气数据，然后展示出来

1. 整体效果、技术点

本章实现的天气预报项目，整体效果如下：



在左上角输入要查询的城市，然后点击查询按钮，就会发送 `HTTP` 请求给服务器，请求回来的天气数据 `JSON` 格式

通过解析 `JSON` 可以获取以下信息：

- 今天的信息

温度、湿度、风向、风力、天气类型（晴、多云、小雨等）、`PM2.5`、温馨提示、感冒指数、日出日落

- 未来15天的信息

日期、星期、天气类型（晴、多云、小雨等）、`PM25`、最高温、最低温

该项目涉及的技术点如下：

1.1 样式表的设置

合理地使用样式表，可以使界面更加美观，这里设置的样式表如下：

- 背景图片

为整个窗体设置一张背景图片

- 背景色

设置控件背景透明，或者设置一个透明度

- 圆角

为控件设置圆角

- 字体颜色和大小

为控件设置合适的字体颜色和字体大小

1.2 JSON 数据格式

`HTTP` 服务端返回的天气数据，是 `JSON` 格式

使用 Qt 提供的，解析 `JSON` 相关的类，可以很方便地解析出其中的字段

1.3 HTTP 请求

根据 `HTTP` 服务端提供的接口，发送 `HTTP` 请求，获取天气数据

1.4 事件

为了界面的美观，我们将窗口设置为无标题栏，这样就无法通过右上角的【关闭】按钮，退出程序。

因此增加了右键菜单退出的功能

还重写了鼠标移动事件，让窗口可以跟随鼠标移动

1.5 绘图

绘制高低温曲线，根据每天高低温数据，可以绘制一个曲线，更直观地展示温度变化趋势

1.6 资源文件

根据不同的天气类型，还可以用不同的图标进行展示，更加直观

而这些图标通常会放到资源文件中，这样它们可以一同被打包进 Qt 的可执行程序中

2. JSON 基础知识

2.1 什么是 JSON?

2.1.1 JSON 的引入

在讲解 `JSON` 之前，首先想这么一种场景：

通常，客户端和服务端要进行通信，那么通信采用什么数据格式呢？

比如 `C++` 写的服务端，创建了一个 `Person` 对象：

```
1 class Person {  
2     string name;  
3     string gender;  
4     int age;  
5 };
```

怎么将服务端创建的 `Person` 对象，传递到客户端呢？

直接传 `Person` 对象肯定是不合适的，因为客户端可能甚至不是 `C++` 写的，可能是 `Java` 写的，`Java` 不认识 `C++` 中的对象。

更有甚者，客户端是一个单片机的设备，是用 `C` 语言写的，`C` 语言是面向过程的，压根就没有类和对象的概念

此时就需要一种通用的数据格式，`JSON` 就应运而生了。

`JSON`（JavaScript Object Notation），中文名 **JS对象表示法**，因为它和 `JS` 中的对象的写法很类似

通常说的 `JSON`，其实就是 `JSON` 字符串，本质上是一种特殊格式的字符串。

`JSON` 是一种轻量级的数据交换格式，客户端和服务端数据交互，基本都是 `JSON` 格式的

2.1.2 JSON 的特点

相对于其他数据传输格式，`JSON` 有以下特点

（1）便于阅读和书写

除了 `JSON` 格式，还有一种数据传输格式 `XML`，相对于 `XML`，`JSON` 更加便于阅读和书写

（2）独立于编程语言

`JSON` 完全独立于编程语言，并不是说名字里有 `JavaScript`，就只能在 `JavaScript` 中使用，不是这样的

`JSON` 和 `JavaScript` 的关系，就类似于：雷锋和雷峰塔、周杰和周杰伦、范伟和范玮琪，可以说没啥关系

几乎在所有的编程语言和开发环境中，都有解析和生成 `JSON` 字符串的库，比如：

常见的 JSON 库

✓ C 语言

Jansson、cJSON

✓ C++

jsonCpp、JSON for Modern C++

✓ Java

json-lib、org-json

✓ Android

GSON、FastJson

✓ QT

QJsonXxx

(3) 网络传输的标准数据格式

基于以上的特点，JSON 成为网络传输使用率最高的数据格式

2.2 JSON 的两种数据格式

JSON 有两种数据格式

- JSON 对象
- JSON 数组

规则：被大括号包裹的是 JSON 对象；被中括号包裹的是 JSON 数组

2.2.1 JSON 数组

JSON 数组格式

[元素1, 元素2, 元素3, ... 元素n]

类似于 C/C++ 中的数组，元素之间以逗号分隔

不同的是，JSON 数组中的元素可以是不同的数据类型，包括：整形、浮点、字符串、布尔类型、JSON 数组、JSON 对象、空值

(1) JSON 数组中的元素是同一类型

```
1 // 元素类型都是数字
2 [1, 2, 3, 4]
3
4 // 元素类型都是字符串
5 ["Spring", "Summer", "Autumn", "Winter"]
```

(2) JSON 数组中的元素是不同类型

```
1 // 元素类型分别是：整型、浮点、字符串、布尔、空置
2 [1, 2.5, "hello", true, false, null]
```

(3) JSON 数组的嵌套

```
1 // 数组中嵌套数组
2 [
3     [1, 2, 3, 4],
4     ["Spring", "Summer", "Autumn", "Winter"],
5     [1, 2.5, "hello", true, false, null]
6 ]
```

(4) JSON 数组嵌套 JSON 对象

```
1 // 数组中嵌套对象
2 [
3     {
4         "name": "Tom",
5         "age": 18,
6         "gender": "male"
7     },
8     {
9         "name": "Tom",
10        "age": 18,
```

```
11     "gender": "male"
12   }
13 ]
```

2.2.2 JSON 对象



JSON 对象格式

```
{
  "key1": value1,
  "key2": value2,
  "key3": value3
}
```

JSON 对象内部使用键值对的方式来组织；

键和值之间使用冒号分隔，多个键值之间使用逗号分隔；

键是字符串类型，值的类型可以是：整形、浮点、字符串、布尔类型、**JSON** 数组、**JSON** 对象、空值

(1) 最简单的 JSON 对象

```
1 {
2   "name": "Tom",
3   "age": 18,
4   "gender": "male"
5 }
```

(2) JSON 对象和 JSON 数组嵌套

JSON 对象中，还可以嵌套 **JSON** 对象和 **JSON** 数组

```
1 {
2   "name": "China",
3   "info": {
4     "capital": "beijing",
5     "asian": true,
6     "founded": 1949
7   },
8   "provinces": [{
```

```
9      "name": "shandong",
10     "capital": "jinan"
11   }, {
12     "name": "zhejiang",
13     "capital": "hangzhou"
14   }]
15 }
```

2.3 JSON 在线解析

JSON 本质就是一种特殊格式的字符串

实际工作中，这个 **JSON** 字符串可能是自己手写的，也可能是来自网络接收的数据

如下是一段 **JSON** 字符串，它可能是我们自己写的，也可能是服务端返回的

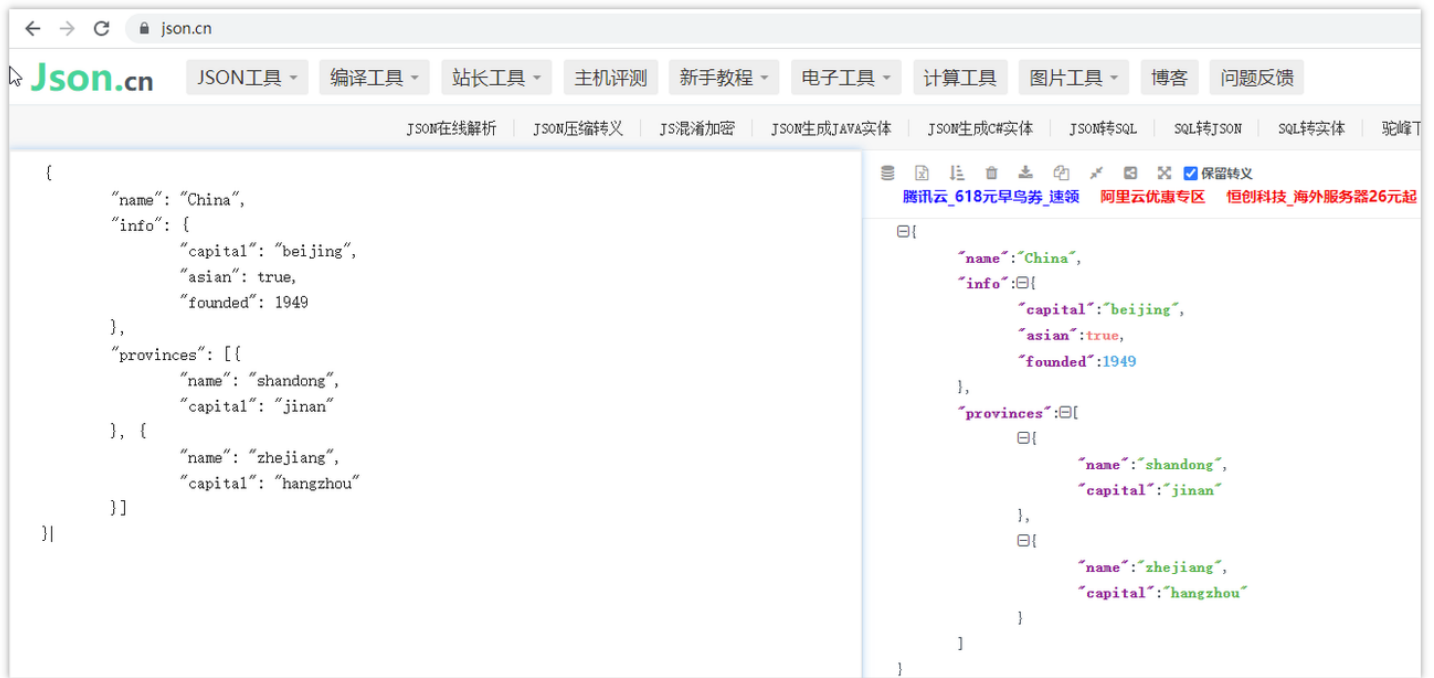
它是**压缩格式**，也就是没有换行和缩进，不方便判断格式是否正确

```
1 {"name":"China","info":
  {"capital":"beijing","asian":true,"founded":1949},"provinces":
  [{"name":"shandong","capital":"jinan"},
  {"name":"zhejiang","capital":"hangzhou"}]}
```

那么有没有一种简单的方式，来校验这个 **JSON** 的格式是否正确呢？

答： **JSON** 在线解析工具

在浏览器中，搜索【JSON 在线解析】，有很多网站提供 **JSON** 在线解析功能，如下：



3. Qt 中使用 JSON

从 Qt 5.0 开始提供了对 JSON 的支持，使用 Qt 提供的 QJson 类，可以很方便地生成 JSON 字符串，以及解析 JSON 字符串。

3.1 JSON 相关的类

Qt 提供的与 JSON 相关的类，主要有四个：

- QJsonObject
- QJsonArray
- QJsonValue
- QJsonDocument

下面依次介绍这几个类

3.1.1 QJsonObject

QJsonObject 封装了 JSON 中的对象，可以存储多个键值对。

其中，键为字符串类型，值为 QJsonValue 类型。

- 创建一个 QJsonObject 对象

```
1 QJsonObject::QJsonObject();
```

- 将键值对添加到 `QJsonObject` 对象中

```
1 QJsonObject::iterator insert(const QString &key, const QJsonValue &value);
```

- 获取 `QJsonObject` 对象中键值对的个数

```
1 int QJsonObject::count() const;  
2 int QJsonObject::size() const;  
3 int QJsonObject::length() const;
```

- 通过 `key` 得到 `value`

```
1 QJsonValue QJsonObject::value(const QString &key) const;  
2 QJsonValue QJsonObject::operator[](const QString &key) const;
```

- 检查 `key` 是否存在

```
1 iterator QJsonObject::find(const QString &key);  
2 bool QJsonObject::contains(const QString &key) const;
```

- 遍历 `key`

```
1 QStringList QJsonObject::keys() const;
```

3.1.2 QJsonArray

`QJsonArray` 封装了 `Json` 中的数组。数组中元素的类型统一为 `QJsonValue` 类型

- 创建一个 `QJsonArray`

```
1 QJsonArray::QJsonArray();
```

- 添加数组元素

```
1 // 添加到头部和尾部
2 void QJsonArray::append(const QJsonValue &value);
3 void QJsonArray::prepend(const QJsonValue &value);
4
5 // 插入到 i 的位置之前
6 void QJsonArray::insert(int i, const QJsonValue &value);
7
8 // 添加到头部和尾部
9 void QJsonArray::push_back(const QJsonValue &value);
10 void QJsonArray::push_front(const QJsonValue &value);
```

- 获取 `QJsonArray` 中元素个数

```
1 int QJsonArray::count() const;
2 int QJsonArray::size() const;
```

- 获取元素的值

```
1 // 获取头部和尾部
2 QJsonValue QJsonArray::first() const;
3 QJsonValue QJsonArray::last() const;
4
5 // 获取指定位置
6 QJsonValue QJsonArray::at(int i) const;
```

```
7 QJsonValueRef QJsonArray::operator[](int i);
```

- 删除元素

```
1 // 删除头部和尾部
2 void QJsonArray::pop_back();
3 void QJsonArray::pop_front();
4
5 void QJsonArray::removeFirst();
6 void QJsonArray::removeLast();
7
8 // 删除指定位置
9 void QJsonArray::removeAt(int i);
10 QJsonValue QJsonArray::takeAt(int i);
```

3.1.3 QJsonValue

它封装了 **JSON** 支持的六种数据类型，分别为：

数据类型	Qt 类
布尔类型	QJsonValue::Bool
浮点类型（包括整形）	QJsonValue::Double
字符串类型	QJsonValue::String
数组类型	QJsonValue::Array
对象类型	QJsonValue::Object
空值类型	QJsonValue::Null

可以通过以下方式构造 **QJsonValue** 对象

```
1 // 字符串
2 QJsonValue(const char *s);
3 QJsonValue(QLatin1String s);
4 QJsonValue(const QString &s);
5
6 // 整形 and 浮点型
7 QJsonValue(qint64 v);
```

```
8 QJsonValue(int v);
9 QJsonValue(double v);
10
11 // 布尔类型
12 QJsonValue(bool b);
13
14 // Json对象
15 QJsonValue(const QJsonObject &o);
16
17 // Json数组
18 QJsonValue(const QJsonArray &a);
19
20 // 空值类型
21 QJsonValue(QJsonValue::Type type = Null);
```

如果已经得到了一个 `QJsonValue` 对象，如何判断其内部封装的是什么类型的数据呢？

答：使用以下判断函数

```
1 // 是否是字符串类型
2 bool isString() const;
3
4 // 是否是浮点类型(整形也是通过该函数判断)
5 bool isDouble() const;
6
7 // 是否是布尔类型
8 bool isBool() const;
9
10 // 是否是Json对象
11 bool isObject() const;
12
13 // 是否是Json数组
14 bool isArray() const;
15
16 // 是否是未定义类型(无法识别的类型)
17 bool isUndefined() const;
18
19 // 是否是空值类型
20 bool isNull() const;
```

通过以上判断函数，获取到其内部数据的实际类型之后，如果有需求就可以再次将其转换为对应的基础数据类型，对应的 **API** 函数如下：

```
1 // 转换为字符串类型
2 QString toString() const;
3 QString toString(const QString &defaultValue) const;
4
5 // 转换为浮点类型
6 double toDouble(double defaultValue = 0) const;
7 // 转换为整形
8 int toInt(int defaultValue = 0) const;
9
10 // 转换为布尔类型
11 bool toBool(bool defaultValue = false) const;
12
13 // 转换为Json对象
14 QJsonObject toObject(const QJsonObject &defaultValue) const;
15 QJsonObject toObject() const;
16
17 // 转换为Json数组
18 QJsonArray toArray(const QJsonArray &defaultValue) const;
19 QJsonArray toArray() const;
```

3.1.4 QJsonDocument

它封装了一个完整的 **JSON** 文档。

它可以从 **UTF-8** 编码的基于文本的表示，以及 Qt 本身的二进制格式，读取和写入该文档。

QJsonObject 和 **QJsonArray** 这两个对象是不能直接转换为字符串类型的，需要通过 **QJsonDocument** 类来完成二者的转换

下面介绍转换的步骤

(1) **QJsonObject** / **QJsonArray** => 字符串

```
1 // 1. 创建 QJsonDocument 对象
2 // 以 QJsonObject 或者 QJsonArray 为参数来创建 QJsonDocument 对象
3 QJsonDocument::QJsonDocument(const QJsonObject &object);
4 QJsonDocument::QJsonDocument(const QJsonArray &array);
5
6 // 2. 将 QJsonDocument 对象中的数据进行序列化
```

```

7 // 通过调用 toXXX() 函数就可以得到文本格式或者二进制格式的 Json 字符串了。
8 QByteArray QJsonDocument::toBinaryData() const; // 二
    进制格式的json字符串
9 QByteArray QJsonDocument::toJson(JsonFormat format = Indented) const; // 文
    本格式
10
11 // 3. 使用得到的字符串进行数据传输, 或者保存到文件

```

(2) 字符串 => QJsonObject / QJsonArray

通常, 通过网络接收或者读取磁盘文件, 会得到一个 **JSON** 格式的字符串, 之后可以按照如下步骤, 解析出 **JSON** 字符串中的一个字段

```

1 // 1. 将 JSON 字符串转换为 QJsonDocument 对象
2 [static] QJsonDocument QJsonDocument::fromBinaryData(const QByteArray &data,
    DataValidation validation = Validate);
3 [static] QJsonDocument QJsonDocument::fromJson(const QByteArray &json,
    QJsonParseError *error = Q_NULLPTR);
4
5 // 2. 将文档对象转换为 json 数组 / 对象
6
7 // 2.1 判断文档对象中存储的数据, 是 JSON 数组还是 JSON 对象
8 bool QJsonDocument::isArray() const;
9 bool QJsonDocument::isObject() const
10
11 // 2.2 之后, 就可以转换为 JSON 数组或 JSON 对象
12 QJsonObject QJsonDocument::object() const;
13 QJsonArray QJsonDocument::array() const;
14
15 // 3. 调用 QJsonArray / QJsonObject 类提供的 API 获取存储在其中的数据

```

3.2 构建 JSON 字符串

在网络传输时, 通常是传输的 **JSON** 字符串

传输之前, 首先要生成 **JSON** 字符串, 接下来使用 Qt 提供的工具类, 来生成如下格式的 **JSON** 字符串

```

1 {
2     "name": "China",
3     "info": {
4         "capital": "beijing",

```

```
5     "asian": true,
6     "founded": 1949
7 },
8 "provinces": [{
9     "name": "shandong",
10    "capital": "jinan"
11 }, {
12     "name": "zhejiang",
13     "capital": "hangzhou"
14 }]
15 }
```

接下来封装一个 `writeJson()` 函数实现以上功能，如下：

```
1 void writeJson() {
2     QJsonObject rootObj;
3
4     // 1. 插入 name 字段
5     rootObj.insert("name", "China");
6
7     // 2. 插入 info 字段
8     QJsonObject infoObj;
9     infoObj.insert("capital", "beijing");
10    infoObj.insert("asian", true);
11    infoObj.insert("founded", 1949);
12    rootObj.insert("info", infoObj);
13
14    // 3. 插入 province 字段
15    QJsonArray provinceArray;
16
17    QJsonObject provinceSdObj;
18    provinceSdObj.insert("name", "shandong");
19    provinceSdObj.insert("capital", "jinan");
20
21    QJsonObject provinceZjObj;
22    provinceZjObj.insert("name", "zhejiang");
23    provinceZjObj.insert("capital", "hangzhou");
24
25    provinceArray.append(provinceSdObj);
26    provinceArray.append(provinceZjObj);
27
28    rootObj.insert("provinces", provinceArray);
29
30    // 4. 转换为 json 字符串
```

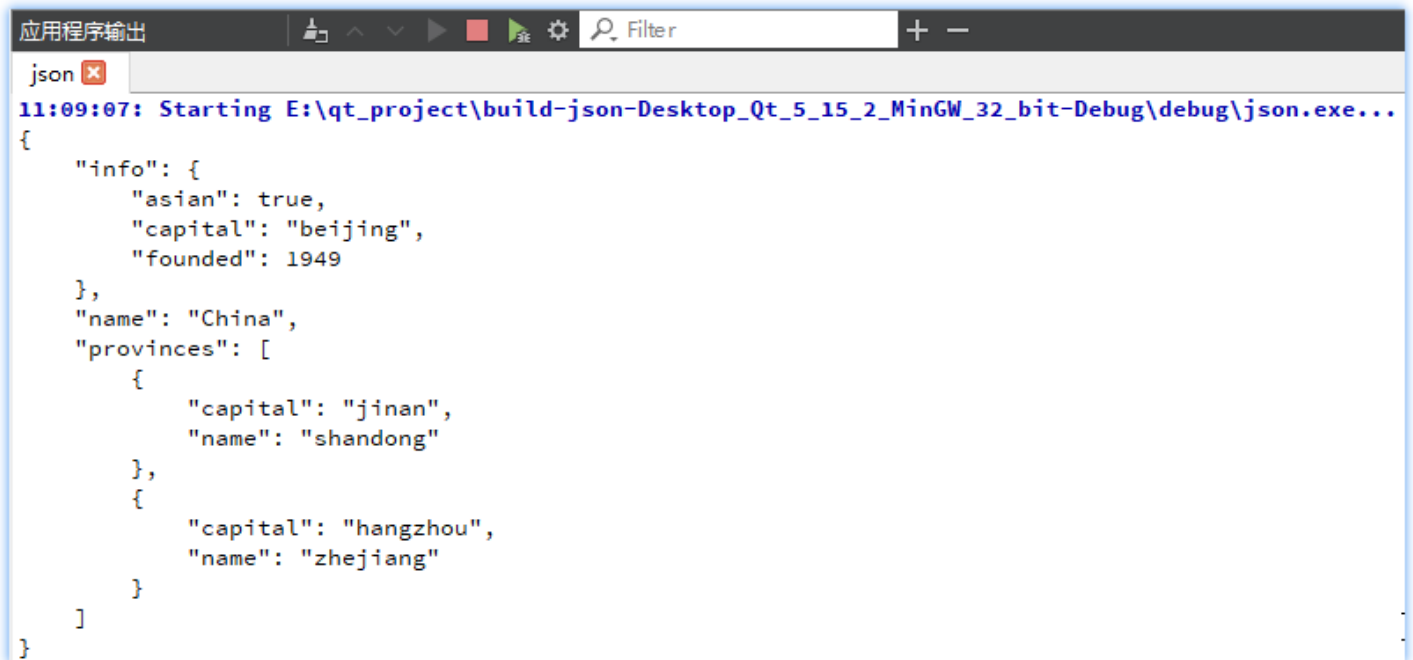


```

31     QJsonDocument doc(rootObj);
32     QByteArray json = doc.toJson();
33
34     // 5.1 打印json字符串
35     qDebug() << QString(json).toUtf8().data();
36
37     // 5.2 将json字符串写到文件
38     QFile file("d:\\china.json");
39     file.open(QFile::WriteOnly);
40     file.write(json);
41     file.close();
42 }

```

运行程序，就可以在控制台输出构建出的 `JSON` 字符串，如下：



The screenshot shows the Qt console output for the application. The title bar reads '应用程序输出' (Application Output). The output text is as follows:

```

11:09:07: Starting E:\qt_project\build-json-Desktop_Qt_5_15_2_MinGW_32_bit-Debug\debug\json.exe...
{
  "info": {
    "asian": true,
    "capital": "beijing",
    "founded": 1949
  },
  "name": "China",
  "provinces": [
    {
      "capital": "jinan",
      "name": "shandong"
    },
    {
      "capital": "hangzhou",
      "name": "zhejiang"
    }
  ]
}

```

并且，将生成的 `JSON` 字符串，保存到了 `china.json` 文件：



3.3 解析 JSON 字符串

通常接收网络数据的格式是 **JSON** 格式

在接收完毕之后，需要解析出其中的每一个字段，根据各个字段的值做相应的显示或者其他处理

接下来使用 **Qt** 提供的工具类，读取文件中的 **JSON** 字符串，把其中的字段解析出来：

文件内容：

```
1 {
2     "name": "China",
3     "info": {
4         "capital": "beijing",
5         "asian": true,
6         "founded": 1949
7     },
8     "provinces": [{
9         "name": "shandong",
10        "capital": "jinan"
11    }, {
12        "name": "zhejiang",
13        "capital": "hangzhou"
14    }]
15 }
```

接下来封装一个 **fromJson()** 函数实现以上功能，如下：

```

1 void fromJson() {
2     QFile file("d:\\china.json");
3     file.open(QFile::ReadOnly);
4     QByteArray json = file.readAll();
5     file.close();
6
7     QJsonDocument doc = QJsonDocument::fromJson(json);
8     if (!doc.isObject()) {
9         qDebug() << "not an object";
10        return;
11    }
12
13    QJsonObject obj = doc.object();
14    QStringList keys = obj.keys();
15    for (int i = 0; i < keys.size(); ++i) {
16        // 获取key-value
17        QString key = keys[i];
18        QJsonValue value = obj.value(key);
19
20        if (value.isBool()) {
21            qDebug() << key << ":" << value.toBool();
22        } else if (value.isString()) {
23            qDebug() << key << ":" << value.toString();
24        } else if (value.isDouble()) {
25            qDebug() << key << ":" << value.toInt();
26        } else if (value.isObject()) {
27            qDebug() << key << ":";
28
29            QJsonObject infoObj = value.toObject();
30            #if 0
31            // 简便起见, 直接使用键来取值
32            QString capital = infoObj["capital"].toString();
33            bool asian = infoObj["asian"].toBool();
34            int founded = infoObj["founded"].toInt();
35
36            qDebug() << "    "
37                    << "capital"
38                    << ":" << capital;
39            qDebug() << "    "
40                    << "asian"
41                    << ":" << asian;
42            qDebug() << "    "
43                    << "founded"
44                    << ":" << founded;
45            #else
46            QStringList infoKeys = infoObj.keys();

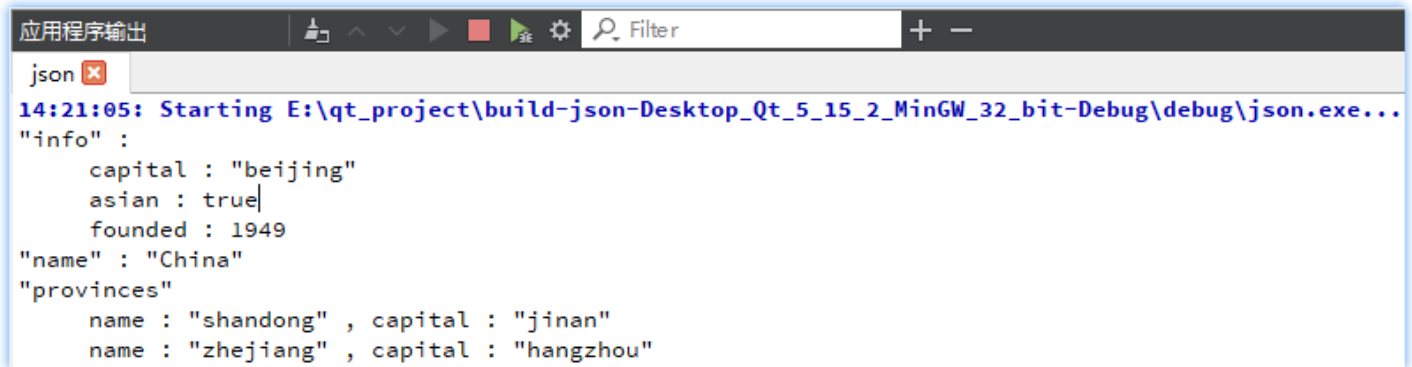
```

```

47         for (int i = 0; i < infoKeys.size(); i++) {
48             QJsonValue v = infoObj.value(infoKeys[i]);
49             if (v.isBool()) {
50                 qDebug() << "    " << infoKeys[i] << ":" << v.toBool();
51             } else if (v.isDouble()) {
52                 qDebug() << "    " << infoKeys[i] << ":" << v.toInt();
53             } else if (v.isString()) {
54                 qDebug() << "    " << infoKeys[i] << ":" << v.toString();
55             }
56         }
57     #endif
58     } else if (value.isArray()) {
59         qDebug() << key;
60
61         QJsonArray provinceArray = value.toArray();
62     #if 1
63         // 简便起见, 直接使用键来取值
64         for (int i = 0; i < provinceArray.size(); i++) {
65             QJsonObject provinceObj = provinceArray[i].toObject();
66
67             QString name = provinceObj["name"].toString();
68             QString capital = provinceObj["capital"].toString();
69
70             qDebug() << "    "
71                     << "name"
72                     << ":" << name << ", capital"
73                     << ":" << capital;
74         }
75     #else
76         for (int i = 0; i < provinceArray.size(); i++) {
77             QJsonObject provinceObj = provinceArray[i].toObject();
78             QStringList provinceKeys = provinceObj.keys();
79             for (int j = 0; j < provinceKeys.size(); j++) {
80                 // 简便起见, 不再进行类型判断
81                 QJsonValue infoValue = provinceObj.value(provinceKeys[j]);
82                 qDebug() << "    " << provinceKeys[j] << ":" <<
infoValue.toString();
83             }
84         }
85     #endif
86     }
87 }
88 }

```

运行程序，在控制台输出解析出的字段

A screenshot of a Qt console window titled '应用程序输出' (Application Output). The window shows the output of a program that has just started, with the path 'E:\qt_project\build-json-Desktop_Qt_5_15_2_MinGW_32_bit-Debug\debug\json.exe...'. The output is a JSON object representing information about China. The JSON is formatted with indentation and line breaks. The fields include 'info' (with 'capital', 'asian', and 'founded'), 'name', and 'provinces' (an array of objects with 'name' and 'capital').

```
14:21:05: Starting E:\qt_project\build-json-Desktop_Qt_5_15_2_MinGW_32_bit-Debug\debug\json.exe...
{"info": {"capital": "beijing", "asian": true, "founded": 1949}, "name": "China", "provinces": [{"name": "shandong", "capital": "jinan"}, {"name": "zhejiang", "capital": "hangzhou"}]}
```

4. HTTP 基础知识

4.1 HTTP 必知必会

既然是实现 HTTP 协议的天气预报，那么 HTTP 相关的知识，必须掌握

HTTP：超文本传输协议（HyperText Transfer Protocol）

HTTP 是浏览器端 Web 通信的基础。

关于 HTTP 的介绍，不论是书本还是网上的博客，介绍的过于学术化，这里以实战出发，介绍几个关键的点：

4.1.1 两种架构

- B/S 架构

Browser/Server，浏览器/服务器架构

B：浏览器，比如 Firefox、Internet Explorer、Google Chrome、Safari、Opera 等

S：服务器，比如 Apache、nginx 等

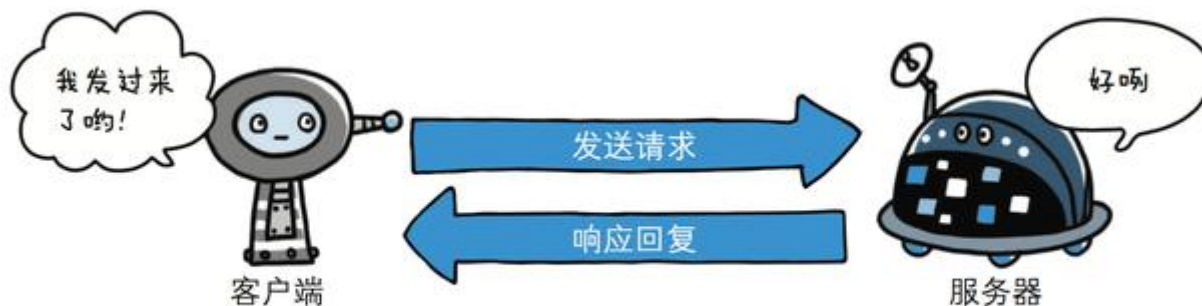
- C/S 架构

Client/Server，客户端/服务器架构

B/S 架构相对于 C/S 架构，客户机上无需安装任何软件，使用浏览器即可访问服务器

因此，越来越多的 C/S 架构正被 B/S 架构所替代

4.1.2 基于请求响应的模式



图：请求必定由客户端发出，而服务器端回复响应

HTTP 协议永远都是客户端发起请求，服务器做出响应

也就是说，请求必定是先从客户端发起的，服务器端在没有接收到请求之前不会发送任何响应

这就无法实现这样一种场景：服务端主动推送消息给客户端

4.1.3 无状态

当浏览器第一次发送请求给服务器时，服务器做出了响应；

当浏览器第二次发送请求给服务器时，服务器同样可以做出响应，但服务器并不知道第二次的请求和第一次来自同一个浏览器

也就是说，服务器是不会记住你是谁，所以是无状态的。

而如果要使 **HTTP** 协议有状态，就可以使浏览器访问服务器时，加入 **cookie**

这样，只要你在请求时有了这个 **cookie**，服务器就能够通过 **cookie** 知道，你就是之前那个浏览器

这样的话，**HTTP** 协议就有状态了。

4.1.4 请求报文

请求报文由四部分组成：

请求行 + 请求头（请求首部字段） + 空行 + 实体

（1）请求行

请求行里面有：

- 请求方法：比如 **GET** 、 **POST**
- 资源对象（ **URI** ）
- 协议名称和版本号（ **HTTP/1.1** ）

```
1 POST /custom/a2873925c34ecbd2/web/cstm?stm=1649149234039 HTTP/1.1
2
3 POST                                     即请求方法
4 /custom/a2873925c34ecbd2/web/cstm?stm=1649149234039 即 URL
5 HTTP/1.1                                即协议和版本
```

(2) 请求头

请求首部字段，请求头由于告诉服务器该请求的一些信息，起到传递额外信息的目的

```
1 Host: api.growingio.com
2 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:91.0)
  Gecko/20100101 Firefox/91.0
3 Accept: */*
4 Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
5 Accept-Encoding: gzip, deflate, br
6 Content-Length: 264
7 Origin: https://leetcode-cn.com
8 Sec-Fetch-Dest: empty
9 Sec-Fetch-Mode: no-cors
10 Sec-Fetch-Site: cross-site
11 Referer: https://leetcode-cn.com/
```

(3) 空行

用来区分开请求头和请求实体。

(4) 请求实体

请求实体即真正所需要传输的数据

4.1.5 响应报文

响应报文同样是由四部分组成：

状态行 + 响应头 + 空行 + 消息体

(1) 状态行

状态行主要由三部分组成：

- HTTP 版本
- 状态码（表示相应的结果）
- 原因短语（解释）

```
1 HTTP/2 200 OK
2
3 HTTP/2          协议和版本
4 200            状态码 - 200代表OK，表示请求成功，404代表NOT FOUND，表示请求失败，所请
                  求资源未在服务器上发现。
5 OK             原因短语
```

(2) 响应头

响应报文首部，和请求报文首部一样，响应报文首部同样是为了传递额外信息，例如：

```
1 date: Tue, 05 Apr 2022 10:48:17 GMT           // 响应时间
2 content-type: application/json                // 响应格式
3 content-length: 127                          // 长度
4 strict-transport-security: max-age=31536000
5 X-Firefox-Spdy: h2
```

(3) 空行

用来区分开响应实体和响应首部

(4) 响应实体

真正存储响应信息的部分

4.1.6 请求方式

HTTP 常用的请求方式有很多中，最常用的是 GET 和 POST

二者最主要的区别就是：

GET 请求的参数位于 URL 中，会显示在地址栏上

POST 请求的参数位于 request body 请求体中。

因此，GET 请求的安全性不如 POST 请求，并且 GET 请求的参数有长度限制，而 POST 没有

4.2 Postman

HTTP 包含客户端和服务端，在讲解 **Postman** 之前，试想下面的两种情况

- 服务端开发完毕，而客户端还未完成开发。此时服务端开发人员能否对自己写的服务端程序进行测试呢？
- 客户端开发人员访问服务端出错，比如无法访问服务端，有没有第三方的测试工具做进一步验证呢？

此时，就用到 **Postman** 了

Postman 是一个接口测试工具，使用 **HTTP** 开发的人几乎是无人不知的

它主要是用来模拟各种 **HTTP** 请求的（比如 **GET** 请求、**POST** 请求等）

在做接口测试的时候，**Postman** 相当于客户端，它可模拟用户发起的各类 **HTTP** 请求，将请求数据发送至服务端，并获取对应的响应结果

接下来，以获取天气北京的天气为例，来介绍 **Postman** 的基本使用

4.2.1 安装

首先要安装 **Postman**

登录 www.postman.com 根据自己的系统，下载对应的版本，并一步步傻瓜式安装即可

安装成功后，会提示要求注册一个账号：



Create an account or sign in

Create Free Account

Sign In

Create your account or sign in later? [Skip and go to the app](#)

直接忽略即可

A free Postman account lets you

- ✓ Organize all your API development in workspaces
- ✓ Create public workspaces to collaborate with over 10 million developers
- ✓ Back up your work on Postman's cloud
- ✓ Experience the best API development platform for free!



在此，既可以根据提示免费注册一个账号，也可以不注册，点击左下角的【Skip】直接忽略即可

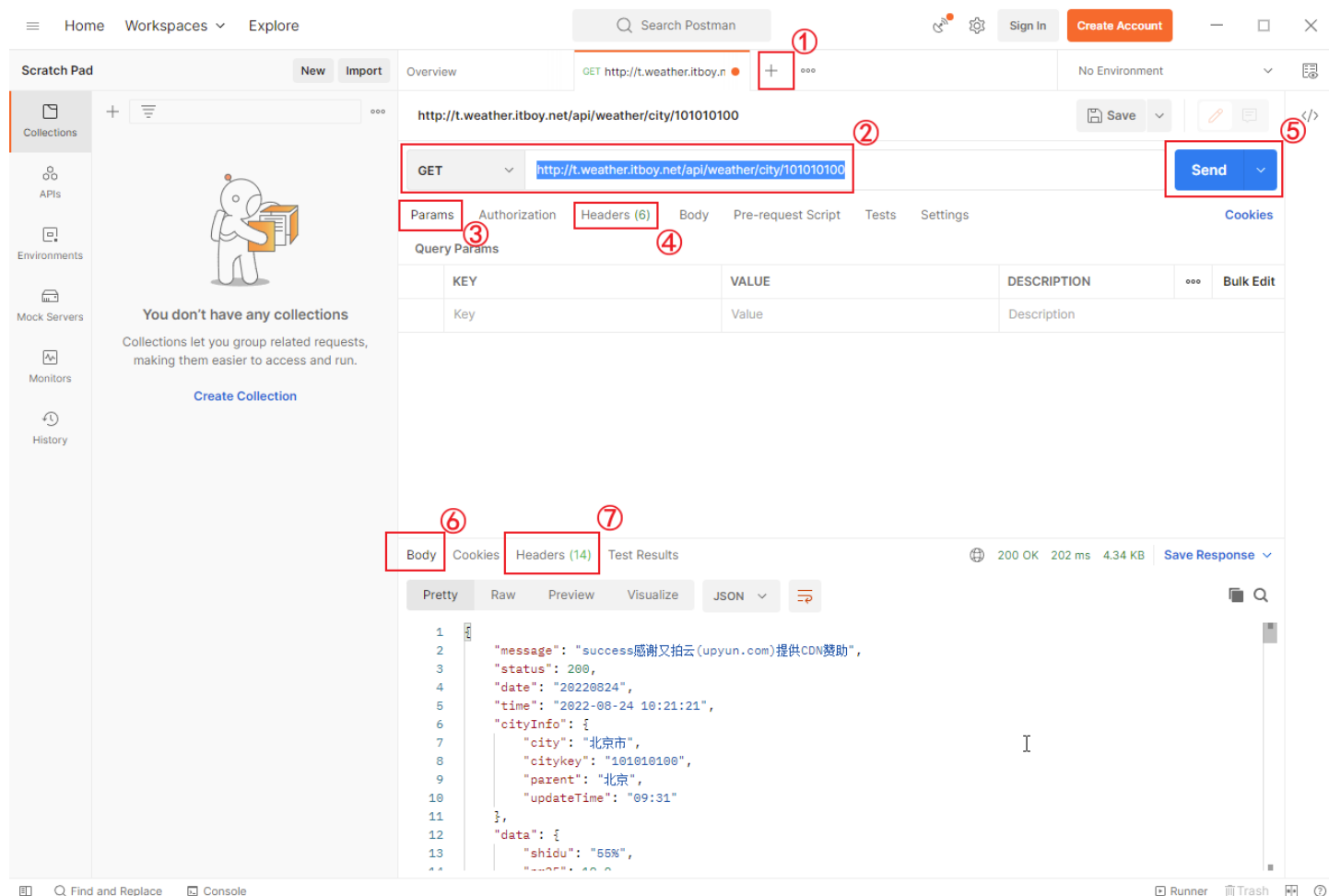
4.2.2 发送请求、获取响应

安装成功后，就可以使用它来发送请求了

这里以获取北京的天气为例

获取北京天气的 **URL** 为：<http://t.weather.itboy.net/api/weather/city/101010100>

其中，**101010100** 是北京的城市编码，是 **9** 位的



上图的序号就是操作的具体步骤：

- ① 点击 “+”，即可打开一个用于发送请求的标签页
- ② 选择请求的方法（GET / POST），并输入请求的 URL
- ③ 点击【Params】，可以添加参数
- ④ 点击【Headers】，可以添加请求头
- ⑤ 点击【Send】，发送该请求到服务器
- ⑥ Body，响应体，也就是服务器返回的实际数据。响应体中选择右侧的【JSON】格式，并选择【Pretty】，可以对 JSON 数据做美化显示
- ⑦ Headers，响应头，也就是服务器返回的响应头数据

5. Qt 实现 HTTP 请求

以上是使用 Postman 来发送 HTTP 请求，那么在 Qt 中如何实现 HTTP 请求呢？

Qt 框架提供了专门的类，可以方便地实现 HTTP 请求

5.1 创建“网络访问管理”对象

首先需要创建一个 `QNetworkAccessManager` 对象，这是 Qt 中进行 `HTTP` 请求的开端

```
1 mNetAccessManager = new QNetworkAccessManager(this);
```

5.2 关联信号槽

在发送 `HTTP` 请求之前，先关联信号槽

```
1 // 获取到数据之后
2 connect(mNetAccessManager, &QNetworkAccessManager::finished, this,
    &MainWindow::onReplied);
```

当请求结束，获取到服务器的数据时，`mNetAccessManager` 会发射一个 `finished` 信号，该信号携带一个 `QNetworkReply` 的参数

服务器返回的所有数据就封装在其中，通过 `QNetworkReply` 类提供的各种函数，就可以获取响应头，响应体等各种数据。

5.3 发送请求

接下来就可以发送 `HTTP` 请求了

```
1 QUrl url("http://t.weather.itboy.net/api/weather/city/101010100");
2 mNetAccessManager->get(QNetworkRequest(url));
```

根据请求的地址构建出一个 `QUrl` 对象，然后直接调用 `QNetworkAccessManager` 的 `get` 函数，即可发送一个 `GET` 请求

5.4 接收数据

由于上面绑定了信号槽，服务器返回数据后，自动调用我们自定义的槽函数 `onReplied`

如下是 `onReplied` 函数的标准写法：

```
1 void MainWindow::onReplied(QNetworkReply* reply)
```

```

2 {
3     // 响应的状态码为200, 表示请求成功
4     int status_code = reply-
    >attribute(QNetworkRequest::HttpStatusCodeAttribute).toInt();
5
6     qDebug() << "operation:" << reply->operation();           // 请求方式
7     qDebug() << "status code:" << status_code;                // 状态码
8     qDebug() << "url:" << reply->url();                          // url
9     qDebug() << "raw header:" << reply->rawHeaderList();        // header
10
11     if ( reply->error() != QNetworkReply::NoError || status_code != 200 ) {
12         QMessageBox::warning(this, "提示", "请求数据失败!", QMessageBox::Ok);
13     } else {
14         //获取响应信息
15         QByteArray reply_data = reply->readAll();
16         QByteArray byteArray = QString(reply_data).toUtf8();
17         qDebug() << "read all:" << byteArray.data();
18
19         // parseJson()
20     }
21
22     reply->deleteLater();
23 }

```

可见，`QNetworkReply` 中封装了服务器返回的所有数据，包括响应头、响应的状态码、响应体等。

6. 新建工程，右键退出

本节课开始，从零一步步实现这个天气预报项目

6.1 新建工程

新建一个工程，命名为 `WeatherForecast`，窗口类继承自 `QMainWindow`，如下：

Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

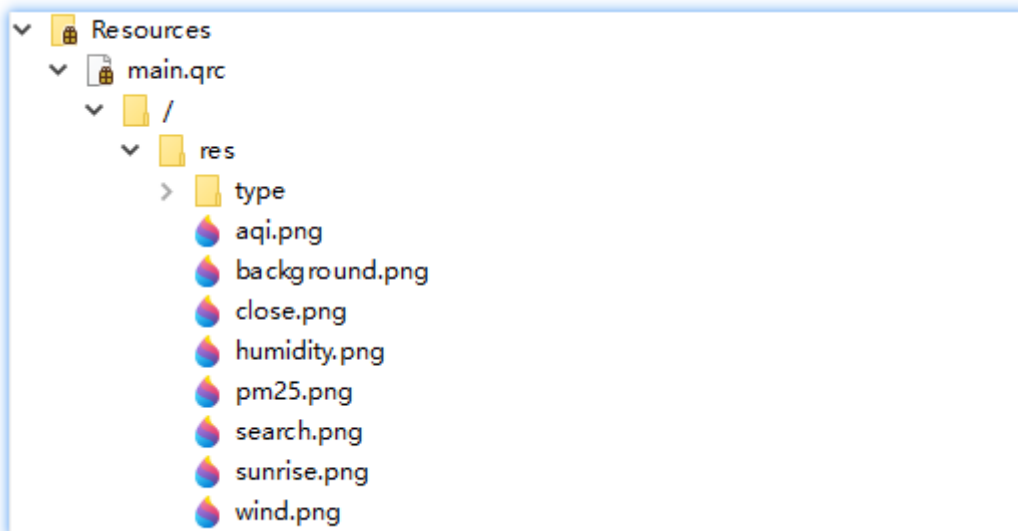
Class name:	<input type="text" value="MainWindow"/>
Base class:	<input type="text" value="QMainWindow"/>
Header file:	<input type="text" value="mainwindow.h"/>
Source file:	<input type="text" value="mainwindow.cpp"/>
	<input checked="" type="checkbox"/> Generate form
Form file:	<input type="text" value="mainwindow.ui"/>

6.2 添加资源文件

由于后边会窗口背景图片，以及各种天气的图标，这里创建一个资源文件，把这些图片图标添加到工程中来

在工程中添加资源文件之前已经详细讲解过了，这里不再赘述

添加之后的效果如下：



6.3 设置窗口固定大小，无边框


首先，在属性窗口中，设置窗口的大小 `800*450`

属性	值
▼ QObject	
objectName	MainWindow
▼ QWidget	
windowModality	NonModal
enabled	<input checked="" type="checkbox"/>
▼ geometry	[(0, 0), 800 x 450]
X	0
Y	0
宽度	800
高度	450

然后，在代码中增加以下两行代码，即可设置窗口为固定大小，且无边框：

```
1 MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new
  Ui::MainWindow)
2 {
3     ui->setupUi(this);
4
5     setWindowFlag(Qt::FramelessWindowHint);    // 无边框
6     setFixedSize(width(), height());           // 固定窗口大小
7 }
```

6.4 右键退出

由于没有了边框，就无法点击右上角的  退出程序，因此添加右键菜单的退出功能

要实现右键菜单，只需重写父类的 `contextMenuEvent` 虚函数即可：

```
1 void QWidget::contextMenuEvent(QContextMenuEvent *event)
```

具体步骤为：

首先，在 `mainwindow.h` 头文件中，声明该虚函数，并添加菜单

```
1 class MainWindow : public QMainWindow {
2
3 protected:
```

```

4     void contextMenuEvent(QContextMenuEvent* event);
5
6 private:
7     QMenu* mExitMenu;    // 右键退出的菜单
8     QAction* mExitAct;   // 退出的行为
9 };

```

然后，来到 `mainwindow.cpp` 源文件，做如下实现：

```

1 #include <QContextMenuEvent>
2
3 MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new
  Ui::MainWindow)
4 {
5     // 右键菜单：退出程序
6     mExitMenu = new QMenu(this);
7     mExitAct = new QAction();
8     mExitAct->setText(tr("退出"));
9     mExitAct->setIcon(QIcon(":/res/close.ico"));
10    mExitMenu->addAction(mExitAct);
11
12    connect(mExitAct, &QAction::triggered, this, [=]() {
13        qApp->exit(0);
14    });
15 }
16
17 // 重写父类的虚函数
18 // 父类中默认的实现是忽略右键菜单事件
19 // 重写之后，处理右键菜单事件
20 void MainWindow::contextMenuEvent(QContextMenuEvent* event)
21 {
22     mExitMenu->exec(QCursor::pos());
23
24     // 调用accept 表示，这个事件我已经处理，不需要向上传递了
25     event->accept();
26 }

```

此时，在窗口上右键，就可以弹出右键菜单，并且点击其中的退出菜单项可以退出程序

7. 界面布局

上一节的窗口，还是空白的没有任何的控件，本节拖拽对应的控件，完成界面的布局，最终布局效果：

界面布局的过程比较繁琐，需要花很多的时间来调整，我就不带大家从零布局了，布局的源码会提供给大家。

但是把一些需要注意的点列出如下：

7.1 删除菜单栏、状态栏

本项目不需要菜单栏、状态栏，删除即可

7.2 布局结构

在 `centralWidget` 下再添加一个 `widget`，之后所有的控件都是放在这个 `widget` 窗口之下

对象	类
MainWindow	QMainWindow
centralwidget	QWidget
widget	QWidget
widget_10	QWidget
lblGanMao	QLabel
verticalSpacer	Spacer
> widget_2	QWidget
widget_11	QWidget
> widget_5	QWidget
> widget_6	QWidget
widget_7	QWidget
lblHighCurve	QLabel
lblLowCurve	QLabel
> widget_8	QWidget
> widget_9	QWidget
widget_3	QWidget
lblTypeIcon	QLabel
verticalLayout_2	QVBoxLayout
> horizontalLayout	QHBoxLayout
> horizontalLayout_2	QHBoxLayout
widget_4	QWidget
btnSearch	QPushButton
horizontalSpacer_3	Spacer
lblDate	QLabel
leCity	QLineEdit

7.3 设置窗口背景

设置 `widget` 样式表，如下：

```

1 QWidget#widget{
2     border-image: url(:/res/background.png);
3 }
4
5 QLabel {
6     font: 25 10pt "微软雅黑";
7     border-radius: 4px;
8     background-color: rgba(60, 60, 60, 100);
9     color: rgb(255, 255, 255);
10 }

```

这样：

- 整个窗口的背景就设置为资源文件中的 `background.png`
- 设置了 `widget` 窗口中的所有标签的字体、边框圆角、背景色、前景色
- 如果要单独设置某个标签的样式，直接将要设置的样式，设置到对应标签上即可

比如“空气质量”的“优”的标签的样式被单独设置为：`background-color: rgba(0, 255, 255, 100);`

7.4 栅格布局

这里用到了多个栅格布局：


- “今天”的湿度、PM2.5、风力风向、空气质量
- “六天”的天气类型中的图标和文本、星期和日期的文本

7.5 标签显示图片

有两种方式设置标签显示图片：

- 属性窗口
- 代码动态设置图片

属性窗口中，设置图片的方法如下：

▼ QLabel	
> text	
textFormat	Auto Text
pixmap	 Qing.png
scaledContents	<input type="checkbox"/>

当然，对于“天气类型”，比如：阴、晴、小雨、中雨等，需要在代码中，根据服务器返回的数据，来动态设置

7.6 弹簧

为了界面的对齐和美观，可以添加一些水平和垂直的弹簧

7.7 修改控件名称

为了便于在代码中引用对应的控件，修改各控件名字，见名知义

8. 窗口随鼠标移动

由于窗口无边框，无法通过标题栏来移动窗口。但是可以重写父类的鼠标事件，来实现窗口随鼠标移动

首先，在 `mainwindow.h` 声明两个鼠标事件

```
1 class MainWindow : public QMainWindow {
2
3 protected:
4     void mousePressEvent(QMouseEvent* event);
5     void mouseMoveEvent(QMouseEvent* event);
6
7 private:
8     QPoint    mOffset;        // 窗口移动时，鼠标与窗口左上角的偏移
9 };
```

然后，在 `mainwindow.cpp` 中，实现如下：

```
1 void MainWindow::mousePressEvent(QMouseEvent* event)
2 {
3     qDebug() << "窗口左上角: " << this->pos() << ", 鼠标坐标点: " << event-
4     >globalPos();
5     mOffset = event->globalPos() - this->pos();
6 }
7 void MainWindow::mouseMoveEvent(QMouseEvent* event)
8 {
```

```
9     this->move(event->globalPos() - mOffset);
10 }
```

注意：

`move()` 函数移动窗口时，其实是设置窗口左上角的位置，因此上面定义了一个全局变量 `mOffset`，来记录鼠标按下的位置

9. 请求天气数据

上面已经讲解了 Qt 中实现 `HTTP` 请求的类，这里就开始真正使用这些类，来请求网络的数据。

9.1 添加网络模块

如果要使用 `HTTP` 获取数据，需要在 `.pro` 文件中，添加网络模块：

```
1 QT += core gui network
```

9.2 声明网络对象

在 `mainwindow.h` 的头文件中，声明用于 `HTTP` 通信的 `QNetworkAccessManager` 指针对象

```
1 #include <QNetworkAccessManager>
2 #include <QNetworkReply>
3
4 class MainWindow : public QMainWindow {
5 private:
6     QNetworkAccessManager* mNetAccessManager;
7 };
```

9.3 声明槽函数

在 `mainwindow.h` 的头文件中，声明 `onReplied()` 槽函数，用于处理服务器返回的数据

```
1 class MainWindow : public QMainWindow {
```

```
2 private slots:
3     void onReplied(QNetworkReply* reply);
4 };
```

9.4 关联信号槽

将 `QNetworkAccessManager` 的 `finished()` 信号，与自定义的 `onReplied()` 槽函数进行关联

```
1 MainWindow::MainWindow(QWidget* parent) : QWidget(parent), ui(new
  Ui::MainWindow)
2 {
3     // 关联信号槽
4     mNetAccessManager = new QNetworkAccessManager(this);
5     connect(mNetAccessManager, &QNetworkAccessManager::finished, this,
  &MainWindow::onReplied);
6 }
```

9.5 发送 http 请求

首先，在 `mainwindow.h` 声明一个用于发送 `HTTP` 请求的成员函数：

```
1 class MainWindow : public QMainWindow {
2
3 protected:
4     void getWeatherInfo(QString cityCode);
5 };
```

然后，调用 `getWeatherInfo()` 来发送 `HTTP` 请求

```
1 MainWindow::MainWindow(QWidget* parent) : QWidget(parent), ui(new
  Ui::MainWindow)
2 {
3     // 直接在构造中，去请求天气数据
4     getWeatherInfo("101010100"); // 101010100 是北京的城市编码
5 }
```

```

6
7 void MainWindow::getWeatherInfo(QString cityCode)
8 {
9     QUrl url("http://t.weather.itboy.net/api/weather/city/" + cityCode);
10    mNetAccessManager->get(QNetworkRequest(url));
11 }

```

`getWeatherInfo()` 接收一个城市编码的参数

北京的城市编码是 `101010100`，暂且记住，下节课会讲解如何通过城市名获取城市编码

9.6 接收服务端数据

当 `HTTP` 请求完毕，服务器返回数据时，`mNetAccessManager` 就会发射一个 `finished()` 信号，进而调用 `onReplied()` 槽函数

```

1 void MainWindow::onReplied(QNetworkReply* reply)
2 {
3     // 响应的状态码为200，表示请求成功
4     int status_code = reply-
>attribute(QNetworkRequest::HttpStatusCodeAttribute).toInt();
5
6     qDebug() << "operation:" << reply->operation();           // 请求方式
7     qDebug() << "status code:" << status_code;                // 状态码
8     qDebug() << "url:" << reply->url();                          // url
9     qDebug() << "raw header:" << reply->rawHeaderList();        // header
10
11     // 如果指定的城市编码不存在，就会报错：
12     // "Error transferring
http://t.weather.itboy.net/api/weather/city/0000000000
13     // - server replied: Not Found"
14     if ( reply->error() != QNetworkReply::NoError || status_code != 200 ) {
15         qDebug("%s(%d) error: %s", __FUNCTION__, __LINE__, reply-
>errorString().toLatin1().data());
16         QMessageBox::warning(this, "天气", "请求数据失败！", QMessageBox::Ok);
17     } else {
18         //获取响应信息
19         QByteArray byteArray = reply->readAll();
20         qDebug() << "read all:" << byteArray.data();
21         //         parseJson(byteArray);
22     }
23
24     reply->deleteLater();
25 }

```

服务器返回的所有数据，都封装在 `QNetworkReply` 中，包括响应头、状态码、响应体等。

服务器返回的天气的数据格式为 `JSON` 格式

将接收到的 `JSON` 数据拷贝到在线解析的工具中，就可以很清晰地看到返回的数据的组织结构

这里仅仅是讲接收到的数据打印了出来，下节课开始再解析 `JSON`，然后显示到控件上

10. 解析天气数据

获取到服务端的数据之后，就可以使用 Qt 提供的 `JSON` 相关的类，来将其中的一个个字段解析出来，并展示到界面上了

10.1 定义两个类

由于界面上主要显示的是“今天”的天气，以及“最近六天”的天气，因此我们新建一个 `weatherdata.h`，并定义两个类：

- **Today**

用于显示今天的所有天气参数，也就是屏幕左侧的数据

- **Day**

用于显示六天的天气参数，也就是屏幕右侧的数据

这样，可以方便地将解析出的数据保存到类的成员变量

```
1 #ifndef WEATHERDATA_H
2 #define WEATHERDATA_H
3
4 #include <QString>
5
6 class Today
7 {
8 public:
9     Today()
10     {
11         date = "2022-10-20";
12         city = "广州";
13
14         ganmao = "感冒指数";
15
16         wendu = 0;
```

```
17         shidu = "0%";
18         pm25 = 0;
19         quality = "无数据";
20
21         type = "多云";
22
23         fl = "2级";
24         fx = "南风";
25
26         high = 30;
27         low = 18;
28     }
29
30     QString date;
31     QString city;
32
33     QString ganmao;
34
35     int wendu;
36     QString shidu;
37     int pm25;
38     QString quality;
39
40     QString type;
41
42     QString fx;
43     QString fl;
44
45     int high;
46     int low;
47 };
48
49 class Day
50 {
51 public:
52     Day()
53     {
54         date = "2022-10-20";
55         week = "周五";
56
57         type = "多云";
58
59         high = 0;
60         low = 0;
61
62         fx = "南风";
63         fl = "2级";
```



```

64
65     aqi = 0;
66 }
67
68 QString date;
69 QString week;
70
71 QString type;
72
73 int high;
74 int low;
75
76 QString fx;
77 QString fl;
78
79 int aqi;
80 };
81
82 #endif // WEATHERDATA_H

```

然后，在 `mainwindow.h` 中定义两个成员变量 `mToday` 和 `mDay[6]`，并声明成员函数 `parseJson()` 来解析 `JSON` 数据

```

1 class MainWindow : public QMainWindow {
2
3 protected:
4     void parseJson(QByteArray& byteArray);
5
6 private:
7     Today mToday;
8     Day mDay[6];
9 };

```

10.2 解析数据

在 `parseJson()` 中解析数据，解析出的数据就保存到两个成员变量 `mToday` 和 `mDay[6]` 中：

```

1 void MainWindow::parseJson(QByteArray& byteArray)
2 {
3     QJsonParseError err;

```

```

4     QJsonDocument doc = QJsonDocument::fromJson(byteArray, &err);
5     if ( err.error != QJsonParseError::NoError ) {
6         qDebug("%s(%d): %s", __FUNCTION__, __LINE__,
err.errorString().toLatin1().data());
7         return;
8     }
9
10    QJsonObject rootObj = doc.object();
11    qDebug() << rootObj.value("message").toString();
12    QString message = rootObj.value("message").toString();
13    if ( !message.contains("success") ) {
14        QMessageBox::warning(this, "天气", "请求数据失败!", QMessageBox::Ok);
15        return;
16    }
17
18    // 1. 解析日期和城市
19    mToday.date = rootObj.value("date").toString();
20    mToday.city =
rootObj.value("cityInfo").toObject().value("city").toString();
21
22    QJsonObject objData = rootObj.value("data").toObject();
23
24    // 2. 解析 yesterday
25    QJsonObject objYestody = objData.value("yesterday").toObject();
26    mDay[0].week = objYestody.value("week").toString();
27    mDay[0].date = objYestody.value("ymd").toString();
28
29    mDay[0].type = objYestody.value("type").toString();
30
31    QString s;
32    s = objYestody.value("high").toString().split(" ").at(1);
33    s = s.left(s.length() - 1);
34    mDay[0].high = s.toInt();
35
36    s = objYestody.value("low").toString().split(" ").at(1);
37    s = s.left(s.length() - 1);
38    mDay[0].low = s.toInt();
39
40    mDay[0].fx = objYestody.value("fx").toString();
41    mDay[0].fl = objYestody.value("fl").toString();
42
43    mDay[0].aqi = objYestody.value("aqi").toDouble();
44
45    // 3. 解析 forecast 中5天的数据
46    QJsonArray forecastArr = objData.value("forecast").toArray();
47    for ( int i = 0; i < 5; i++ ) {
48        QJsonObject objForecast = forecastArr[i].toObject();

```

```

49     mDay[i + 1].week = objForecast.value("week").toString();
50     mDay[i + 1].date = objForecast.value("ymd").toString();
51
52     mDay[i + 1].type = objForecast.value("type").toString();
53
54     QString s;
55     s = objForecast.value("high").toString().split(" ").at(1);
56     s = s.left(s.length() - 1);
57     mDay[i + 1].high = s.toInt();
58
59     s = objForecast.value("low").toString().split(" ").at(1);
60     s = s.left(s.length() - 1);
61     mDay[i + 1].low = s.toInt();
62
63     mDay[i + 1].fx = objForecast.value("fx").toString();
64     mDay[i + 1].fl = objForecast.value("fl").toString();
65
66     mDay[i + 1].aqi = objForecast.value("aqi").toDouble();
67 }
68
69 // 4. 解析今天的数据
70 mToday.ganmao = objData.value("ganmao").toString();
71
72 mToday.wendu = objData.value("wendu").toString().toInt();
73 mToday.shidu = objData.value("shidu").toString();
74 mToday.pm25 = objData.value("pm25").toDouble();
75 mToday.quality = objData.value("quality").toString();
76
77 // 5. forecast中的第一个数组，也是今天的数据
78 mToday.type = mDay[1].type;
79
80 mToday.fx = mDay[1].fx;
81 mToday.fl = mDay[1].fl;
82
83 mToday.high = mDay[1].high;
84 mToday.low = mDay[1].low;
85
86 // 6. 更新 UI
87 //     updateUI();
88 }

```

11. 更新 UI 界面

上一节将解析出的天气数据放到了 `mToday` 和 `mDay[6]` 中，这样就可以很方便地将数据展示到界面上。

11.1 创建控件数组

更新6天的数据时，为了方便使用循环来更新控件的显示，将同一组控件放到一个数组中。

首先，在 `mainwindow.h` 中，添加几个列表，如下：

```
1 class MainWindow : public QMainWindow
2 {
3 private:
4     // 星期和日期
5     QList<QLabel*> mWeekList;
6     QList<QLabel*> mDateList;
7
8     // 天气和天气图标
9     QList<QLabel*> mTypeList;
10    QList<QLabel*> mTypeIconList;
11
12    // 天气指数
13    QList<QLabel*> mAqiList;
14
15    // 风向和风力
16    QList<QLabel*> mFxList;
17    QList<QLabel*> mFlList;
18 };
```

然后，在 `mainwindow.cpp` 的构造中，初始化列表，如下：

```
1 MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new
  Ui::MainWindow)
2 {
3     // UI初始化
4     // 将控件放到数组里面，方便使用循环进行处理
5     // 星期和日期
6     mWeekList << ui->lblWeek0 << ui->lblWeek1 << ui->lblWeek2 << ui->lblWeek3
  << ui->lblWeek4 << ui->lblWeek5;
7     mDateList << ui->lblDate0 << ui->lblDate1 << ui->lblDate2 << ui->lblDate3
  << ui->lblDate4 << ui->lblDate5;
8
9     // 天气和天气图标
```

```

10     mTypeList << ui->lblType0 << ui->lblType1 << ui->lblType2 << ui->lblType3
    << ui->lblType4 << ui->lblType5;
11     mTypeIconList << ui->lblTypeIcon0 << ui->lblTypeIcon1 << ui->lblTypeIcon2
    << ui->lblTypeIcon3 << ui->lblTypeIcon4 << ui->lblTypeIcon5;
12
13     // 天气指数
14     mAqiList << ui->lblQuality0 << ui->lblQuality1 << ui->lblQuality2 << ui-
    >lblQuality3 << ui->lblQuality4 << ui->lblQuality5;
15
16     // 风向和风力
17     mFxList << ui->lblFx0 << ui->lblFx1 << ui->lblFx2 << ui->lblFx3 << ui-
    >lblFx4 << ui->lblFx5;
18     mFlList << ui->lblFl0 << ui->lblFl1 << ui->lblFl2 << ui->lblFl3 << ui-
    >lblFl4 << ui->lblFl5;
19 }

```

11.2 更新

首先，在 `mainwindow.h` 中声明一个成员函数 `updateUI()` 用于将解析出的数据，更新到界面显示

```

1 class MainWindow : public QMainWindow {
2
3 protected:
4     void updateUI();
5 };

```

然后，在 `mainwindow.cpp` 中实现，如下：

```

1 void MainWindow::updateUI()
2 {
3     // 1. 更新日期和城市
4     ui->lblDate->setText(QDateTime::fromString(mToday.date,
    "yyyyMMdd").toString("yyyy/MM/dd") + " " + mDay[1].week);
5     ui->lblCity->setText(mToday.city);
6
7     // 2. 更新今天
8     ui->lblTypeIcon->setPixmap(mTypeMap[mToday.type]);
9     ui->lblTemp->setText(QString::number(mToday.wendu) + "°");
10    ui->lblType->setText(mToday.type);

```

```
11     ui->lblLowHigh->setText(QString::number(mToday.low) + "~" +
    QString::number(mToday.high) + "°C");
12
13     ui->lblGanMao->setText("感冒指数: " + mToday.ganmao);
14
15     ui->lblWindFx->setText(mToday.fx);
16     ui->lblWindFl->setText(mToday.fl);
17
18     ui->lblPM25->setText(QString::number(mToday.pm25));
19
20     ui->lblShiDu->setText(mToday.shidu);
21
22     ui->lblQuality->setText(mToday.quality);
23
24     // 3. 更新六天
25     for ( int i = 0; i < 6; i++ ) {
26         // 3.1 更新星期和日期
27         mWeekList[i]->setText("周" + mDay[i].week.right(1));
28
29         //设置 昨天/今天/明天 的星期显示 - 不显示星期几, 而是显示
30         //“昨天”、“今天”、“明天”
31         ui->lblWeek0->setText("昨天");
32         ui->lblWeek1->setText("今天");
33         ui->lblWeek2->setText("明天");
34
35         QStringList ymdList = mDay[i].date.split("-");
36         mDateList[i]->setText(ymdList[1] + "/" + ymdList[2]);
37
38         // 3.2 更新天气类型
39         mTypeIconList[i]->setPixmap(mTypeMap[mDay[i].type]);
40         mTypeList[i]->setText(mDay[i].type);
41
42         // 3.3 更新空气质量
43         if ( mDay[i].aqi >= 0 && mDay[i].aqi <= 50 ) {
44             mAqiList[i]->setText("优");
45             mAqiList[i]->setStyleSheet("background-color: rgb(121, 184, 0);");
46         } else if ( mDay[i].aqi > 50 && mDay[i].aqi <= 100 ) {
47             mAqiList[i]->setText("良");
48             mAqiList[i]->setStyleSheet("background-color: rgb(255, 187, 23);");
49         } else if ( mDay[i].aqi > 100 && mDay[i].aqi <= 150 ) {
50             mAqiList[i]->setText("轻度");
51             mAqiList[i]->setStyleSheet("background-color: rgb(255, 87, 97);");
52         } else if ( mDay[i].aqi > 150 && mDay[i].aqi <= 200 ) {
53             mAqiList[i]->setText("中度");
54             mAqiList[i]->setStyleSheet("background-color: rgb(235, 17, 27);");
55         } else if ( mDay[i].aqi > 200 && mDay[i].aqi <= 300 ) {
56             mAqiList[i]->setText("重度");
```

```

57         mAqiList[i]->setStyleSheet("background-color: rgb(170, 0, 0);");
58     } else {
59         mAqiList[i]->setText("严重");
60         mAqiList[i]->setStyleSheet("background-color: rgb(110, 0, 0);");
61     }
62
63     // 3.4 更新风力/风向
64     mFxList[i]->setText(mDay[i].fx);
65     mFlList[i]->setText(mDay[i].fl);
66 }
67 }

```

11.3 天气图标的处理

由于在资源文件中，添加中文名称的图标，会报错，而请求回来的天气类型又都是中文，比如“晴”、“小雨”等

因此，在 `mainwindow.h` 中定义了一个 `MAP`，用于英文到中文的转换：

```

1 class MainWindow : public QMainWindow
2 {
3 private:
4     QMap<QString, QString> mTypeMap;
5 };

```

并且在构造中，以天气类型为 `key`，以图标资源的路径为 `value`，初始化 `MAP`，如下：

```

1 MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new
  Ui::MainWindow)
2 {
3     ui->setupUi(this);
4
5     mTypeMap.insert("暴雪", ":/res/type/BaoXue.png");
6     mTypeMap.insert("暴雨", ":/res/type/BaoYu.png");
7     mTypeMap.insert("暴雨到大暴雨", ":/res/type/BaoYuDaoDaBaoYu.png");
8     mTypeMap.insert("大暴雨", ":/res/type/DaBaoYu.png");
9     mTypeMap.insert("大暴雨到特大暴雨", ":/res/type/DaBaoYuDaoTeDaBaoYu.png");
10    mTypeMap.insert("大到暴雪", ":/res/type/DaDaoBaoXue.png");
11    mTypeMap.insert("大雪", ":/res/type/DaXue.png");
12    mTypeMap.insert("大雨", ":/res/type/DaYu.png");
13    mTypeMap.insert("冻雨", ":/res/type/DongYu.png");
14    mTypeMap.insert("多云", ":/res/type/DuoYun.png");

```

```

15     mTypeMap.insert("浮沉", ":/res/type/FuChen.png");
16     mTypeMap.insert("雷阵雨", ":/res/type/LeiZhenYu.png");
17     mTypeMap.insert("雷阵雨伴有冰雹", ":/res/type/LeiZhenYuBanYouBingBao.png");
18     mTypeMap.insert("霾", ":/res/type/Mai.png");
19     mTypeMap.insert("强沙尘暴", ":/res/type/QiangShaChenBao.png");
20     mTypeMap.insert("晴", ":/res/type/Qing.png");
21     mTypeMap.insert("沙尘暴", ":/res/type/ShaChenBao.png");
22     mTypeMap.insert("特大暴雨", ":/res/type/TeDaBaoYu.png");
23     mTypeMap.insert("undefined", ":/res/type/undefined.png");
24     mTypeMap.insert("雾", ":/res/type/Wu.png");
25     mTypeMap.insert("小到中雪", ":/res/type/XiaoDaoZhongXue.png");
26     mTypeMap.insert("小到中雨", ":/res/type/XiaoDaoZhongYu.png");
27     mTypeMap.insert("小雪", ":/res/type/XiaoXue.png");
28     mTypeMap.insert("小雨", ":/res/type/XiaoYu.png");
29     mTypeMap.insert("雪", ":/res/type/Xue.png");
30     mTypeMap.insert("扬沙", ":/res/type/YangSha.png");
31     mTypeMap.insert("阴", ":/res/type/Yin.png");
32     mTypeMap.insert("雨", ":/res/type/Yu.png");
33     mTypeMap.insert("雨夹雪", ":/res/type/YuJiaXue.png");
34     mTypeMap.insert("阵雪", ":/res/type/ZhenXue.png");
35     mTypeMap.insert("阵雨", ":/res/type/ZhenYu.png");
36     mTypeMap.insert("中到大雪", ":/res/type/ZhongDaoDaXue.png");
37     mTypeMap.insert("中到大雨", ":/res/type/ZhongDaoDaYu.png");
38     mTypeMap.insert("中雪", ":/res/type/ZhongXue.png");
39     mTypeMap.insert("中雨", ":/res/type/ZhongYu.png");
40 }

```

此时，就可以将请求回来的实际数据，展示到界面上了。

请求不同的城市天气，可以看到界面的展示，是跟着变化的，如下可以请求北京和广州的天气：

```

1  MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new
    Ui::MainWindow)
2  {
3      ui->setupUi(this);
4
5      //     getWeatherInfo("101010100"); // 101010100 是北京的城市编码
6      getWeatherInfo("101280101"); // 101280101 是广州的城市编码
7  }

```

解析出数据之后，还会绘制温度曲线，这个在下节讲解。

12. 获取城市编号

上面已经看到了，获取天气数据的接口，需要传递一个 9 位的城市编码，而作为用户在界面上通常是输入城市名称或者县城名称

那么这就需要将城市名称转换为对应的城市编码，已经有别人已经做好了一个 JSON 文档：

✓ citycode-2019-08-23.json

里面就是记录了城市名称和编码的对应关系，全国很多的城市和县城，因此该文件有 2 万多行：

```
1  [  
2    {  
3      "id": 1,  
4      "pid": 0,  
5      "city_code": "101010100",  
6      "city_name": "北京",  
7      "post_code": "100000",  
8      "area_code": "010",  
9      "ctime": "2019-07-11 17:30:06"  
10   },  
11   {  
12     "id": 24,  
13     "pid": 0,  
14     "city_code": "101020100",  
15     "city_name": "上海",  
16     "post_code": "200000",  
17     "area_code": "021",  
18     "ctime": "2019-07-11 17:30:08"  
19   },  
20   {  
21     "id": 75,  
22     "pid": 5,  
23     "city_code": "101280101",  
24     "city_name": "广州",  
25     "post_code": "510000",  
26     "area_code": "020",  
27     "ctime": "2019-07-11 17:30:21"  
28   }  
29   ...  
30  ]
```

接下来我们就来实现一个工具类：WeatherTool.h，用来实现城市名称转城市编码

```
1 #ifndef WEATHERTOOL_H
2 #define WEATHERTOOL_H
3
4 #include <QCoreApplication>
5 #include <QFile>
6 #include <QJsonArray>
7 #include <QJsonDocument>
8 #include <QJsonObject>
9 #include <QJsonParseError>
10 #include <QJsonValue>
11 #include <QMap>
12
13 class WeatherTool
14 {
15 private:
16     static void initCityMap()
17     {
18         // 默认将城市列表放在了 E 盘根目录下
19         QString filePath = "E:/citycode-2019-08-23.json";
20
21         QFile file(filePath);
22         file.open(QIODevice::ReadOnly | QIODevice::Text);
23         QByteArray json = file.readAll();
24         file.close();
25
26         QJsonParseError err;
27         QJsonDocument doc = QJsonDocument::fromJson(json, &err);
28         if ( err.error != QJsonParseError::NoError ) {
29             qDebug("%s(%d) parse json failed: %s", __FUNCTION__, __LINE__,
30 err.errorString().toStdString().data());
31             return;
32         }
33
34         // 文件应该是一个 JSON 数组
35         if ( !doc.isArray() ) {
36             qDebug("%s(%d) parse json failed: not an array", __FUNCTION__,
37 __LINE__);
38             return;
39         }
40
41         QJsonArray citys = doc.array();
42         for ( int i = 0; i < citys.size(); i++ ) {
43             QString code = citys[i].toObject().value("city_code").toString();
44             QString city = citys[i].toObject().value("city_name").toString();
45             // 省份没有 city_code
46             if ( code.size() > 0 ) {
47                 mCityMap.insert(city, code);
48             }
49         }
50     }
51
52 public:
53     WeatherTool() {}
54     ~WeatherTool() {}
55
56     void initCityMap();
57     QMap<QString, QString> getCityMap();
58
59 private:
60     QMap<QString, QString> mCityMap;
61 };
62
63 #endif
```

```

46         }
47     }
48 }
49
50 public:
51     static QString getCityCode(QString cityName)
52     {
53         if ( mCityMap.isEmpty() ) {
54             initCityMap();
55         }
56
57         // 有的城市没有加 “市”，比如 “北京”
58         // 县，都是加了 “县”的
59         QMap<QString, QString>::const_iterator it = mCityMap.find(cityName);
60
61         if ( it == mCityMap.end() ) {
62             it = mCityMap.find(cityName + "市");
63         }
64
65         if ( it != mCityMap.end() ) {
66             return it.value();
67         }
68
69         return "";
70     }
71
72 private:
73     // 声明一个静态成员变量
74     static QMap<QString, QString> mCityMap;
75 };
76
77 // 初始化静态成员变量
78 QMap<QString, QString> WeatherTool::mCityMap = {};
79
80 #endif // WEATHERTOOL_H

```

此时，`getWeatherInfo()` 函数，就可以修改为如下：

```

1 void MainWindow::getWeatherInfo(QString cityName)
2 {
3     QString cityCode = WeatherTool::getCityCode(cityName);
4     if ( cityCode.isEmpty() ) {
5         QMessageBox::warning(this, "天气", "请检查输入是否正确!",
6                               QMessageBox::Ok);

```

```

6         return;
7     }
8
9     QString base = "http://t.weather.itboy.net/api/weather/city/";
10    QUrl url(base + cityCode);
11    mNetAccessManager->get(QNetworkRequest(url));
12 }

```

接下来实现，点击搜索按钮，来获取指定城市的天气数据

```

1 void MainWindow::on_btnSearch_clicked()
2 {
3     QString cityName = ui->leCity->text();
4     getWeatherInfo(cityName);
5 }

```

这样就实现了，通过输入城市名称，来获取天气数据！

13. 绘制温度曲线

将 6 天的数据，使用曲线连接起来，使得天气趋势一目了然

13.1 安装事件过滤器

为高低温的标签 `lblHighCurve` 和 `lblLowCurve` 安装事件过滤器

```

1 MainWindow::MainWindow(QWidget* parent) : QMainWindow(parent), ui(new
  Ui::MainWindow)
2 {
3     ui->setupUi(this);
4
5     // 安装事件过滤器
6     // 参数指定为 this，也就是当前窗口对象 MainWindow
7     ui->lblHighCurve->installEventFilter(this);
8     ui->lblLowCurve->installEventFilter(this);
9 }

```

13.2 重写 eventFilter() 函数

重写 `MainWindow` 的 `eventFilter()` 函数

这样就可以在 `eventFilter()` 函数中拦截发向 `lblHighCurve` 和 `lblLowCurve` 这两个控件的事件

具体实现如下：

首先，在 `mainwindow.h` 中声明 `eventFilter()`，并声明两个成员函数：

- `paintHighCurve()` - 绘制高温曲线
- `paintLowCurve()` - 绘制低温曲线

```
1 class MainWindow : public QMainWindow
2 {
3 public:
4     bool eventFilter(QObject* watched, QEvent* event);
5
6 protected:
7     // 绘制高低温曲线
8     void paintHighCurve();
9     void paintLowCurve();
10 }
```

然后，在 `mainwindow.cpp` 中实现：

```
1 bool MainWindow::eventFilter(QObject* watched, QEvent* event)
2 {
3     if ( watched == ui->lblHighCurve && event->type() == QEvent::Paint ) {
4         paintHighCurve();
5     }
6     if ( watched == ui->lblLowCurve && event->type() == QEvent::Paint ) {
7         paintLowCurve();
8     }
9     return QWidget::eventFilter(watched, event);
10 }
```

这样，当解析完数据，调用高低温标签的 `update()` 函数，就可以实现曲线的绘制：

```

1 void MainWindow::parseJson(QByteArray& byteArray)
2 {
3     ...
4
5     // 绘制温度曲线
6     ui->lblHighCurve->update();
7     ui->lblLowCurve->update();
8 }

```

具体流程：

- 调用标签的 `update()` 函数
- 框架发送 `QEvent::Paint` 事件 给标签
- 事件被 `MainWindow` 拦截，进而调用其 `eventFilter()` 函数
- 在 `eventFilter()` 中，调用 `paintHighCurve()` 和 `paintLowCurve()` 来真正绘制曲线

13.3 绘制曲线

以下是绘制高温曲线的代码：

```

1 void MainWindow::paintHighCurve()
2 {
3     QPainter painter(ui->lblHighCurve);
4
5     // 抗锯齿
6     painter.setRenderHint(QPainter::Antialiasing, true);
7
8     // 1. 获取 x 轴坐标
9     int pointX[6] = {0};
10    for ( int i = 0; i < 6; i++ ) {
11        pointX[i] = mWeekList[i]->pos().x() + mWeekList[i]->width() / 2;
12    }
13
14    // 2. 获取 y 轴坐标
15
16    // int temp[6] = {0};
17    int tempSum = 0;
18    int tempAverage = 0;
19
20    // 2.1 计算平均值
21    for ( int i = 0; i < 6; i++ ) {
22        tempSum += mDay[i].high;

```

```

23     }
24
25     tempAverage = tempSum / 6; // 最高温平均值
26
27     qDebug() << "paintHighCurve" << tempAverage;
28
29     // 2.2 计算 y 轴坐标
30     int pointY[6] = {0};
31     int yCenter = ui->lblHighCurve->height() / 2;
32     for ( int i = 0; i < 6; i++ ) {
33         pointY[i] = yCenter - ((mDay[i].high - tempAverage) * INCREMENT);
34     }
35
36     // 3. 开始绘制
37     // 3.1 初始化画笔
38     QPen pen = painter.pen();
39     pen.setWidth(1); //设置画笔宽度为1
40     pen.setColor(QColor(255, 170, 0)); //设置颜色
41     painter.save();
42
43     painter.setPen(pen);
44     painter.setBrush(QColor(255, 170, 0)); //设置画刷颜色
45
46     // 3.2 画点、写文本
47     for ( int i = 0; i < 6; i++ ) {
48         painter.drawEllipse(QPoint(pointX[i], pointY[i]), POINT_RADIUS,
POINT_RADIUS);
49         painter.drawText(QPoint(pointX[i] - TEXT_OFFSET_X, pointY[i] -
TEXT_OFFSET_Y), QString::number(mDay[i].high) + "°");
50     }
51
52     // 3.3 绘制曲线
53     for ( int i = 0; i < 5; i++ ) {
54         if ( i == 0 ) {
55             pen.setStyle(Qt::DotLine); //虚线
56             painter.setPen(pen);
57         } else {
58             pen.setStyle(Qt::SolidLine); // 实线
59             painter.setPen(pen);
60         }
61         painter.drawLine(pointX[i], pointY[i], pointX[i + 1], pointY[i + 1]);
62     }
63
64     painter.restore();
65 }

```

13.4 程序说明

(1) 宏定义

定义了以下几个宏，而不是在代码中直接写死：

```
1 // 温度曲线相关的宏
2 #define INCREMENT      3 // 温度每升高/降低1度，y轴坐标的增量
3 #define POINT_RADIUS   3 // 曲线描点的大小
4 #define TEXT_OFFSET_X 12 // 温度文本相对于点的偏移
5 #define TEXT_OFFSET_Y 10 // 温度文本相对于点的偏移
```

(2) 绘图相关

绘制曲线时，有很多的小细节，就不详细展开了，比如：

- 画笔、画刷的颜色
- 划线、画圆
- 实线、虚线

相信你学过 Qt 中的绘图一节的话，这是再简单不过了！