STM32 FOTA 例程之 cJSON 使用

前言

在 STM32 OTA 例程中，设备端（stm32F769 探索版）与云端交换数据使用的是 json 格式。在本篇文章中，将对 json 格式以及 Cjson 的使用及注意事项进行说明。

JSON 格式

JSON（JavaScript Object Notation）是一个轻量级的数据交换格式。既便于开发者读写，也便于机器分析和构建。它独立于开发语言，是一种文本格式，很适用描述在各个系统间交换的数据。

JSON 格式的数据看起来就像下面这个样子：

```
{
  “Room1”: {
      “LED”: “on”,
    “Temperature”: “36”
   }
}
```

这个 JSON 数据描述的是 Room1 的 LED 灯状态以及温度值。它由一组“名称（key）”以及对应的“ 值（value）”组成。“名称”和“ 值”之间由“ ：”分开。各组“名称：值”对之间由“，”符号进行分割。

cJSON 的使用

针对不同的开发语言，网上有很多 JSON 的实现, demo 里使用的是 Cjson，版本 1.6。它的源码可以在 https://github.com/DaveGamble/cJSON 上下载。

将 Cjson 添加到工程

Cjson 只有一个 C 文件 Cjson.C 和一个头文件 Cjson.h。所以只需要将这两个文件拷贝到工程文件夹中，并将 Cjson.C 添加到工程中就可以了。

数据结构

Cjson 中使用下面的数据结构来表示 JSON 数据。

```c
/* The cJSON structure: */
typedef struct cJSON
{
    struct cJSON *next;
    struct cJSON *prev;
    struct cJSON *child;
    int type;
    char *valuestring;
    /* writing to valueint is DEPRECATED, use cJSON_SetNumberValue instead */
    int valueint;
    double valuedouble;
    char *string;
} cJSON;
```

*next 和*prev 指针可以用来遍历“矩阵”或者“对象”类型的 JSON 数据链表；这两种类型的 JSON 数据还会有一个子数据指针*child

type : 表示该 json 数据的类型，比如数字，字符串、矩阵、对象等

*valuestring, valueint, valuedouble 和*string 指针分别指向该 json 数据类型具体的值，视其类型而定。

使用 cjson 生成 json 数据

下面我们看看如何使用 cjson 来生成下面的数据：

```
{
  “reported”: {
      “LED”: “on”,
    “status” : “normal”
   }
}
```

见下面的代码：

```c
cJSON * reported_obj;
cJSON * device_obj;

reported_obj = cJSON_CreateObject();

device_obj = cJSON_CreateObject();


//add LED status
if(flag_led)
  cJSON_AddStringToObject(reported_obj, "LED", "on");
else
  cJSON_AddStringToObject(reported_obj, "LED", "off");

//add device status
cJSON_AddStringToObject(reported_obj,"status", "normal");
```

创建空的对象：reproted_obj 和 device_obj

在 reproted_obj 中添加对象“LED”作为它的子集

在 reproted_obj 中添加对象“status”作为它的子集

```
cJSON_AddItemToObject(device_obj, "reported", reported_obj);
```

将 reproted_obj 添加到 device_obj，作为它的子集

现在就已经在 cjson 中，构件好了和前面的数据对应的数据结构。但现在这个数据结构还不能发送出去，需要调用 cJSON_Print 将其打印成串行的数据，存放在 buffer 中，以便后面进行发送。

```
cjson_print_buf = cJSON_Print(device_obj);
```

cJSON_Print 执行的时候会向系统申请一段内存来保存串行化了的数据，并返回其指针。这里一定要注意的是，cJSON_Print 中申请的内存，一定要记得释放（cjson 的代码中不会自动去做释放动作），否则就会出现内存泄漏。

通过 cJSON_CreateObject 创建的对象，也需要调用 cJSON_Delete 来进行删除并释放占用的内存。否则也会出现内存泄漏。见下面的代码：

```
if(cjson_print_buf!=NULL)
{
  snprintfreturn = snprintf(buf,bufsize,cjson_print_buf);
  cJSON_free(cjson_print_buf);
}
cJSON_Delete(device_obj);
```

仔细的同学可能会发现为什么调用了两次 cJSON_CreateObject，但只看到释放了其中的 device_obj。这里也是需要注意的一个地方，从前面的代码中，我们可以看到，reported_obj 最终是作为子对象添加到了 device_obj 中，所以在删除 device_obj 时，cJSON_Delete 会自动删除 device_obj 中所有的子对像，故而不需要再调用 cJSON_Delete 对 reported_obj 进行删除。

使用 cjson 解析 json 数据

可以通过 cJSON_Parse()函数来解析接收到的 json 数据，cJSON_Parse()函数会对数据进行解析，并申请一段内存保存解析后的 cjson 的数据结构，并返回指针。

通过 cJSON_GetObjectItem()函数可以获取解析后的 cjson 数据结构中的第一级子对象。

使用 cJSON_Parse()后，切记也一定要通过 cJSON_Delete 释放之前所申请的内存。

下面代码是对收到的 json 数据的解析过程。收到的数据内容为：

```
{
  “desired”: {
     “LED”: “on”,
    “SW_Version” : “02010100”
  }
}
```

解析收到的消息 message->payload

```
receive_obj = cJSON_Parse((char*)message->payload);
desired_obj = cJSON_GetObjectItem(receive_obj, "desired");
item_obj = desired_obj->child;
while(item_obj)
{
```

提取 desired 对象

依次解析 desired 对象下的子对象值

```
    string = item_obj->string;
    if(!strcmp(string, "LED")){
      if(!strcmp(item_obj->valuestring,"on"))
      {
        Led_On();
        flag_led = 1;
        msg_info("LED On!\n");
      }
      else
      {
        Led_Off();
        flag_led =0;
        msg_info("LED Off!\n");
      }
    }

  item_obj = item_obj->next;
  }

  cJSON_Delete(receive_obj);
```

释放 receieve_obj 及其子对象

总结

cJSON 代码量不大，用起来也方便。使用的时候一定要注意前面提到对使用完的内存空间进行释放。否则会造成内存泄漏。

通知 – 请仔细阅读

意法半导体公司及其子公司（“ST”）保留随时对ST 产品和/ 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于ST 产品的最新信息。ST 产品的销售依照订单确认时的相关ST 销售条款。

买方自行负责对ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的ST 产品如有不同于此处提供的信息的规定，将导致ST 针对该产品授予的任何保证失效。

ST 和ST 徽标是ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。