Protobuf编解码模块优化说明

# 代码自动生成pbstru简介

Protobuf是由Google主推的一种二进制接口规范，可快速的对结构化数据进行序列化和反序列化以便于在网络中传输。

为了使用者的方便，Google及第三方为Protobuf提供了多种语言的API，包括C++/Java/Python等。出于效率及编程习惯，CDB产品需要使用C语言实现相关的API。CDB项目曾经比较和使用了多个第三方的C语言实现方案，实际运行中发现其性能都无法达到要求，所以决定自研Protobuf的编解码功能。

自研Pbstru（Protobuf to Structure）工具可自动生成编解码模块的代码，实现了Protobuf码流和C语言数据结构间的相互转换。此工具的输入文件为proto文件（Protobuf标准接口定义文件，目前支持版本2），输出为编解码模块的整套代码，将自动生成的代码联编入目标系统，即可实现结构化数据的序列化和反序列化。

例如：

输入的proto定义如下：

message AsyncConfirmReq {

repeated uint32 slotid\_list = 1;

}

通过pbstru可自动生成对应的C数据结构如下，st\_AsyncConfirmReq结构和AsyncConfirmReq消息是对应关系

#define MAX\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ 144

typedef struct {

size\_t count;

DWORD item[MAX\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ]; /\* tag:1 type:uint32 \*/

} st\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ\_uint32\_list;

typedef struct {

st\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ\_uint32\_list var\_slotid\_list; /\* tag:1 \*/

} st\_AsyncConfirmReq;

同时生成下列编解码函数，分别实现了清除、编码和解码功能

void clear\_message\_AsyncConfirmReq(st\_AsyncConfirmReq \*msg);

size\_t encode\_message\_AsyncConfirmReq(const st\_AsyncConfirmReq\* const msg, BYTE\* const buf);

BOOL decode\_message\_AsyncConfirmReq(BYTE\* const buf, const size\_t buf\_len, st\_AsyncConfirmReq\* const msg);

**序列化时调用者需要做:**

1. 调用clear函数，将msg结构全部清空，而后对msg结构进行完整赋值；
2. 调用encode函数，且将buf赋值为NULL，返回序列化后的缓冲区长度（避免超出缓冲区空间）
3. 再次调用encode函数，将buf赋值为有效的缓冲区，序列化后的数据将填充缓冲区；

**反序列化时调用者需要做：**

1. 调用clear函数，将msg结构全部清空；
2. 调用decode函数，将buf缓冲区中的数据解码到msg结构

编解码内部细节暂略，有兴趣可以深入到编解码函数内部实现。

# 编解码功能与CDB的集成

Pbstru工具代码存在/cdbtools/pbstru目录下，可通过执行/cdbtools/pbstru/pbstru/build.sh进行编译，生成的可执行文件保存在/cdbtools/pbstru/pbstru/bin目录下，可执行文件需要提交到git。

CDB/Cache整系统编译时，pbstru工具会拷贝到build/pbstru目录，输入文件为cdb.proto及cdb\_ccc.proto，生成的源代码保存在build/pbstru/codec目录。

**以上需要注意的只有一点：如果修改了pbstru的源代码，注意手工编译，并将可执行程序也提交到git。**

# 编解码功能优化

## 支持动态数组

参考上面的例子，在下列结构中，如果数组的元素的个数由144增长，则此固定结构会使用大量空间，而实际上大多数情况下又不需要用那么多空间，从而造成内存空间的浪费。

#define MAX\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ 144

typedef struct {

size\_t count;

DWORD item[MAX\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ]; /\* tag:1 type:uint32 \*/

} st\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ\_uint32\_list;

对此问题，pbstru需要支持动态数组，以支持某些结构需要动态申请内存的需求。对以上例子，结构将优化为：

#define MAX\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ 0

// Dynamic array to store unlimited repeated field.

typedef struct {

size\_t count;

**size\_t max\_size; /\* max size of dynamic array \*/**

DWORD item[]; /\* tag:1 type:uint32 \*/

} st\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ\_uint32\_list;

数组结构st\_SLOTID\_LIST\_IN\_ASYNCCONFIRMREQ\_uint32\_list中item定义为指针，其具体的元素个数不确定。增加了max\_size，标识item中通过动态申请内存得到的空间所能够保存的元素个数。count标识item中目前有效的元素个数。

由于max\_size信息在结构初始化时其值是不确定的，并且在此结构在调用者重复使用过程中其中的max\_size信息又需要保持不变，所以考虑增加构造和析构函数：

/\* construct msg when first use, call clear\_message\_\*() internaly. \*/

void constru\_message\_AsyncConfirmReq(st\_AsyncConfirmReq\* msg);

/\* destruct msg \*/

void destru\_message\_AsyncConfirmReq(st\_AsyncConfirmReq\* msg);

在constru构造函数中实现以下逻辑：

1. 嵌套调用，将结构中的max\_size设置为0，item设置为NULL，标识所有动态数组未初始化
2. 嵌套调用clear函数，将结构中的count设置为0，标识无有效数据；

在destru构析函数中实现以上逻辑：

1. 嵌套调用，若结构中的item为非空指针，则释放内存

**相应的，序列化及反序列化时的改进措施：**

**序列化时调用者需要做:**

1. 调用结构的constru构造函数，出于效率考虑只在初始化时调用一次；
2. 调用clear函数，将msg结构全部清空；
3. 对msg结构进行完整赋值，**判断动态数据空间不够时，申请新的连续内存，将原数据拷贝到新申请的内存中，释放原内存，更新max\_size值；内存不足时编码失败；**
4. 调用encode函数，且将buf赋值为NULL，返回序列化后的缓冲区长度（避免超出缓冲区空间）
5. 再次调用encode函数，将buf赋值为有效的缓冲区，序列化后的数据将填充缓冲区；
6. 当结构不再使用时，调用结构的destru析构函数释放内存

**反序列化时调用者需要做：**

1. 调用结构的constru构造函数，出于效率考虑只在初始化时调用一次；
2. 调用clear函数，将msg结构全部清空；
3. 调用decode函数，将buf缓冲区中的数据解码到msg结构，**若动态数组空间不够时，自动申请新的连续内存，将原数据拷贝到新申请的内存中，释放原内存，更新max\_size值；内存不足时解码失败；**
4. 当结构不再使用时，调用结构的destru析构函数释放内存

## 线程安全

在使用pbstru生成的代码时，如果消息结构较为复杂，或者其中包含有较大的静态数组时，结构所占用的空间非常大，此时不建议使用临时变量保存较大的结构，否则会导致堆栈溢出错误。

单线程情况下，建议将数据结构保存在全局变量中，流程中对全局变量进行重复使用。多线程情况下，数据结构的空间可从线程相关的数据区中申请，同样流程中可对此数据区进行重复使用。另外动态数据所使用的内存也需要从线程相关的数据区中申请。

为了适配所有情况，pbstru中将不使用自身的内存申请释放函数，此部分功能将由调用者实现。

void \*pbstru\_malloc(size\_t size);

void pbstru\_free(void \*ptr);