**SafeJson手册**

-金其明

目录

[原因： 2](#_Toc375915910)

[实现方案选择： 2](#_Toc375915911)

[使用方法： 3](#_Toc375915912)

[特殊功能： 4](#_Toc375915913)

[string ⬄ number 4](#_Toc375915914)

[json重建 4](#_Toc375915915)

[快捷语句 5](#_Toc375915916)

[检查路径是否可达 5](#_Toc375915917)

[获取test失败信息 7](#_Toc375915918)

[内部实现机制 8](#_Toc375915919)

[最后 9](#_Toc375915920)

# 原因：

1. jsoncpp经常由于值类型不合预期，导致程序挂掉，其原因是jsoncpp内部进行了JSON\_ASSERT\_MESSAGE断言。

这种断言初衷是为了及时发现异常的字段，但是实际环境下，输入的内容无法预计，错误还是会经常发生。

1. 为了避免代码执行到断言处，我们必须进行一系列的isXX()判断。不仅如此，有时候判断了却依然判断不全。这一方面是由于判断全面太麻烦，另一方面是在正常输入下程序运行良好，不容易显现判断不全导致的异常。

SafeJson就是期望解决这些问题，让开发人员摆脱掉复杂的判断语句；同时又不用担心判断不够严谨，使可以安全地使用json；甚至可以提供一定的异常信息供调试用。

# 实现方案选择：

1. 继承Json::Value类，然后对子类进行安全检查和访问控制。

这种方案的初衷是为了能无缝替换Json::Value类的使用。如下面的接口，只要能让参数response实际传的是其子类，那么函数里面的代码就可以保持用法和结果不变。

bool CConvertEncode::GetConfig(Json::Value& response, int channel){…}

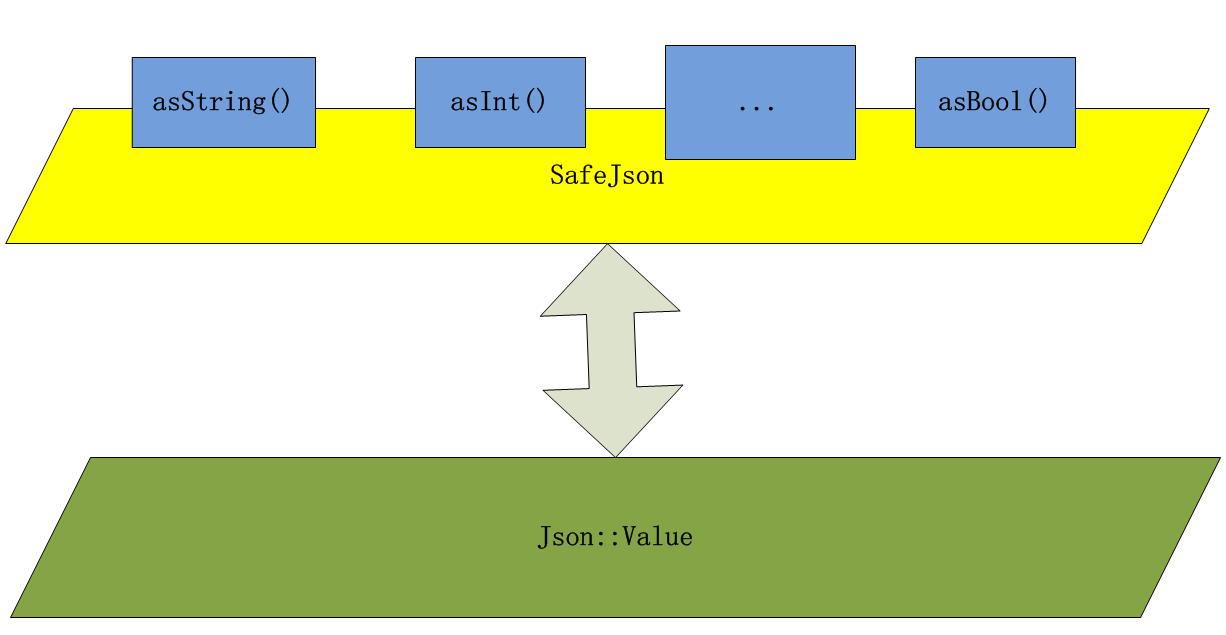
但是有两个问题：

1. 这类接口是RPC框架定好的，它只可能给你传Json::Value类型，而不会是他的子类。
2. Json::Value的接口都不是虚函数，因此无法通过基类的指针(或者引用)调用子类函数

因此继承的方法没有优势。

1. 使用一个工具类，这个工具类的接口与Json::Value基本一致。通过这个工具类能安全地获取目标对象的值，同时相似的接口又能保持使用习惯不变。

我这里使用的是这种方案。



# 使用方法：

假如request的json信息如下：

request:{

"params":{

"userName":"admin",

"password":"admin"

}

}

为了安全地取出userName和password字段需要进行如下判断：

if(request.isObject()

&& request.isMember("params")

&& request["params"].isObject()

&& (request["params"]["password"].isNull() || request["params"]["password"].isString())

&& (request["params"]["userName"].isNull() || request["params"]["userName"].isString()))

{

Std::string username = request["params"]["password"].asString() ;

Std::string password = request["params"]["password"].asString() ;

……

}

而使用safejson则不用如此,直接取值即可：

SafeJson **sjson**(request);//先定义再使用

Std:string Username = **sjson**["params"]["userName"].asString() ;

std:string Password = **sjson**["params"]["password"].asString() ;

除了定义变量外，没有增加额外的语句。同时用法完全没变。

注：sjson不能分步操作，如

sjson=sjson["params"];username=sjson["username"].asString().

这是为了使内部逻辑简单，我把operator=( const SafeJson& other );设为private了，因此分步写是编不过的（所以也不用特别去记这一点）。

而一步到位地取值通常也符合使用习惯。

# 特殊功能：

### string ⬄ number

字符串与数值的等价转换由参数FLAG\_STREQNUM控制。

打开该功能后可以达到字符串和数字自动转换的功能。即”123” ⬄ 123.如：

request：{

"a" : [ 1, 2, 3 ],

"b" : 123,

"c" : "123.0"

}

那么**sjson[** "**c**"**].asInt()** 结果不但不会报错而且能得到整数123.（如果不开这个功能将因为类型不对而得到默认值0）.

使用这个功能的方法如下：

SafeJson **sjson**(request, FLAG\_STREQNUM);// 在创建的时候包含**FLAG\_STREQNUM**标志。

**int I = sjson[** "**c**"**].asInt()**

**注：**字符和数值等价只是在asXX()时才起作用，在取子对象的操作中是不起作用的。

如request["123"]与request[123]是不同的

### json重建

json重建可以无条件的赋值，来覆盖掉原先的值。功能的场景如下：

request：{

"a" : [ 1, 2, 3 ],

"account" : 123

}

假如我需要给request的account赋值,期望的结果如下

request：{

"a" : [ 1, 2, 3 ],

"account" : {

"user":"admin",

"pass":"admin"

}

}

那么如果按以前的，request["account"]["user"] = "admin";将直接断言。

而采用SafeJson你可以有两种选择：

1. 不重建json，这时侯SafeJson发现sjson["account"]["user"] = "admin";企图将数值类型当object类型来用，是一种会引起断言的行为。它便自动忽略该操作。等效于什么也没做。这也是默认的策略。
2. 重建json，这时候，当SafeJson发现json["account"]["user"] = "admin";明显是把account当做一个object来用。于是它就自动重建account成为一个object，并完成赋值操作。后面如果执行request["account"][2] = "admin";则它又自动重建成Array对象。重建过程不会引起内存泄露。

打开json重建功能的方法如下：

SafeJson **sjson**(request, FLAG\_REWRITE);// 在创建的时候包含**FLAG\_REWRITE**标志。

sjson["account"]["user"] = "admin";

注：重建是在["key"]和[index]取子对象的时候起作用，对于json = "abc"赋值的时候json本来就自动会覆盖原来的值的。

### 快捷语句

前面我们通过定义SafeJson **sjson**(request)来建立对象。如果我连这一步都不想写能做到吗。

我们可以用匿名对象来实现一步操作的功能。

std::string user = (SafeJson(request))["account"]["user"].asString();

这里我定义了宏来代替匿名对象：

#define ONCEJSON(json) (SafeJson(json))

于是代码就变成std::string user=ONCEJSON(request)["account"]["user"].asString();

用宏的好处是可以顾名思议，同时可以把一些功能参数包含在里面。

比如#define AUTOONCE(json) (SafeJson(json,FLAG\_STREQNUM))

就能用AUTOONCE创建一个字符和数值等价的SafeJson匿名对象。

其他快捷操作宏可参见头文件。

这种用法比较适合在if语句中：

If(“admin” == ONCEJSON(request)["account"]["user"].asString()){…}

### 检查路径是否可达

前面都是直接获取数据的，假如我们在获取数据前，想要知道json的某个路径是否可达，某个值类型是否正确，那么要怎么做呢？

这里可以使用SafeJson的test()接口。

/\*

\* 是否能自然转成目标类型，

\* 返回false可能是某个路径中对某个非object取了["key"]

\* 或者对非array取了[5]，

\* 或者asXX()时jsontype无法转换

\* jsontype 为值的目标类型

\* 取Json::nullValue（默认值）时，表示只检查[xx]路径的可达性，并不检查转换成某种类型

\*/

bool test(Json::ValueType jsontype = Json::nullValue);

它的功能是检查路径及值的类型。这里有两个层次的需求：

1.能够执行asXX()而不被断言。

假如我们想要知道request[“account”][“user”].asString()能否正确的获取而不出错。

那么我们可以这么做：

SafeJson sjson(request);

If(sjson[“account”][“user”].test(Json::stringValue))

{

Std::string user = request[“account”][“user”].asString();

}

但是这其实是一种不严格的检查，假如我执行sjson[“notExist”][“user”].test(Json::stringValue)一样返回true。

这时因为request本身是一个object，而object在发现”key”不存在时，会创建一个空对象作为它的值，不会断言。之后对空对象的任何操作都是安全可达的。所以也会返回true。所以有了另一个层次的要求。

2.能够执行asXX()不断言，并且所有中间的key(对于object类型) 必须存在，index(对于array类型)都小于数组的size。

这是一种严格的检查，目的是为了能够检查出某个字段是否存在，某个下标是否超出数组的size.

用法如下：

SafeJson sjson(request, FLAG\_STRICTMEMBER); //增加**FLAG\_STRICTMEMBER**严格检查标志

If(sjson[“notExist”][“user”].test(Json::stringValue))

{

Std::string user = request[“notExist”][“user”].asString();

}

else

{

…

}

注：这种标志打开后，原本sjson[“notExist”][“user”]=”admin”;能够成功赋值，这时这种赋值会因为key不存在而无效，啥也不做。

这里可能会有个疑问**FLAG\_REWRITE**和**FLAG\_STRICTMEMBER**是否是两个互斥的功能。

一个是不管已有的内容，一律覆盖，一个是严格检查已有的内容，不存在就啥也不做。

其实不是的：

1. FLAG\_REWRITE是在当前类型无法转成object或者array时，才重建object或者array。

比如对于123 == request[“account”].asInt()的情况下我执行

SafeJson **sjson**(request, FLAG\_REWRITE);sjson[“account”][“user”] = “admin”;

这时原来的int类型被重建成object类型。

1. FLAG\_STRICTMEMBER则是在当前类型是object或者array,即类型没有错的情况下，发现字段或者下标不存在，而做出的忽略行为。

比如request={ "a" : [ 1, 2, 3 ]}的情况下，我执行

SafeJson sjson(request, FLAG\_STRICTMEMBER);sjson[“a”][3]=111;

这时严格检查数组的下标，将导致赋值操作置为无效。

1. 由于两个标志并不互斥，甚至相互独立，这两个功能（其实是任何一个其它功能）是可以同时打开的。

SafeJson **sjson**(request, FLAG\_REWRITE | FLAG\_STRICTMEMBER);

### 获取test失败信息

上面我们依靠test来判断路径是否可达，类型是否正确。

但是如果test返回false，我们能不能知道是具体哪个路径导致的呢，这将有助于我们的排错。

我们可以用getException().

/\*

\* 查询当前的状态，正确或者异常

\* 返回值中

\* code:取自JSON\_STATE

\* info:是错误信息，格式为"[原内容]->[引起错误的操作]"

\* 原内容为:1.[Object] 2.[Array] 3.[String] 4.[Boolean:true/false] 5.[Number:123/1.23] 6.[Null]

\* 引起错误的操作：1.["key"] 2.[12] 3.[asString] 4.[asInt] ...(其余见getasXXTag())

\*/

const JsonException& getException();

接口返回的是一个结构体：

typedef struct JSONEXCEPTION

{

int code;

std::string info;

}JsonException,\*pJsonException;

其中code代表错误号，取自下面的枚举值：

enum JSON\_STATE

{

JSON\_RIGHT,

JSON\_WRONG\_GETOBJECT, //cannot ["key"]

JSON\_WRONG\_GETARRAY, //cannot [3]

JSON\_WRONG\_ASSOMETYPE, //cannot asXX()

JSON\_WRONG\_NOMEMBER, //key不存在或者下标超出范围，打开严格检查时才会有

};

Info则是对应的字符串信息。

举个例子：

request：{

"a" : [ 1, 2, 3 ],

"account" : 123

}

这时我们执行：

SafeJson sjson(request);

If(sjson[“a”][“b”][“c”].test(Json::stringValue))

{

}

else

{

std::cout << sjson.getException().code << std::endl;

std::cout << sjson.getException().info << std::endl;

}

我们可以发现sjson[“a”][“b”][“c”].test(Json::stringValue)中最先出错的是[“b”].这一步试图把一个array当做object来用，这是一种会引起断言的操作。所以SafeJson就记录下了这第一个错误。

这里code是JSON\_WRONG\_GETOBJECT，代表是在试图获取子对象时出错。

Info内容是[Array]->[“b”]。

同样如果我执行sjson[“account”].test(Json::stringValue)那么code就是JSON\_WRONG\_ASSOMETYPE，

Info就是[Number:123]->[asString].

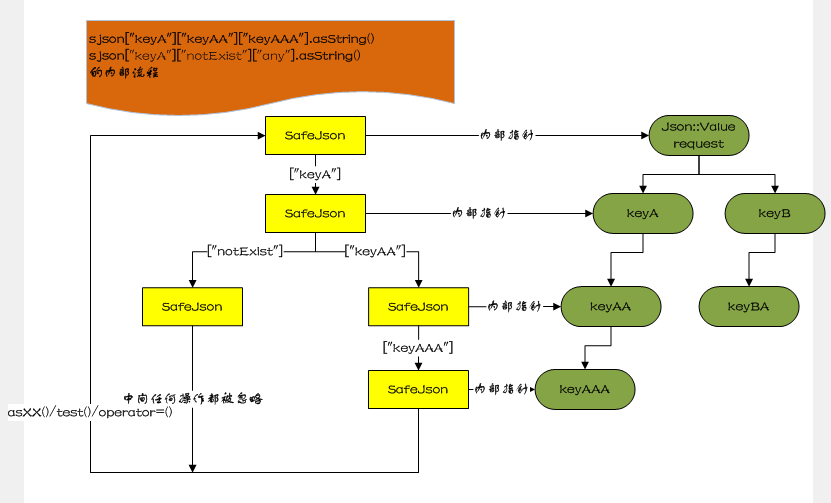
# 内部实现机制

内部有一个指向Json::Value的指针，每次[“key”]/[4]都会将指针引向下一级json对象。

当[“key”]/[4]无法获取，就设置标记和错误信息。并且后续的[“key”]/[4]都会被忽略掉。

最后执行asXX()/test()/operator=()即取值或检查或赋值是这些序列的最后一步。对于之前有错误标记的。这里就是直接返回一个默认值。

执行完这最后一步，内部状态会自动回到开始的时候，重新等待你[“key”]/[4]的操作。



# 最后

总结一下：

1.对于常规用法，我们是先定义，在使用，使用的接口与Json::Value保持一致。

SafeJson **sjson**(request);//先定义再使用

Std:string Username = **sjson**["params"]["userName"].asString() ;

2.对于特殊功能，我们可以打开相应的功能开关，这几个开关为

FLAG\_REWRITE

FLAG\_STREQNUM

FLAG\_STRICTMEMBER 。

可以在创建时传给构造函数，这几个功能是互相独立的，可以按需打开，不会互相影响。

SafeJson **sjson**(request, FLAG\_REWRITE | FLAG\_STREQNUM );

3.对于检查路径是否可达，类型是否正确，key是否存在，可以使用test接口。

并通过getException()获取test失败的原因。