目录

API 参考文件

前言

所有声明都在 jansson.h 中，
include < jansson.h >
在每个源文件中。所有常量都以 JSON_作为前缀(除了那些描述库版本的常量，以 JANSSON_作为前缀)。其他标识符以 json_作为前缀。类型名称的后缀是_t 和 typedef 'd，因此不需要使用 struct 关键字。

库版本

Jansson 版本的形式是 A.B.C，其中 A 是主版本，B 是次版本，C 是微版本。如果微版本为 0，则从版本字符串中省略，即版本字符串只是 A.B.

当一个新版本只修复 bug 而不添加新特性或功能时，微版本就会增加。当以向后兼容的方式添加新特性时，次要版本将增加，而微版本将设置为 0。当存在向后不兼容的更改时，主要版本将递增，其他版本将设置为零。

以下预处理器常量指定库的当前版本:
JANSSON_MAJOR_VERSION、JANSSON_MINOR_VERSION 、JANSSON_MICRO_VERSION 分别指定主版本、次版本和微版本的整数。
JANSSON_VERSION 当前版本的字符串表示。“1.2.1”或“1.3”。

JANSSON_VERSION_HEX 版本的 3 字节十六进制表示，例如，版本 1.2.1 是 0x010201，版本 1.3 是 0x010300。这在数值比较中很有用，例如:

```
#if    JANSSON_VERSION_HEX    >=    0x010300
/*特定于 1.3 版及以上版本的代码*/
# endif
```

此外，还有一些函数可以在运行时确定 Jansson 的版本:

const char * jansson_version_str ()

以与 JANSSON_VERSION 预处理器常量相同的格式返回 Jansson 库的版本。

新版本 2.13。

int jansson_version_cmp(int major, int minor, int micro)

如果发现 Jansson 的运行时版本分别小于、匹配或大于提供的主版本、次版本和微版本，则返回一个小于、等于或大于零的整数。

新版本 2.13。

JANSSON_THREAD_SAFE_REFCOUNT

如果定义了这个值，那么 Jansson 中的所有只读操作和引用计数都是线程安全的。此值不适用于 2.11 以上版本或编译器不提供内置原子函数时定义。

值表示形式

JSON 规范(RFC 4627)定义了以下数据类型:对象、数组、字符串、数字、布尔值和 null。JSON 类型是动态使用的;数组和对象可以容纳任何其他数据类型，包括它们自己。因此，Jansson 的类型系统在本质上也是动态的。有一个 C 类型表示所有 JSON 值，这个结构知道它所持有的 JSON 值的类型。

json_t

这个数据结构在整个库中用于表示所有 JSON 值。它总是包含它持有的 JSON 值的类型和值的引用计数。其余取决于值的类型。

json_t 的对象总是通过指针使用。有 api 用于查询类型、操作引用计数以及构造和操作不同类型的值。

除非另有说明，否则如果发生错误时，所有 API 函数都会返回一个错误值。根据函数的签名，错误值可以是 NULL，也可以是-1。无效的参数或无效的输入是错误的明显来源。内存分配和 I/O 操作也可能导致错误。

类型

enum json_type

JSON 值的类型。定义了以下成员:

JSON_OBJECT

JSON_ARRAY

JSON_STRING

JSON_INTEGER

JSON_REAL

JSON_TRUE

JSON_FALSE

JSON_NULL
它们对应于 JSON 对象、数组、字符串、数字、布尔值和 null。数字由类型 JSON_INTEGER 或类型 JSON_REAL 的值表示。true boolean 值由 JSON_TRUE 类型的值表示，false 由 JSON_FALSE 类型的值表示。
int json_typeof(const json_t *json)
返回 JSON 值的类型(a json_type cast to int)。json 不能为空。这个函数实际上是作为一个宏来实现的。
json_is_object (const json_t * json)
json_is_array (const json_t * json)
json_is_string (const json_t * json)
json_is_integer (const json_t * json)
json_is_real (const json_t * json)
json_is_true (const json_t * json)
json_is_false (const json_t * json)
json_is_null (const json_t * json)
这些函数(实际上是宏)对给定类型的值返回 true(非零)，对其他类型的值和 NULL 返回 false(零)。

json_is_number (const json_t * json)
对于类型 JSON_INTEGER 和 JSON_REAL 返回 true，对于其他类型和 NULL 返回 false。
json_is_boolean (const json_t * json)
对于类型 JSON_TRUE 和 JSON_FALSE 返回 true，对于其他类型的值和 NULL 返回 false。
json_boolean_value (const json_t * json)
json_is_true()的别名，即 JSON_TRUE 返回 1，否则返回 0。
新版本 2.7。

引用计数

引用计数用于跟踪值是否仍在使用中。当创建一个值时，它的引用计数被设置为 1。如果保存了对某个值的引用(例如，某个值被存储在某个地方以供以后使用)，则其引用计数将递增，当不再需要该值时，引用计数将递减。当引用计数降为 0 时，就没有任何引用了，并且可以销毁该值。
json_t * json_incref (json_t * json)
如果 json 不为空，则增加其引用计数。返回 json。
void json_decref (json_t * json)
减少 json 的引用计数。一旦调用 json_decref()将引用计数降为零，该值就会被销毁，不能再使用。
创建新 JSON 值的函数将引用计数设置为 1。这些函数被称为返回一个新的引用。其他返回(现有)JSON 值的函数通常不会增加引用计数。这些函数被称为返回借来的引用。因此，如果用户将持有作为借来的引用返回的值的引用，那么他必须调用 json_incref()。一旦不再需要这个值，就应该调用 json_decref()来释放引用。通常，所有接受 JSON 值作为参数的函数都会管理引用，即根据需要增加或减少引用计数。但是，有些函数会窃取引用，也就是说，它们的结果与用户在调用函数后立即对参数调用 json_decref()的结果相同。这些函数的后缀是 _new 或在其名称的某个地方有_new_。

例如，下面的代码创建了一个新的 JSON 数组并向其追加一个整数:

```
json_t *array,*integer;
array= json_array ();
integer= json_integer (42);
json_array_append(array,  integer);
json_decref(integer);
```

注意，调用者必须通过调用 json_decref()来释放对整数值的引用。通过使用一个引用窃取函数 json_array_append_new()代替 json_array_append()，代码变得简单多了:

```
json_t *array = json_array();
json_array_append_new(array,json_integer (42));
```

在这种情况下，用户不必显式地释放对整数值的引用，因为 json_array_append_new()在将该值附加到数组时，会窃取该引用。在下面的部分中，一个函数是否会返回一个新的或借来的引用，或者是否会偷取对其参数的引用，都将被清楚地记录下来。

循环引用

当直接或间接地将对象或数组插入自身时，将创建循环引用。直接的例子很简单:

```
json_t *obj = json_object();
json_object_set (obj,“foo”obj);
```

Jansson 将拒绝这样做，并且 json_object_set()(以及用于对象和数组的所有其他此类函数)将返回一个错误状态。间接的情况是危险的:

```
json_t *arr1 = json_array()，  *arr2 = json_array();
json_array_append (arr1 arr2);
json_array_append (arr2 arr1);
```

在本例中，数组 arr2 包含在数组 arr1 中，反之亦然。Jansson 无法在不影响性能的情况下检查此类间接循环引用，所以要由用户来避免它们。如果创建了循环引用，则 json_decref()不能释放值所消耗的内存。引用计数从不降为零，因为这些值保持了引用之间的相互引用。此外，尝试用任何编码函数对值进行编码都会失败。编码器检测循环引用并返回错误状态。

非关联化范围

新版本 2.9。

可以使用 json_auto_t 类型自动取消对范围末尾值的引用。例如:

```
void function(void)
{
json_auto_t *value = NULL;
value = json_string (" foo ");
/* json_decref(value)被自动调用。*/
}
```

此功能仅在 GCC 和 Clang 上可用。因此，如果您的项目对其他编译器有可移植性要求，您应该避免该特性。此外，在将值传递给偷取引用的函数时，应该一如既往地小心。

True,False and Null

这三个值是作为单例实现的，因此返回的指针在调用这些函数时不会改变。

json_t * json_true(void)

返回值:新引用。

返回 JSON 真值。

json_t * json_false(void)

返回值:新引用。

返回 JSON 假值。

json_t * json_boolean (val)

返回值:新引用。如果 val 为 0，则返回 JSON false，否则返回 JSON true。这是一个宏，相当于 val ? json_true (): json_false ()。

新版本 2.4。

json_t * json_null(void)

返回值:新引用。返回 JSON 空值。

字符串

Jansson 使用 UTF-8 作为字符编码。所有 JSON 字符串必须是有效的 UTF-8(或 ASCII，因为它是 UTF-8 的子集)。允许使用所有的 Unicode 代码点 U+0000 到 U+10FFFF，但是如果希望在字符串中嵌入空字节，则必须使用长度感知函数。

json_t *json_string(const char *value)

返回值:新引用。返回一个新的 JSON 字符串，或在出现错误时返回 NULL。值必须是有效的空终止的 UTF-8 编码的 Unicode 字符串。

json_t *json_stringn(const char *value, size_t len)

返回值:新引用。与 json_string()类似，但是使用显式长度，所以值可以包含空字符，也可以不以空结尾。

新版本 2.7。

json_t *json_string_nocheck(const char *value)

返回值:新引用。与 json_string()类似，但不检查该值是否是有效的 UTF-8。只有在你确定情况确实如此时才使用这个函数(例如，你已经用其他方法检查过它)。

json_t *json_stringn_nocheck(const char *value, size_t len)

返回值:新引用。与 json_string_nocheck()类似，但是使用显式长度，所以值可以包含空字符，也可以不以空结尾。

新版本 2.7。

const char *json_string_value(const json_t *string)

返回以空结尾的 UTF-8 编码的字符串的关联值，如果字符串不是 JSON 字符串，则返回 null。返回的值是只读的，用户不能修改或释放它。只要字符串存在，它就是有效的，也就是说，只要它的引用计数没有下降到零。

size_t json_string_length(const json_t *string)返回 UTF-8 表示的字符串长度，如果字符串不是 JSON 字符串，则返回 0。

新版本 2.7。

int json_string_set(json_t *string, const char *value)

将字符串的关联值设置为 value。值必须是有效的 UTF-8 编码的 Unicode 字符串。成功返回 0，错误返回-1。

int json_string_setn(json_t *string, const char *value, size_t len)

与 json_string_set()类似，但是使用显式长度，所以值可以包含空字符，也可以不以空结尾。

新版本 2.7。

int json_string_set_nocheck(json_t *string, const char *value)

类似于 json_string_set()，但不检查该值是否是有效的 UTF-8。只有在你确定情况确实如此时才使用这个函数(例如，你已经用其他方法检查过它)。

int json_string_setn_nocheck(json_t *string, const char *value, size_t len)

与 json_string_set_nocheck()类似，但是使用显式长度，所以值可以包含空字符，也可以不以空结尾

新版本 2.7。

json_t *json_sprintf(const char *format，…)

json_t *json_vsprintf(const char *format, va_list ap)

返回值:新引用。从格式字符串和变量格构造 JSON 字符串，就像 printf()一样。

新版本 2.11。

数字

JSON 规范只包含一种数字类型“number”。C 编程语言对于整数和浮点数有不同的类型，因此出于实际原因，Jansson 对于这两个数字也有不同的类型。它们分别被称为“整数”和“实数”。有关更多信息，请参见 RFC Conformance(RFC 一致性)。

json_int_t

这是用于存储 JSON 整数值的 C 类型。它表示系统上可用的最宽整数类型。实际上，如果编译器支持 long long，它只是一个 typedef，否则就是 long long。通常，您可以安全地使用普通的 int 来代替 json_int_t，其余的由隐含的 C 整数转换来处理。只有当您知道需要完整的 64 位范围时，才应该显式地使用 json_int_t。

JSON_INTEGER_IS_LONG_LONG

这是一个预处理器变量，如果 json_int_t 是 long long，则它的值为 1，如果是 long，则为 0。它的使用方法如下:

```
#if   JSON_INTEGER_IS_LONG_LONG
/*代码特定的 long long*/
#else
/*代码特定的 long*/
# endif
```

JSON_INTEGER_FORMAT

这是一个宏，它扩展为一个与 json_int_t 对应的 printf()转换说明符，没有前面的%符号，即“lld”或“ld”。这个宏是必需的，因为 json_int_t 的实际类型可以是 long 或 long long，而 printf()需要这两个长度修饰符。

例子:

```
json_int_t x = 123123123;
printf(“x is %”JSON_INTEGER_FORMAT“\n”， x);
```

json_t * json_integer (json_int_t value)

返回值:新引用。返回一个新的 JSON 整数，或在出现错误时返回 NULL。
json_int_t json_integer_value(const json_t *integer)
返回 integer 的关联值，如果 json 不是 JSON 整数，则返回 0。
int json_integer_set(const json_t *integer, json_int_t value)
将 integer 的关联值设置为 value。如果整数不是 JSON 整数，则返回-1。
json_t * json_real(double value)
返回值:新引用。返回一个新的 JSON 实数，或 NULL 错误。
double json_real_value(const json_t *real)
返回 real 的关联值，如果 real 不是 JSON real，则返回 0.0。
int json_real_set(const json_t *real，double value)
将关联的 real 值设置为 value。如果 real 不是 JSON real，则返回-1。
double json_number_value(const json_t *json)
返回 JSON 整数或 JSON 实际 JSON 的关联值，无论实际类型如何转换为 double。如果 json 既不是 json 实数，也不是 json 整数，则返回 0.0。

数组

JSON 数组是其他 JSON 值的有序集合。
json_t * json_array(void)
返回值:新引用。返回一个新的 JSON 数组，或在出现错误时返回 NULL。最初，数组是空的。
size_t json_array_size(const json_t *array)返回数组中的元素数量(类似于 strlen())，如果数组为 NULL 或非 JSON 数组，则返回 0。
json_t *json_array_get(const json_t *array, size_t index)
返回值:返回数组中位置索引处的元素。索引的有效范围是从 0 到（json_array_size() - 1）的返回值。如果数组不是 JSON 数组，如果数组为 NULL，或者如果索引超出范围，则返回 NULL。
int json_array_set(json_t *array, size_t index, json_t *value)
将数组中位置索引中的元素替换为值。索引的有效范围是从 0 到 json_array_size() - 1 的返回值。成功返回 0，错误返回-1。
int json_array_set_new(json_t *array, size_t index, json_t *value)
与 json_array_set()类似，但是窃取了对值的引用。当值是新创建的而不是在调用后使用的时候，这是很有用的。
Int json_array_append(json_t *array, json_t *value)
将值追加到数组的末尾，使数组的大小增加 1。成功返回 0，错误返回-1。
Int json_array_append_new(json_t *array, json_t *value)类似于 json_array_append()，
但是窃取了对值的引用。当值是新创建的而不是在调用后使用的时候，这是很有用的。
Int json_array_insert(json_t *array, size_t index, json_t *value)
在位置索引处向数组插入值，在索引处移动元素，然后将元素移到数组的末尾。成功返回 0，错误返回-1。
Int json_array_insert_new(json_t *array, size_t index, json_t *value)
与 json_array_insert()类似，但是窃取了对值的引用。当值是新创建的而不是在调用后使用的时候，这是很有用的。
Int json_array_remove(json_t *array，size_t index)
删除数组中位置索引处的元素，将元素在索引 1 之后移动到数组的开始位置。成功返回 0，错误返回-1。删除值的引用计数递减。

int json_array_clear (json_t *array)
从数组中删除所有元素。成功返回 0，错误返回-1。所有删除值的引用计数都递减。
int json_array_extend(json_t *array, json_t *other_array)
将 other_array 中的所有元素附加到数组的末尾。成功返回 0，错误返回-1。
json_array_foreach(array, index, value)
遍历数组的每个元素，每次运行后面的代码块，并将适当的值分别设置为类型 size_t 和 json_t *的变量 index 和 value。
例子:

```
/* array 是一个 JSON 数组*/
size_t index;
json_t *value;
json_array_foreach(array, index, value) {  使用索引和值的代码块  }
```

项目按递增索引顺序返回。这个宏在预处理时扩展为一个普通的 for 语句，因此它的性能相当于使用数组访问功能的手写代码等效。这个宏的主要优点是它使复杂性抽象化，并使代码更简洁和可读。
新版本 2.5。

对象

JSON 对象是键-值对的字典，其中键是 Unicode 字符串，值是任何 JSON 值。即使在字符串值中允许空字节，在对象键中也不允许。
json_t * json_object(void)
返回值:新引用。返回一个新的 JSON 对象，或在出现错误时返回 NULL。最初，对象是空的。
size_t json_object_size((const json_t *object)
返回对象中的元素数量，如果对象不是 JSON 对象，则返回 0。
json_t *json_object_get(const json_t *object, const char *key)
返回值:从对象中获取与键对应的值。如果没有找到键值，则返回 NULL。
int json_object_set(json_t *object, const char *key, json_t *value)
将 key 的值设置为 object 中的值。密钥必须是有效的空终止的 UTF-8 编码的 Unicode 字符串。如果已经有一个 for key 值，那么它将被新值替换。成功返回 0，错误返回-1。
json_object_set_nocheck(json_t *object, const char *key, json_t *value)
类似于 json_object_set()，
但不检查键是否是有效的 UTF-8。只有在你确定情况确实如此时才使用这个函数(例如，你已经用其他方法检查过它)。
json_object_set_new(json_t *object, const char *key, json_t *value)
与 json_object_set()类似，
但是窃取对值的引用。当值是新创建的而不是在调用后使用的时候，这是很有用的。

json_object_set_new_nocheck(json_t *object, const char *key, json_t *value)
类似于 json_object_set_new()，
但不检查键是否为有效的 UTF-8。只有在你确定情况确实如此时才使用这个函数(例如，你已经用其他方法检查过它)。
json_object_del(json_t *object, const char *key)

从对象中删除键(如果它存在)。如果成功，返回 0;如果没有找到键，返回-1。删除值的引用计数递减。

int json_object_clear (json_t *对象)

从对象中删除所有元素。如果对象不是 JSON 对象，则返回-1。所有删除值的引用计数都递减。

json_object_update(json_t *object, json_t *other)

使用来自其他的键-值对更新对象，覆盖现有的键。如果成功，返回 0;如果失败，返回-1。

json_object_update_existing(json_t *object, json_t *other)

与 json_object_update()类似，

但是只更新现有键的值。不创建新的密钥。如果成功，返回 0;如果失败，返回-1。

新版本 2.3。

json_object_update_missing(json_t *object, json_t *other)

与 json_object_update()类似，但是只创建新键。任何现有键的值都不会更改。如果成功，返回 0;如果失败，返回-1。

新版本 2.3。

json_object_foreach(对象、键、值)

遍历对象的每个键-值对 object，每次运行后面的代码块，并将相应的值分别设置为 const char *和 json_t *类型的变量 key 和 value。例子:

```
/*obj 是一个 JSON 对象*/
const char *key;
json_t *value;
json_object_foreach(obj, key, value)
{使用键和值的代码块}
```

这些项按插入对象的顺序返回。

注意:在迭代期间调用 json_object_del(object, key)是不安全的。如果需要，可以使用 json_object_foreach_safe()。这个宏在预处理时扩展为普通的 for 语句，因此它的性能相当于使用对象迭代协议(参见下面)编写的迭代代码。这个宏的主要优点是它抽象了迭代背后的复杂性，并使代码更加简洁和可读。

新版本 2.3。

json_object_foreach_safe(object、tmp、key、value)

类似于 json_object_foreach()，但是在迭代期间调用 json_object_del(object, key)是安全的。您需要传递一个额外的 void *参数 tmp，该参数用于临时存储。

新版本 2.8。

以下函数可用于遍历对象中的所有键-值对。项按插入对象的顺序返回。

void * json_object_iter (json_t *object)

返回一个不透明的迭代器，该迭代器可用于遍历对象中的所有键值对，如果对象为空则返回 NULL。

void *json_object_iter_at(json_t *object, const char *key)

与 json_object_iter()类似，但是返回一个迭代器到对象中的键值对，该对象的键值等于 key，如果在对象中没有找到 key，则返回 NULL。如果键恰好是底层哈希表中的第一个键，那么向前迭代到对象末尾只会产生对象的所有键-值对。

void *json_object_iter_next(json_t *object, void *iter)

返回一个迭代器，该迭代器指向 iter 之后对象中的下一个键-值对，如果整个对象已被遍历，则返回 NULL。

const char *json_object_iter_key(void *iter)
从 iter 中提取相关的密钥。
json_t * json_object_iter_value (void * iter)
返回值:借来的参考。从 iter 中提取相关值。
int json_object_iter_set(json_t *object, void *iter, json_t *value)
将对象中由 iter 指向的键-值对的值设置为 value。
int json_object_iter_set_new(json_t *object, void *iter, json_t *value)
与 json_object_iter_set()类似，
但是窃取了对值的引用。当值是新创建的而不是在调用后使用的时候，这是很有用的。
void *json_object_key_to_iter(const char *key)
类似于 json_object_iter_at()，但是要快得多。仅对 json_object_iter_key()返回的值有效。使用其他密钥会导致分段错误。此函数在内部用于实现 json_object_foreach()。例子:

```
/*obj 是一个 JSON 对象*/
const char *key;
json_t *value;
void *iter = json_object_iter(obj);
while(iter)
{
key= json_object_iter_key (iter);
value= json_object_iter_value (iter);
/*使用键和值…* /
iter = json_object_iter_next(obj, iter);
}
```

新版本 2.3。
void json_object_seed (size_t seed)
在 Jansson 的散列表实现中使用散列函数。种子用来随机化哈希函数，这样攻击者就不能控制它的输出。如果 seed 为 0,Jansson 通过从操作系统的熵源读取随机数据来生成 seed。如果没有熵源可用，则返回到使用当前时间戳(如果可能，使用微秒精度)和进程 ID 的组合。如果调用了该函数，则必须在调用 json_object()之前调用该函数，不管是显式的还是隐式的。如果用户没有调用这个函数，那么对 json_object()的第一次调用(显式或隐式)将播下散列函数的种子。有关线程安全的注意事项，请参见线程安全。如果需要可重复的结果，例如单元测试，可以通过在程序启动时调用具有常量值的 json_object_seed()来“非随机化”散列函数，例如 json_object_seed(1)。
新版本 2.6。

错误报告

Jansson 使用单一结构类型将错误信息传递给用户。请参阅使用此结构传递错误信息的函数的解码、构建值、解析和验证值部分。
json_error_t
char text[ ]
错误消息(在 UTF-8 中)，如果消息不可用，则为空字符串。此数组的最后一个字节包含一个数字错误代码。使用 json_error_code()来提取这段代码。

char source[ ]
错误的来源。这可以是文件名的一部分，也可以是尖括号(例如<string>)中的特殊标识符。
Int line
发生错误的行号。
Int column
发生错误的列。注意，这是字符列，而不是字节列，即多字节 UTF-8 字符算作一列。
Int position
从输入开始的位置(以字节为单位)。这对于调试 Unicode 编码问题非常有用。json_error_t 的正常用法是在堆栈上分配它，并将指针传递给函数。例子:

```
int main ()
{
    json_t * json;
    json_error_t error;
    json = json_load_file("/path/to/file.json,0,&error);
    if(! json)
{
    /*错误变量包含错误信息*/
}
…
}
```

还要注意，如果调用成功(在上面的例子中是 json != NULL)，错误的内容通常是未指定的。译码函数写的位置成员也成功。更多信息见解码。
所有函数也接受 NULL 作为 json_error_t 指针，在这种情况下，不会向调用者返回错误信息。
enum json_error_code
包含数字错误代码的枚举。当前定义的错误如下:
json_error_unknown 未知的错误。只对非错误 json_error_t 结构应该返回此值。
json_error_out_of_memory 库不能分配任何堆内存。
json_error_stack_overflow 嵌套太深。
json_error_cannot_open_file 无法打开输入文件。
json_error_invalid_argument 函数参数无效。
json_error_invalid_utf8 输入字符串不是有效的 UTF-8。
json_error_premature_end_of_input 输入以 JSON 值的中间结束。
json_error_end_of_input_expected JSON 值后面有一些文本。查看 JSON_DISABLE_EOF_CHECK 标志。
json_error_invalid_syntax JSON 的语法错误。
json_error_invalid_format 用于打包或解包的格式字符串无效。
json_error_wrong_type 打包或解压缩时，值的实际类型与格式字符串中指定的类型不同。
json_error_null_character 在 JSON 字符串中检测到空字符。查看 JSON_ALLOW_NUL 标志。
json_error_null_value 在打包或解包时，某些键或值为空。
json_error_null_byte_in_key 对象键将包含一个空字节。Jansson 不能表示这样的键;请参阅 RFC 一致性。
json_error_duplicate_key 对象中的键重复。看到 json_reject_duplicates 标志。
json_error_numeric_overflow 将 JSON 数字转换为 C 数字类型时，会检测到数字溢出。

json_error_item_not_found 未找到键入的对象。
json_error_index_out_of_range 数组索引超出范围。

新版本 2.11。

enum json_error_code json_error_code(const json_error_t *error)
返回嵌入在 error->文本中的错误代码。

新版本 2.11。

编码

本节描述可用于将值编码为 JSON 的函数。默认情况下，只有对象和数组可以直接编码，因为它们是 JSON 文本中惟一有效的根值。要编码任何 JSON 值，请使用 JSON_ENCODE_ANY 标志(参见下文)。
默认情况下，输出没有换行，数组和对象元素之间使用空格作为可读输出。可以使用下面描述的 JSON_INDENT 和 JSON_COMPACT 标志更改此行为。换行永远不会附加到经过编码的 JSON 数据的末尾。每个函数都有一个 flags 参数，该参数控制数据编码的某些方面。它的默认值是 0。下面的宏可以一起得到标记。
JSON_INDENT (n)
使用数组和对象项之间的新行，并使用 n 个空格缩进，以漂亮的方式打印结果。n 的有效范围是 0 到 31(包括 31)，其他值会产生未定义的输出。如果不使用 JSON_INDENT 或 n 为 0，则数组和对象项之间不插入新行。
JSON_MAX_INDENT 常量定义了可以使用的最大缩进，其值为 31。
版本 2.7 的变化:添加了 JSON_MAX_INDENT。
JSON_COMPACT
该标志支持紧凑表示，即将数组和对象项之间的分隔符设置为“,”，将对象键和值之间的分隔符设置为“:”。如果没有此标志，则对应的分隔符是“，”和“: ”，以便获得更可读的输出。
JSON_ENSURE_ASCII
如果使用此标志，则保证输出只包含 ASCII 字符。这是通过转义 ASCII 范围之外的所有 Unicode 字符来实现的。
JSON_SORT_KEYS
如果使用此标志，则输出中的所有对象都按键排序。这是有用的，例如，如果两个 JSON 文本是不同的或在视觉上比较。
JSON_PRESERVE_ORDER
从 2.8 版本开始就被弃用:对象键的顺序始终保持不变。在版本 2.8 之前:如果使用这个标志，输出中的对象键将按照它们第一次插入对象的顺序排序。
例如，对 JSON 文本进行解码，
然后使用此标记进行编码，以保留对象键的顺序。
JSON_ENCODE_ANY
指定这个标志可以对任何 JSON 值进行编码。没有它，只有对象和数组可以作为 json 值传递给编码函数。注意:在某些情况下，编码任何值都是有用的，但通常不鼓励这样做，因为它违反了与 RFC 4627 的严格兼容性。如果使用这个标记，就不要期望与其他 JSON 系统具有互

操作性。
新版本 2.1。
JSON_ESCAPE_SLASH
用 \/ 转义字符串中的 / 字符。
新版本 2.4。
JSON_REAL_PRECISION (n)
输出精度不超过 n 位的所有实数。n 的有效范围是 0 到 31(包括 31)，其他值会导致未定义的行为。默认情况下，精度为 17，可以正确无损地编码所有 IEEE 754 双精度浮点数。
新版本 2.7。
JSON_EMBED
如果使用此标志，则在编码期间省略顶级数组('['，']')或对象('{'，'}')的开始和结束字符。当将多个数组或对象连接到流中时，此标志非常有用。
新版本 2.10。
这些函数输出 UTF-8:
char *json_dumps(const json_t *json, size_t flags)
返回 JSON 作为字符串的 JSON 表示形式，或在出现错误时返回 NULL。上面已经描述了标记。调用者必须使用 free()释放返回值。注意，如果您已经调用了 json_set_alloc_funcs()来覆盖 free()，那么您应该调用定制的 free 函数来释放返回值。
size_t json_dumpb(const json_t *json, char *buffer, size_t size, size_t flags)
将 JSON 的 JSON 表示形式写入大小为字节的缓冲区。返回将写入的字节数或出错时为 0 的字节数。上面已经描述了标记。缓冲区不是以空值结尾的。此函数写入的字节数从不超过大小。如果返回值大于 size，则缓冲区的内容是未定义的。此行为使您能够指定空缓冲区来确定编码的长度。
例如:

```
size_t size = json_dumpb(json, NULL, 0, 0);
if (size == 0)
return -1;
char *buf = alloca(size);
size = json_dumpb(json, buf, size, 0);
```

新版本 2.10。
int json_dumpf(const json_t *json, FILE *output, size_t flags)
将 JSON 的 JSON 表示形式写入流输出。上面已经描述了标记。成功返回 0，错误返回-1。如果发生错误，可能已经将某些内容写入了输出。在这种情况下，输出是未定义的，很可能不是有效的 JSON。
int json_dumpfd(const json_t *json, int output, size_t flags)
将 JSON 的 JSON 表示形式写入流输出。上面已经描述了标记。成功返回 0，错误返回-1。如果发生错误，可能已经将某些内容写入了输出。在这种情况下，输出是未定义的，很可能不是有效的 JSON。需要注意的是，这个函数只能在流文件描述符(例如 SOCK_STREAM)上成功执行。在非流文件描述符上使用此函数将导致未定义的行为。对于非流文件描述符，请参见 json_dumpb()。这个函数需要 POSIX，但在所有非 POSIX 系统上都失败了。
新版本 2.10。
int json_dump_file(const json_t *json, const char *path, size_t flags)

将 JSON 的 JSON 表示形式写入文件路径。如果路径已经存在，它将被覆盖。上面已经描述了标记。成功返回 0，错误返回-1。

json_dump_callback_t

一种调用的函数的类型定义 json_dump_callback()：

typedef int (*json_dump_callback_t)(const char *buffer, size_t size, void *data);

缓冲区指向一个包含输出块的缓冲区，size 是缓冲区的长度，data 是通过的对应 json_dump_callback()参数。缓冲区保证是一个有效的 UTF-8 字符串(即保留多字节码单元序列)。缓冲区从不包含嵌入的空字节。如果出现错误，函数应该返回-1 来停止编码过程。如果成功，应该返回 0。

新版本 2.2。

int json_dump_callback(const json_t*json, json_dump_callback_t callback,void*data,size_t flags)

重复调用回调，每次传递一个 JSON 表示的块。上面已经描述了标记。成功返回 0，错误返回-1。

新版本 2.2。

解码

本节描述可以用于将 JSON 文本解码为 JSON 数据的 Jansson 表示形式的函数。

JSON 规范要求 JSON 文本要么是序列化的数组，要么是对象，这一要求也通过以下函数得以实现。换句话说，要解码的 JSON 文本中的顶级值必须是数组或对象。要解码任何 JSON 值，请使用 JSON_DECODE_ANY 标志(参见下面)。有关 Jansson 对 JSON 规范的一致性的讨论，请参阅 RFC 一致性。它解释了许多设计决策，特别是影响的行为解码器。每个函数都接受一个 flags 参数，该参数可用于控制解码器的行为。它的默认值是 0。下面的宏可以一起得到标记。

JSON_REJECT_DUPLICATES

如果输入文本中的任何 JSON 对象包含重复的键，则发出一个解码错误。如果没有此标志，每个键的最后一次出现的值将出现在结果中。逐个字节检查键等价性，不使用特殊的 Unicode 比较算法。

新版本 2.1。

JSON_DECODE_ANY

默认情况下，解码器期望一个数组或对象作为输入。启用此标志后，解码器将接受任何有效的 JSON 值。注意:在某些情况下，解码任何值都是有用的，但通常不鼓励这样做，因为它违反了与 RFC 4627 的严格兼容性。如果使用这个标记，就不要期望与其他 JSON 系统具有互操作性。

新版本 2.3。

JSON_DISABLE_EOF_CHECK

默认情况下，解码器期望其整个输入构成一个有效的 JSON 文本，如果在其他有效的 JSON 输入之后有额外的数据，则会发出一个错误。启用此标记后，解码器将在对一个有效的 JSON 数组或对象进行解码后停止，因此允许在 JSON 文本之后添加额外的数据。通常，当遇到 JSON 输入中的最后一个] 或 } 时，读取将停止。如果使用 JSON_DISABLE_EOF_CHECK 和 JSON_DECODE_ANY 标志，则解码器可以读取一个额外的 UTF-8 代码单元(最多 4 个字节的输入)。例如，解码 4true 可以正确地解码整数 4，但也可以读取 t。因此，如果读取多个不是数组或对象的连续值，它们之间应该至少用一个空白字符隔开。

新版本 2.1。

JSON_DECODE_INT_AS_REALJSON

只定义了一种数字类型。Jansson 区分了 int 和 reals。有关更多信息，请参见实数和整数。有了这个标志，解码器就可以将所有的数字解释为实数。没有精确双精度表示的整数将导致精度损失。导致双重溢出的整数将导致错误。

新版本 2.5。

JSON_ALLOW_NUL

允许\u0000 在字符串值中转义。这是一种安全措施;如果您知道您的输入可以包含空字节，请使用此标志。如果你不使用这个标志，你不必担心字符串里面的空字节，除非你使用 json_stringn()或者 json_pack()的 s#格式说明符来显式地创建它们。即使使用此标志，对象键也不能嵌入空字节。

新版本 2.6。

每个函数还接受一个可选的 json_error_t 参数，如果解码失败，该参数将填充错误信息。它也更新成功;将输入读的字节数写入其位置字段。这在使用 JSON_DISABLE_EOF_CHECK 读取多个连续的 JSON 文本时特别有用。

新版本 2.3:

将输入读的字节数写入 json_error_t 结构的位置字段。如果不需要错误或位置信息，则可以传递 NULL。

json_t *json_loads(const char *input, size_t flags, json_error_t *error)

返回值:新引用。对 JSON 字符串输入进行解码并返回它所包含的数组或对象，或 NULL on error，在这种情况下，error 将被有关错误的信息填充。上面已经描述了标记。

json_t *json_loadb(const char *buffer, size_t buflen, size_t flags, json_error_t *error)

返回值:新引用。对长度为 buflen 的 JSON 字符串缓冲区进行解码，并返回它所包含的数组或对象，或 NULL on error，在这种情况下，error 将被有关错误的信息填充。这类似于 json_loads()，只是字符串不需要以 null 结尾。上面已经描述了标记。

新版本 2.1。

json_t *json_loadf(FILE *input, size_t flags, json_error_t *error)

返回值:新引用。对流输入中的 JSON 文本进行解码，并返回它所包含的数组或对象，或在出现错误时返回 NULL，在这种情况下，错误将被有关错误的信息填充。上面已经描述了标记。这个函数将从输入文件所在的任何位置开始读取输入，而不需要首先尝试查找。如果发生错误，文件位置将不确定。如果成功，文件位置将位于 EOF，除非使用了 JSON_DISABLE_EOF_CHECK 标志。在这种情况下，文件位置将位于 JSON 输入中最后一个]或}之后的第一个字符。这允许对同一个文件对象多次调用 json_loadf()，如果输入由连续的 JSON 文本组成，可能用空格分隔。

json_t *json_loadfd(int input, size_t flags, json_error_t *error)

返回值:新引用。对流输入中的 JSON 文本进行解码，并返回它所包含的数组或对象，或在出现错误时返回 NULL，在这种情况下，错误将被有关错误的信息填充。上面已经描述了标记。这个函数将从输入文件描述符所在的位置开始读取输入，而不需要首先尝试查找。如果发生错误，文件位置将不确定。如果成功，文件位置将位于 EOF，除非使用了 JSON_DISABLE_EOF_CHECK 标志。在这种情况下，文件描述符的位置将位于 JSON 输入中最后一个]或}之后的第一个字符。这允许在相同的文件描述符上多次调用 json_loadfd()，如果输入由连续的 JSON 文本组成，可能用空格分隔。需要注意的是，这个函数只能在流文件描述符(例如 SOCK_STREAM)上成功执行。在非流文件描述符上使用此函数将导致未定义的行为。对于非流文件描述符，请参见 json_loadb()。这个函数需要 POSIX，但在所有非 POSIX 系统上都失败了。

新版本 2.10。

json_t *json_load_file(const char *path, size_t flags，json_error_t *error)
返回值:新引用。解码文件路径中的 JSON 文本并返回它所包含的数组或对象，或返回 NULL on error，在这种情况下，error 将用有关错误的信息填充。上面已经描述了标记。
json_load_callback_t
调用 json_load_callback()的函数的类型定义，用于读取输入数据块:
typedef size_t (*json_load_callback_t)(void *buffer, size_t buflen, void *data);
buffer 指向一个 buflen 字节的缓冲区，数据是相应的 json_load_callback()参数。
如果成功，函数应该写入最多 buflen 字节来缓冲，并返回写入的字节数;返回值 0 表示没有产生任何数据，并且已经到达文件的末尾。如果出现错误，该函数应该返回(size_t)-1 来终止解码过程。在 UTF-8 中，一些代码点被编码为多字节序列。回调函数不需要担心这个问题，因为 Jansson 在更高的级别上处理它。例如，您可以安全地从网络连接读取固定数量的字节，而不必关心由块边界分割的代码单元序列。

新版本 2.4。
json_t*json_load_callback(json_load_callback_t callback, void *data, size_t flags, json_error_t *error)
返回值:新引用。解码对回调的重复调用产生的 JSON 文本，并返回它所包含的数组或对象，或 NULL on error，在这种情况下，error 将用有关错误的信息填充。每次调用时都将数据传递给回调。上面已经描述了标记。

新版本 2.4。

构建值

本节描述帮助创建或打包复杂 JSON 值的函数，特别是嵌套的对象和数组。值构建基于一个格式字符串，该字符串用于告诉函数有关预期参数的信息。例如，格式字符串“i”指定一个整数值，而格式字符串“[ssb]”或等效的“[s, s, b]”指定一个数组值，其中包含两个字符串和一个布尔值:
/*创建 JSON 整数 42 */
json_pack(“i”,42);
/*创建 JSON 数组["foo"， "bar"， true] */
json_pack("[ssb]"， "foo"， "bar"， 1);
下面是格式说明符的完整列表。圆括号中的类型表示得到的 JSON 类型，括号中的类型(如果有的话)表示作为相应参数的 C 类型。
s (string) [const char *]
将空终止的 UTF-8 字符串转换为 JSON 字符串。
s? (string) [const char *]类似于 s，但如果参数为 NULL，则输出 JSON NULL 值。
新版本 2.8。
s* (string) [const char *]类似于 s，但如果参数为空，则不输出任何值。这种格式只能在对象或数组中使用。如果在对象中使用，则当值被省略时，相应的键将被取消。参见下面的示例。
新版本 2.11。

s# (string) [const char *， int]将给定长度的 UTF-8 缓冲区转换为 JSON 字符串。
新版本 2.5。
s%(字符串)[const char *， size_t]类似于 s#，但是长度参数的类型是 size_t。
新版本 2.6。
+ [const char *] 像 s，但是连接到前面的字符串。仅在 s、s#、+或+#之后有效。
新版本 2.5。
+# [const char *， int]类似于 s#，但是连接到前面的字符串。仅在 s、s#、+或+#之后有效。
新版本 2.5。
+%(string)[const char *， size_t]比如+#，但是长度参数的类型是 size_t。
新版本 2.6。
n(NULL)输出一个 JSON 空值。没有消耗任何参数。
b (boolean)[int]将 C int 转换为 JSON 布尔值。将 0 转换为 false，将非 0 转换为 true。
i (integer)[int]将一个 C 整数转换成 JSON 整数。
I (integer)[json_int_t]将 C json_int_t 转换为 JSON 整数。
f (real)[double]将 C double 转换为 JSON real。
o(any value)[json_t *]输出任意给定的 JSON 值。如果将值添加到数组或对象中，则容器会窃取对传递给 o 的值的引用。
O(any value)[json_t *]类似于 o，但是参数的引用计数是递增的。如果您将其打包到一个数组或对象中，并且希望自己保留 O 使用的 JSON 值的引用，那么这是非常有用的。
o?,o?(any value)[json_t *]类似于 o 和 O，但如果参数为 NULL，则输出 JSON NULL 值。
新版本 2.8。
o*,O*(any value)[json_t *]类似于 o 和 O，但如果参数为 NULL，则不输出任何值。这种格式只能在对象或数组中使用。如果在对象内部使用，相应的键将被额外抑制。参见下面的示例。
新版本 2.11。
(fmt)(array)用内部格式字符串的内容构建一个数组。fmt 可以包含对象和数组，即支持递归值构建。
{fmt}(object)用内部格式字符串 fmt 的内容构建一个对象。第一个、第三个等格式说明符表示一个键，并且必须是一个字符串(见上面的 s、s#、+和+#)，因为对象键总是字符串。第二、第四等格式说明符表示一个值。任何值都可以是对象或数组，即支持递归值构建。将忽略空格（ ），冒号（:），逗号（,）。

json_t *json_pack(const char *fmt，…)返回值:新引用。根据格式字符串 fmt 构建一个新的 JSON 值。对于每个格式说明符(除了{}[]n)，将使用一个或多个参数来构建相应的值。返回 NULL 错误。
json_t *json_pack_ex(json_error_t *错误，size_t 标志，const char *fmt，…)(json_error_t *错误，size_t 标志，const char *fmt, va_list ap)
返回值:新引用。与 json_pack()类似，但在出现错误的情况下，如果不是 NULL，则将错误消息写入 error。flags 参数当前未使用，应该设置为 0。由于只有格式字符串中的错误(以及内存不足的错误)可以被封隔器捕获，所以这两个函数很可能只对调试格式字符串有用。

更多的例子:
/*构建一个空的 JSON 对象*/
json_pack (“{}”);
/*构建 JSON 对象{"foo": 42， "bar": 7} */

```
json_pack("{sisi}"， "foo"， 42， "bar"， 7);
/*像上面一样，':'，'，'和空白被忽略*/
json_pack("{s:i, s:i}"， "foo"， 42， "bar"， 7);
/*构建 JSON 数组[[1,2]，{"cool": true}] */
json_pack("[[i,i]，{s:b}]"， 1,2， "cool"， 1);
/*从非空终止的缓冲区*/
构建一个字符串 char buffer[4] = {'t'， 'e'， 's'， 't'};json_pack(“s#”,buffer,4);
/*将字符串连接在一起，构建 JSON 字符串“foobarbaz”*/
json_pack("s++"， "foo"， "bar"， "baz");
/*创建一个空的对象或数组时，可选的成员是失踪*/
json_pack(“{s:s*,s:o*, s:o*}”,“foo”,NULL,“bar”,NULL,“baz”,NULL);
json_pack("[s*,o*,o*]"， NULL, NULL, NULL);
```

解析和验证值

本节描述一些函数，这些函数帮助验证复杂的值并从中提取或解包数据。与构建值一样，这也是基于格式字符串的。在对 JSON 值进行解压缩时，将检查格式字符串中指定的类型，以匹配 JSON 值。这是验证过程的一部分。除此之外，解包函数还可以检查数组和对象的所有项是否被解包。使用格式说明符启用此检查!或者使用 JSON_STRICT 标志。详见下文。下面是格式说明符的完整列表。圆括号中的类型表示 JSON 类型，括号中的类型(如果有)表示应该传递其地址的 C 类型。

s (string) [const char *]

将一个 JSON 字符串转换为一个指向以空结尾的 UTF-8 字符串的指针。结果字符串是通过内部使用 json_string_value()提取的，因此只要仍然有对相应 JSON 字符串的引用，它就会存在。

s% (string)[const char *， size_t *]

将一个 JSON 字符串转换为一个指向空结束的 UTF-8 字符串及其长度的指针。

新版本 2.6。

n (null)

期望 JSON 空值。没有提取。

b (boolean)(int)

将 JSON boolean 值转换为 C int，这样 true 就转换为 1,false 转换为 0。

i (integer)(int)

将 JSON 整数转换为 C int。

I (integer)(json_int_t)将 JSON 整数转换为 C json_int_t。

f (real) [double]将 JSON real 转换为 C double。

F (integer or real) [double]将 JSON 数字(整数或实数)转换为 C double。

o (any value) [json_t *]存储不转换为 json_t 指针的 JSON 值。

O (any value) [json_t *]与 o 类似，但是 JSON 值的引用计数是递增的。存储指针在使用 unpack 之前应该初始化为 NULL。调用者负责释放所有因解包而增加的引用，即使发生错误也是如此。

[fmt] (array)根据内部格式字符串转换 JSON 数组中的每一项。fmt 可以包含对象和数组，即支持递归值提取。

{fmt} (object)根据内部格式字符串 fmt 转换 JSON 对象中的每个项目。第一个、第三个等格式说明符表示一个键，并且必须是 s。解包函数的相应参数被读取为对象键。第二个、第四

个等格式说明符表示一个值，并作为相应的参数写入给定的地址。请注意，其他每个参数都是从其中读取的，而其他每个参数都是写入的。fmt 可以包含对象和数组作为值，即支持递归值提取。

新版本 2.3:

任何表示密钥的 s 都可以加后缀 a ?使键可选。如果没有找到密钥，则不提取任何内容。参见下面的示例。

! 这个特殊的格式说明符用于在每个值的基础上检查是否访问了所有对象和数组项。它必须作为最后一个格式说明符出现在数组或对象中，然后是右括号或大括号。要全局启用检查，请使用 JSON_STRICT 解压标志。

* 这个特殊的格式说明符是!的反义词。如果使用 JSON_STRICT 标志，则可以使用*在每个值的基础上禁用严格检查。它必须作为最后一个格式说明符出现在数组或对象中，然后是右括号或大括号。将忽略 空格 : 和 ,

json_unpack(json_t *root，const char *fmt，…)

根据格式字符串 fmt 验证和解压 JSON 值根。成功返回 0，失败返回-1。

json_unpack_ex(json_t *root，json_error_t *error，size_t flags，const char *fmt，…)

json_unpack_ex(json_t *root，json_error_t *error，size_tflags，const char *fmt, va_list ap)

根据格式字符串 fmt 验证和解压 JSON 值根。如果发生错误且错误不为空，则将错误信息写入 error。标志可以用来控制行为的 unpacker，见下文以了解标志。成功返回 0，失败返回-1。

请注意

所有解包函数的第一个参数是 json_t *root，而不是 const json_t *root，因为使用 O 格式说明符会增加 root 的引用计数，或者从 root 中可以访问的一些值。此外，可以使用 o 格式说明符按原样提取值，这允许修改从根访问的值的结构或内容。

如果没有使用 O 和 o 格式说明符，那么在与这些函数一起使用时，将 const json_t *变量转换为普通的 json_t *是完全安全的。下列的拆包标志可供选择:

JSON_STRICT

启用额外的验证步骤，检查所有对象和数组项是否已解压缩。这相当于附加格式说明符!到格式字符串中每个数组和对象的末尾。

JSON_VALIDATE_ONLY

不要提取任何数据，只需根据给定的格式字符串验证 JSON 值即可。请注意，对象键仍然必须在格式字符串之后指定。

例子:

```
/* root 是 JSON 整数 42 */
Int myint;
json_unpack(root, "i", &myint);
assert(myint == 42);
/*根是 JSON 对象{"foo": "bar"，  "quux": true} */
const char *str;
  int boolean;
  json_unpack(root, "{s:s, s:b}", "foo", &str, "quux", &boolean);
  assert(strcmp(str, "bar") == 0 && boolean == 1);
/*根是 JSON 数组[[1,2]，{"baz": null} */
json_error_t error; json_unpack_ex(root, &error, JSON_VALIDATE_ONLY, "[[i,i], {s:n}]", "baz");
```

/*返回 0 表示验证成功，没有提取任何内容*/
/*根是 JSON 数组[1,2,3,4,5]*/
int myint1 myint2;
json_unpack(root,”[ii!]”,&myint1 &myint2);
/ *如果验证失败，返回-1 */
/* root 是一个空的 JSON 对象*/
int myint = 0, myint2 = 0, myint3 = 0;json_unpack(root, "{s?i, s?[ii]}",“foo”,&myint1,“bar”,&myint2 &myint3);
/* myint1, myint2 或 myint3 不被触摸，因为"foo"和"bar"不存在*/

相等

不能使用==操作符来测试两个 JSON 值是否相等。==操作符中的等式表示两个 json_t 指针指向完全相同的 JSON 值。但是，两个 JSON 值不仅可以是相同的值，而且还可以是相同的“内容”:

1. 如果包含的数值相等，则两个整数值或实数值相等。但是，整数值永远不等于实值。
2. 如果两个字符串包含的 UTF-8 字符串逐字节相等，则它们相等。Unicode 比较算法未实现。
3. 如果两个数组的元素数量相同，且第一个数组中的每个元素与第二个数组中的相应元素相同，则两个数组相等。
4. 如果两个对象具有完全相同的键，并且第一个对象中每个键的值等于第二个对象中相应键的值，则两个对象是相等的。
5. 两个 true、false 或 null 值没有“内容”，所以如果它们的类型相等，它们就是相等的。(因为这些值是单例的，它们的相等性实际上可以用==来测试。)

int json_equal(json_t *value1, json_t *value2)
如果 value1 和 value2 相等，则返回 1，如上所定义。如果它们不相等，或者一个或两个指针都为空，则返回 0。

拷贝

由于引用计数，传递 JSON 值不需要复制它们。但有时需要一个 JSON 值的新拷贝。例如，如果您需要修改一个数组，但仍然希望在以后使用原始的数组，则应该首先获取它的副本。Jansson 支持两种复制:浅拷贝和深拷贝。这些方法只有在数组和对象之间有区别。浅复制只复制第一级的值(数组或对象)，并在复制的值中使用相同的子值。深度复制也会创建子值的新副本。此外，所有的子值都以递归方式进行深度复制。复制对象保留了键的插入顺序。

json_t * json_copy (json_t *value)
返回值:新引用。返回值的浅拷贝，或在出现错误时返回 NULL。
json_t *json_deep_copy(const json_t *value)
返回值:新引用。返回值的深度副本，或在出现错误时返回 NULL。

自定义的内存分配

默认情况下 Jansson 使用 malloc()和 free()分配内存。如果需要自定义行为，可以覆盖这些函数。

json_malloc_t

具有 malloc()签名的函数指针的 typedef：
typedef void * (* json_malloc_t) (size_t);
json_free_t
具有 free()签名的函数指针的 typedef：
typedef void (*json_free_t)(void *);

void json_set_alloc_funcs(json_malloc_t malloc_fn, json_free_t free_fn)
使用 malloc_fn 代替 malloc()，使用 free_fn 代替 free()。这个函数必须在任何其他 Jansson 的 API 函数之前调用，以确保所有内存操作都使用相同的函数。
void json_get_alloc_funcs(json_malloc_t *malloc_fn, json_free_t *free_fn)
获取当前使用的 malloc_fn 和 free_fn。两个参数都可以为空。

2.8 版的新功能。
例子:
使用应用程序的 malloc()和 free()来绕过 Windows 上不同 CRT 堆的问题:
json_set_alloc_funcs (malloc,free);
使用 Boehm 的保守垃圾收集器进行内存操作:
json_set_alloc_funcs (GC_malloc, GC_free);
通过释放时将所有内存清零，允许将敏感数据（例如密码或加密密钥）存储在 JSON 结构中：

```
static void *secure_malloc(size_t size)
 {
      /* Store the memory area size in the beginning of the block */
      void *ptr = malloc(size + 8);
      *((size_t *)ptr) = size;
       return ptr + 8;
}
static void secure_free(void *ptr)
 {
      size_t size;
      ptr -= 8;
      size = *((size_t *)ptr);
      guaranteed_memset(ptr, 0, size + 8);
      free(ptr);
 }
int main()
 {
       json_set_alloc_funcs(secure_malloc, secure_free);
      /* ... */
}
```

有关在内存中存储敏感数据的更多信息，请参见 http://www.dwheeler.com/secu-programs/secu-programs-howto/protect-secrets.html。该页面还解释了示例中使用的 guaranteed_memset()函数，并给出了示例实现.

RFC 一致性

JSON 在中指定 RFC 4627，“JavaScript 对象符号（JSON）的 application / json 媒体类型”。

字符编码

Jansson 仅支持 UTF-8 编码的 JSON 文本。它不支持或自动检测 RFC 中提到的任何其他编码，即 UTF-16LE，UTF-16BE，UTF-32LE 或 UTF-32BE。支持纯 ASCII，因为它是 UTF-8 的子集。

字符串

JSON 字符串映射到 C 样式的以 null 终止的字符数组，并且内部使用 UTF-8 编码。
字符串值中允许所有 Unicode 代码点 U + 0000 至 U + 10FFFF。但是，由于 API 限制，对象键中不允许使用 U + 0000。
绝对不会对任何字符串（字符串值或对象键）执行 Unicode 规范化或任何其他转换。在检查字符串或对象键的等效性时，将在字符串的原始 UTF-8 表示形式之间逐字节进行比较。

数字

实数与整数

JSON 不区分实数和整数。詹森做到了。实数映射到 double 类型，整数映射到类型，json_int_t 类型是或 的 typedef ，具体取决于编译器是否支持。long longlonglong long
甲 JSON 数目被认为是一个实数，如果其词汇表示包括一个 e，E 或.; 如果不考虑它的实际数值是一个真正的整数（例如，所有的 1E6， 3.0，400E-2，和 3.14E3 是数学整数，但将被视为实际值）。JSON_DECODE_INT_AS_REAL 设置了解码器标志后，所有数字均被解释为实数。所有其他 JSON 数字均视为整数。
当编码为 JSON 时，实数值总是用小数部分表示。例如，double 值 3.0 将在 JSON 中表示为 3.0，而不是 3。

溢出，下溢和精度

绝对值太小而无法用 C 表示的实数 double 将被静默估计为 0.0。因此，根据平台，非常接近零的 JSON 数字（例如 1E-999）可能会导致 0.0。
绝对值太大而不能用 C 表示的实数 double 将导致溢出错误（JSON 解码错误）。因此，根据平台的不同，JSON 数字（例如 1E + 999 或-1E + 999）可能会导致解析错误。
同样，其绝对值太大而无法在 json_int_t 类型中表示的整数（请参见上文）将导致溢出错误（JSON 解码错误）。因此，根据平台的不同，JSON 数字（例如 1000000000000000）可能会导致解析错误。
解析 JSON 实数可能会导致精度损失。只要不发生溢出（即完全失去精度），就将舍入使用近似值。因此，根据平台的不同，JSON 编号 1.000000000000000005 的 double 值可能为 1.0。

签名零

JSON 没有声明数字的含义；但是 Javascript（ECMAscript）确实声明+0.0 和-0.0 必须视为不同的值，即-0.0≠0.0。Jansson 依赖于其在其中编译的 C 环境中的基础浮点库。因此，0.0 和-0.0 是否为不同值取决于平台。使用 IEEE 754 浮点标准的大多数平台将支持带符号的零。
注意，这仅适用于浮点数。JSON，C 或 IEEE 都不支持有符号整数零的概念。

类型

Jansson 不提供对 json_int_t 和以外的任何 C 数字类型的支持 double。这不包括的东西，如无符号类型，等。显然，更短的类型的喜欢，，（如果是），并 隐含地经由普通的 C 类型强制规则处理（视情况而溢出语义）。另外，没有为任何补充的“ bignum”类型附加软件包提供支持或挂钩。long doubleshortintlongjson_int_tlong longfloat

嵌套值的深度

为了避免堆耗尽，Jansson 的目前限制了嵌套深度为阵列和对象到一定值（默认值：2048），其定义为一个宏 JSON_PARSER_MAX_DEPTH 内 jansson_config.h。
该限制允许由 RFC 设置；没有建议的值或要求的最小深度支持