# Chipmunk2D中文手册

Chipmunk2D中文手册，由泰然翻译组翻译。转载请著名出处。
翻译：ChildhoodAndy（完成了大部分的翻译）， uouo， gloryming。
校对：涵紫

github贡献地址：https://github.com/iTyran/ChipmunkDocsCN
欢迎大家斧正错误，提交PR。

## Chipmunk2D 6.2.1

Chipmunk2D是一个基于MIT协议的2D刚体物理仿真库。设计宗旨:极快、可移植、稳定、易用。出于这个原因，它已经被用于数以百计的游戏，而且几乎横跨了所有系统。这些游戏包括了iPhoneAppStore上一些顶级出色的TOP1游戏，如*Night Sky*等。这几年来，我投入了大量的时间来发展Chipmunk，才使得Chipmunk走到今天。如果您发现Chipmunk2D为您节省了许多时间，不妨考虑捐赠下。这么做会使一个独立游戏制作者非常开心！

首先，我要非常感谢ErinCatto（译者注：Box2D作者），早在2006年的时候，Chipmunk的冲量求解器便是受到他的范例代码的启发而完成（现在已经发展成一个成熟的物理引擎：Box2D.org)。他持久接触的想法允许对象的稳定堆栈只进行极少的求解器迭代，而我以前的求解器为了让模拟稳定模拟会产生大量的对象或者会消耗大量的CPU资源。

### 为什么是一个C库

很多人问我为什么用C来写Chipmunk2D，而不是一个我喜欢的其他语言。我通常会对不同的编程语言很兴奋，几个月来，挑选的语言有Scheme, OCaml, Ruby, Objective-C, ooc, Lua, Io等等。它们都有一个共同点，那就是都很容易绑定到C代码。同时我也希望Chipmunk2D高效、易移植、优化简单并且容易调试，而使用C语言就能很简单的达到这些目标。

我从来没有，将来也不太可能去用C来写一个完整的游戏。这里有很多比C有趣的语言，它们有垃圾回收，闭包，面向对象运行时等高级特性。如果你在其它语言中使用Chipmunk2D，可以在Bindings and Ports中找到有用的信息。因为Chipmunk2D基于C99的字集编写，使得它很容易集成到C、C++、Object-C等其它开发语言中。

### C API的局限

如果您使用的是C++，Chipmunk提供了操作符*，+和 – （一元和二元）的重载，但如果使用的是C，那么需要退回使用cpvadd() 和 cpvsub()。这有一点点不利于代码阅读，不过当你习惯之后这将不是个问题。大部分的向量操作可能并没任何形式的符号对应（至少不在键盘上）。

C API的另一个问题是访问限制。Chipmunk有许多结构体，字段，函数只能内部使用。要解决这个问题，我把Chipmunk的全部私有API分离到头文件chipmunk_private.h中，同时在共有结构中使用

CP_PRIVATE()来改名。你可以通过包含这个头文件或使用这个宏来自由访问私有API，但请注意这些私有API可能在未来版本中改变或消失，并且不会在文档中体现，同时也没有私有API的文档计划。

## Chipmunk2D Pro

我们同时在出售Chipmunk2D的扩展版本： Chipmunk2D Pro。主要的特性有：ARM和NEON指令优化，多线程优化，一个为iOS/Mac开发提供的Objective-C封装层，以及自动几何工具。优化主要集中在提高移动性能，同时多线程特性能在支持pthread的平台运行。Objective-C封装层能让你无缝整合到Cocos2D或UIKit等框架，并能获得本地内存管理的优势（包括ARC）。同时Pro版本有大量优秀的API扩展。自动几何工具让你能从图像数据或程序生成并使用几何。

另外，出售Chipmunk2D Pro让我们得以生存，并保持Chipmunk2D的开源。捐献也能棒，但是购买Pro版本你将获得捐献之外的某些东西。

## 下载与编译

如果你还没有下载，你总可以在这里获取到Chipmunk2D的最新版本。里面包含了CMake的命令行编译脚本, Xcode工程以及Visual Studio '09 和 '10工程。

## Debug 或 Release?

Debug模式可能略慢，但是包含了大量的错误检测断言，可以帮助你快速定位类似重复移除对象或无法检测的碰撞之类的BUG。我强烈建议你使用Debug模式，直到你的游戏即将Release发售。

## XCode (Mac/iPhone)

源码中的Xcode工程可直接build出一个Mac或iOS静态库。另外，你可以运行**macosx/iphonestatic.command**或**macosx/macstatic.command**来生成一个带头文件和debug/release静态库的目录，以便你可以方便的集成到你的项目中。直接在你的项目中引入Chipmunk源码以及正确的编译选项并非易事。iPhone编译脚本能生成一个可用在iOS模拟器和设备的通用库（"fat" library），其中的模拟器版本用的debug模式编译，而设备版本用的release模式编译。

## MSVC

我很少使用MSVC，其他开发者帮忙维护了Visual Studio工程文件。MSVC 10工程应该能正常运行，因为我经常在发布稳定版本前测试它。MSVC 9工程可能运行不正常，我很少也没有必要去运行这个工程，如何你遇到问题，请通知我。

## 命令行

CMake编译脚本能在任何你安装了CMake的系统上运行。它甚至能生成XCode或MSVC工程（查看CMake文档获取更多信息）。

下面的命令编译一个Debug的Chipmunk:

```
cmake -D CMAKE_BUILD_TYPE=Debug .
make
```

如何没有-D CMAKE_BUILD_TYPE=Debug参数，将生成一个release版本。

为什么使用CMake？一个非常好心的人完成了这个脚本的最初版本，然后我发现CMake能非常方便的解决跨平台编译问题。我知道有些人非常讨厌安装一些胡乱的non-make编译系统来编译某些东西，但是CMake确实节省了我大量的时间和精力。

## Hello Chipmunk（World）

下面的Hello World示例项目中，创建一个模拟世界，模拟一个球掉落到一个静态线段上然后滚动出去，并打印球的坐标。

```
#include <stdio.h>
#include <chipmunk.h>


int main(void){
  // cpVect是2D矢量，cpv()为初始化矢量的简写形式
  cpVect gravity = cpv(0, -100);

  // 创建一个空白的物理世界
  cpSpace *space = cpSpaceNew();
  cpSpaceSetGravity(space, gravity);

  // 为地面创建一个静态线段形状
  // 我们稍微倾斜线段以便球可以滚下去
  // 我们将形状关联到space的默认静态刚体上，告诉Chipmunk该形状是不可移动的
  cpShape *ground = cpSegmentShapeNew(space->staticBody, cpv(-20, 5),
cpv(20, -5), 0);
  cpShapeSetFriction(ground, 1);
  cpSpaceAddShape(space, ground);

  // 现在让我们来构建一个球体落到线上并滚下去
  // 首先我们需要构建一个 cpBody 来容纳对象的物理属性
  // 包括对象的质量、位置、速度、角度等
  // 然后我们将碰撞形状关联到cpBody上以给它一个尺寸和形状

  cpFloat radius = 5;
  cpFloat mass = 1;
```

```
  // 转动惯量就像质量对于旋转一样
  // 使用 cpMomentFor*() 来近似计算它
  cpFloat moment = cpMomentForCircle(mass, 0, radius, cpvzero);

  // cpSpaceAdd*() 函数返回你添加的东西
  // 很便利在一行中创建并添加一个对象
  cpBody *ballBody = cpSpaceAddBody(space, cpBodyNew(mass, moment));
  cpBodySetPos(ballBody, cpv(0, 15));

  // 现在我们会球体创建碰撞形状
  // 你可以为同一个刚体创建多个碰撞形状
  // 它们将会附着关联到刚体上并移动更随
  cpShape *ballShape = cpSpaceAddShape(space, cpCircleShapeNew(ballBody,
radius, cpvzero));
  cpShapeSetFriction(ballShape, 0.7);

  // 现在一切都建立起来了，我们通过称作时间步的小幅度时间增量来步进模拟空间中的所有物体
  // *高度*推荐使用固定长的时间步
  cpFloat timeStep = 1.0/60.0;
  for(cpFloat time = 0; time < 2; time += timeStep){
    cpVect pos = cpBodyGetPos(ballBody);
    cpVect vel = cpBodyGetVel(ballBody);
    printf(
      "Time is %5.2f. ballBody is at (%5.2f, %5.2f). It's velocity is
(%5.2f, %5.2f)\n",
      time, pos.x, pos.y, vel.x, vel.y
    );

    cpSpaceStep(space, timeStep);
  }

  // 清理我们的对象并退出
  cpShapeFree(ballShape);
  cpBodyFree(ballBody);
  cpShapeFree(ground);
  cpSpaceFree(space);

  return 0;
```

## 支持

获得支持最好的方式就是访问Chipmunk论坛。上面有许多人使用Chipmunk，应用在我知道的各个平台上。如果你在做一个商业项目，Howling Moon Software（我的公司）可给与支持。我们可以帮助你实现自定义Chipmunk行为，以及bug修复和性能优化。

## 联系

如果你发现Chipmunk中的任何bug，错误或者该文档中坏掉的链接，又或者对于Chipmunk有任何疑问、评论，都可以通过 slembcke@gmail.com (email或者GTalk)联系我。

## 开源协议

Chipmunk基于MIT协议。



这项协议意味着对于商业项目你不必购买许可证或者支付任何费用就能使用Chipmunk。（虽然我们真

的很感谢捐赠）

### 链接

- Chipmunk论坛 – Chipmunk2D官方论坛
- Howling Moon Software – 我合办的软件公司（我们提供外包工作）
- Chipmunk2D Pro – Chipmunk的增强版本，我们为ARM或者多核平台做了一些特定的优化，如从图像或程序数据中进行自动几何操作，以及为Objective-C做了API封装。
- 游戏 – 使用Chimunk做的游戏清单。至少一小部分我们知道。

## Chipmunk2D 基础

### 概述

在Chimpmunk中有4种基本对象类型，分别是

- 刚体：一个刚体容纳着一个对象的物理属性（如质量、位置、角度、速度等）。默认情况下，它并不具有任何形状，直到你为它添加一个或者多个碰撞形状进去。如果你以前做过物理粒子，你会发现它们的不同之处是刚体可以旋转。在游戏中，通常刚体都是和一个精灵一一对应关联的。你应该构建你的游戏以便可以使用刚体的位置和角度来绘制你的精灵。
- 碰撞形状：因为形状与刚体相关联，所以你可以为一个刚体定义形状。为了定义一个复杂的形状，你可以给刚体绑定足够多的形状。形状包含着一个对象的表面属性如摩擦力、弹性等。
- 约束/关节：约束和关节被用来描述刚体之间是如何关联的
- 空间：空间是Chipmunk中模拟对象的容器。你将刚体、形状、关节添加进入一个空间，然后将空间作为一个整体进行更新。空间控制着所有的刚体、形状和约束之间的相互作用。

### 内存管理

对于你将使用的大多数结构体来说，Chipmunk采用了一套或多或少的标准和简单直接的内存管理方式。拿`cpSpace`结构体来举例：

- cpSpaceNew() – 分配并初始化一个`cpSpace`结构体。它先后调用了`cpSpaceAlloc()`和`cpSpaceInit(cpSpace *space)`
- cpSpaceFree(cpSpace *space) – 破坏并释放`cpSpace`结构体

你对任何你所分配过空间的结构体都负有释放的责任。 Chipmunk没有采用引用计数和垃圾回收机制。如果你调用了一个`new`函数，则必须匹配调用`free`函数来释放空间，否则会引起内存泄漏。

另外当你需要在栈上分配临时结构体，或者写一个语言绑定又或者在一个低内存容量的环境下编码，这时你需要更多分配和初始化的控制权，便可以使用下面的函数。*大部分人永远都不会使用下面几个函数。*

- cpSpaceAlloc() – 为一个`cpSpace`结构体分配空间，但不进行初始化。所有的分配空间的函数看

起来大致就像这样:`return (cpSpace *)cpcalloc(1, sizeof(cpSpace));`。如果需要的话你可以自己实现自己的分配空间函数
- cpSpaceInit(cpSpace *space) – 初始化`cpSpace`结构体
- cpSpaceDestroy(cpSpace *space) – 释放由`cpSpaceInit()`申请的所有内存空间，但并不释放`cpSpace`结构体本身

就像`new`和`free`函数的对应调用一样，任何由`alloc`函数分配的内存都要由`cpfree()`或类似的函数来释放，任何由`init`函数初始化申请空间的对象都要通过`destroy`函数来释放。

## 基本类型

`chipmunk_types.h`定义了Chipmunk使用的一些基本数据类型。这些数据类型可以在编译时改变以便适应你的需求:

- cpFloat: 浮点型，默认为double
- cpVect: 2D矢量，cpVect相关文档
- cpBool: 像每一个优秀的C语言库一样，具有跨语言兼容性，你可以定义自己的布尔类型，默认为int
- cpDataPointer: 指针类型，可以是回调、用户自定义数据的指针，默认是void*
- cpCollistionType: 碰撞形状类型的唯一标识符，默认是unsigned int。自定义类型必须支持==运算符
- cpGroup: 碰撞组唯一标识符，默认是unsigned int。当你不想区分组别的时候，可以定义一个`CP_NO_GROUP`。自定义类型必须支持==运算符
- cpLayers: 该类型被用作为层的掩码，默认是unsigned int。`CP_ALL_LAYERS`被用来定义为所有层位。自定义类型必须支持位操作&运算符

## 数学运算

首先，Chipmunk默认使用双精度浮点数进行数学计算。在大多数现代台式机处理器下这样很可能更快点，并意味着你可以不用过多担心浮点舍入引起的误差。在编译库的时候你可以修改Chipmunk使用的浮点类型。请查看`chipmunk_types.h`。

Chipmunk为一些常用的数学函数定义了别名以便你可以用Chimpmunk的浮点型来代表`float`或者`double`类型计算。在你的代码里，这或许不是一个很充分的理由，但在你使用了错误的`float/double`版本的数学函数而造成了2%的性能损失，请使用这些别名函数。

有一些函数或许你会发现非常有用:

- cpFloat cpfclamp(cpFloat f, cpFloat min, cpFloat max) – 截断f在min和max之间
- cpFloat cpflerp(cpFloat f1, cpFloat f2, cpFloat t) – 对f1和f2进行线性插值
- cpFloat cpflerpconst(cpFloat f1, cpFloat f2, cpFloat d) – 从f1到f2不超过d的线性插值

浮点数无穷大被定义为INFINITY,很多数学库中这样定义，但这实际上并不是C标准库的一部分。

## Chipmunk矢量：cpVect

### 结构体定义、常量和构造函数

定义：

```
typedef struct cpVect{
    cpFloat x, y;
} cpVect
```

零向量常量：

```
static const cpVect cpvzero = {0.0f,0.0f};
```

创建新结构体所用的便捷构造函数：

```
cpVect cpv(const cpFloat x, const cpFloat y)
```

### 操作运算

- cpBool cpveql(const cpVect v1, const cpVect v2) – 检测两个向量是否相等。在使用C++程序时，Chipmunk提供一个重载操作符==。（比较浮点数时要小心！）
- cpVect cpvadd(const cpVect v1, const cpVect v2) – 两个向量相加。在使用C++程序时，Chipmunk提供一个重载操作符+。
- cpVect cpvsub(const cpVect v1, const cpVect v2) – 两个向量相减。在使用C++程序时，Chipmunk提供一个重载操作符-。
- cpVect cpvneg(const cpVect v) – 使一个向量反向。在使用C++程序时，Chipmunk提供一个重载一个一元负操作符-。
- cpVect cpvmult(const cpVect v, const cpFloat s) – 标量乘法。在使用C++程序时，Chipmunk提供一个重载操作符*。
- cpFloat cpvdot(const cpVect v1, const cpVect v2) – 向量的点积。
- cpFloat cpvcross(const cpVect v1, const cpVect v2) – 2D向量交叉相乘的模。2D向量交叉相乘的积作为一个只有z坐标的3D向量的z值。函数返回z坐标的值。
- cpVect cpvperp(const cpVect v) – 返回一个垂直向量。（旋转90度）
- cpVect cpvrperp(const cpVect v) – 返回一个垂直向量。（旋转-90度）
- cpVect cpvproject(const cpVect v1, const cpVect v2) – 返回向量v1在向量v2上的投影。
- cpVect cpvrotate(const cpVect v1, const cpVect v2) – 使用复杂的乘法运算将向量v1按照向量v2旋转。如果v1不是单位向量，则v1会被缩放。
- cpVect cpvunrotate(const cpVect v1, const cpVect v2) – 和cpvrotate()相反。
- cpFloat cpvlength(const cpVect v) – 返回v的长度。

- cpFloat cpvlengthsq(const cpVect v) – 返回v的长度的平方，如果只是比较长度的话它的速度比cpvlength()快。
- cpVect cpvlerp(const cpVect v1, const cpVect v2, const cpFloat t) – 在v1和v2之间线性插值。
- cpVect cpvlerpconst(cpVect v1, cpVect v2, cpFloat d) – 以长度d在v1和v2之间线性插值。
- cpVect cpvslerp(const cpVect v1, const cpVect v2, const cpFloat t) – 在v1和v2之间球形线性插值。
- cpVect cpvslerpconst(const cpVect v1, const cpVect v2, const cpFloat a) – 在v1和v2之间以不超过角a的弧度值球形线性插值。
- cpVect cpvnormalize(const cpVect v) – 返回a的一个归一化副本。作为特殊例子，在调用cpvzero时返回cpvzero。
- cpVect cpvclamp(const cpVect v, const cpFloat len) – 将v固定到len上。
- cpFloat cpvdist(const cpVect v1, const cpVect v2) – 返回v1和v2间的距离。
- cpFloat cpvdistsq(const cpVect v1, const cpVect v2) – 返回v1和v2间的距离的平方。如果只是比较距离的话它比cpvdist()快。
- cpBool cpvnear(const cpVect v1, const cpVect v2, const cpFloat dist) – 如果v1和v2间的距离小于dist则返回真。
- cpVect cpvforangle(const cpFloat a) – 返回所给角（以弧度）单位向量。
- cpFloat cpvtoangle(const cpVect v) – 返回v所指的角度方向的弧度。

## Chipmunk轴对齐包围盒：cpBB

### 结构体定义和构造函数

- 简单的包围盒结构体，存储着left,bottom,right,top等值。

```
typedef struct cpBB{
    cpFloat l, b, r ,t;
} cpBB
```

- 便捷的构造函数，如cpv()函数一样返回一个副本而不是一个申请的指针。

```
cpBB cpBBNew(const cpFloat l, const cpFloat b, const cpFloat r, const
cpFloat t)
```

- 便捷的构造函数，用来构造一个位置为p，半径为r的一个圆的包围盒

```
cpBB cpBBNewForCircle(const cpVect p, const cpFloat r)
```

### 操作运算

- cpBool cpBBIntersects(const cpBB a, const cpBB b) – 如果边界框相交返回true
- cpBool cpBBContainsBB(const cpBB bb, const cpBB other) – 如果bb完全包含other返回true

- cpBool cpBBContainsVect(const cpBB bb, const cpVect v) – 如果bb包含v返回true
- cpBB cpBBMerge(const cpBB a, const cpBB b) – 返回包含a和b的最小的边界框
- cpBB cpBBExpand(const cpBB bb, const cpVect v) – 返回包含bb和v的最小的边界框
- cpVect cpBBCenter(const cpBB bb) – 返回bb的中心点矢量
- cpFloat cpBBArea(cpBB bb) – 返回bb矢量表示的边界框的面积
- cpFloat cpBBMergedArea(cpBB a, cpBB b) – 合并a和b然后返回合并后的矢量的边界框的面积
- cpFloat cpBBSegmentQuery(cpBB bb, cpVect a, cpVect b) – 返回分段查询相交bb的相交点个数，如果没有相交，返回INFINITY
- cpBool cpBBIntersectsSegment(cpBB bb, cpVect a, cpVect b) – 如果由a和b两端点定义的线段和bb相交返回true
- cpVect cpBBClampVect(const cpBB bb, const cpVect v) – 返回v在边界框中被截断的矢量的副本
- cpVect cpBBWrapVect(const cpBB bb, const cpVect v) – 返回v包含边界框的矢量的副本

## Chipmunk刚体：cpBody

### 流氓和静态刚体

一般当我们创建一个刚体并将它添加到空间上后，空间就开始对之进行模拟，包括了对刚体位置、速度、受力以及重力影响等的模拟。没被添加到空间（没有被模拟）的刚体我们把它称之为*流氓刚体*。流氓刚体最重要的用途就是用来当作静态刚体，但是你仍然可以使用它们来实现直接控制物体，如移动平台。

静态刚体是流氓刚体，但被设置了一个特殊的标志以便让Chipmunk知道它们从不移动除非你要求这么做。静态刚体有两个目的。最初，它们被加入用来实现休眠功能。因为静态刚体不移动，Chipmunk知道让与它们接触或者连接的物体安全的进入休眠。接触或连接常规流氓刚体的物体从不允许休眠。静态刚体的第二个目的就是让Chipmunk知道关联到它们的形状从不需要更新它们的碰撞检测数据。Chipmunk也不需要关心静态物体之间的碰撞。一般所有水平几何都会被关联到一个静态刚体上除了移动平台或门等物体。

在Chipmunk5.3版本之前，你要创建一个无限大质量的流氓刚体通过cpSpaceAddStaticShape()来添加静态形状。现在你不必这样做了，并且如果你想使用休眠功能也不应该这样做了。每一个空间都有一个专用的静态刚体，你可以使用它来添加静态形状。Chipmunk也会自动将形状作为静态形状添加到静态刚体上。

### 内存管理函数

```
cpBody *cpBodyAlloc(void)
cpBody *cpBodyInit(cpBody *body, cpFloat m, cpFloat i)
cpBody *cpBodyNew(cpFloat m, cpFloat i)

void cpBodyDestroy(cpBody *body)
```

```
void cpBodyFree(cpBody *body)
```

如上是一套标准的Chipmunk内存管理函数。m和i是刚体的质量和转动惯量。猜想刚体的质量通常是好的，但是猜想刚体的转动惯量却会导致一个很差的模拟。在任何关联到刚体的形状或者约束从空间移除之前注意不要释放刚体。

### 创建额外静态刚体

每一个cpSpace都有一个可以直接使用的内置的静态刚体，构建你自己的也非常便利。一个潜在的用途就是用在关卡编辑器中。通过将关卡的物块关联到静态刚体，你仍然可以相互独立的移动和旋转物块。然后你要做的就是在结束后调用cpSpaceRehashStatic()来重建静态碰撞检测的数据。

关于流氓和静态刚体的更多信息，请看Chipmunk空间。

```
cpBody *cpBodyAlloc(void);
cpBody *cpBodyInitStatic(cpBody *body)
cpBody *cpBodyNewStatic()
```

创建额外的具有无限的质量和转动惯量的静态物体。

### 属性

Chipmunk为刚体的多个属性提供了getter/setter函数。如果刚体在休眠状态，设置大多数属性会自动唤醒它们。如果你想，你也可以直接在cpBody结构体内设置字段。它们都在头文件中有记录。

```
cpFloat cpBodyGetMass(const cpBody *body)
void cpBodySetMass(cpBody *body, cpFloat m)
```

刚体的质量。

```
cpFloat cpBodyGetMoment(const cpBody *body)
void cpBodySetMoment(cpBody *body, cpFloat i)
```

刚体的转动惯量（MoI（译者注：Moment Of Inertia即转动惯量的缩写）或有时只说惯量）。惯量就像刚体的旋转质量。请查阅下面的函数来帮助计算惯量。

```
cpVect cpBodyGetPos(const cpBody *body)
void cpBodySetPos(cpBody *body, cpVect pos)
```

刚体重心的位置。当改变位置的时候如果你要计划对空间进行任何查询，你可能还需要调用cpSpaceReindexShapesForBody()来更新关联形状的碰撞检测信息。

```
cpVect cpBodyGetVel(const cpBody *body)
void cpBodySetVel(cpBody *body, const cpVect value)
```

刚体重心的线速度。

```
cpVect cpBodyGetForce(const cpBody *body)
void cpBodySetForce(cpBody *body, const cpVect value)
```

施加到刚体重心的力。

```
cpFloat cpBodyGetAngle(const cpBody *body)
void cpBodySetAngle(cpBody *body, cpFloat a)
```

刚体的角度，弧度制。当改变角度的时候如果你要计划对空间进行任何查询，你可能还需要调用cpSpaceReindexShapesForBody()来更新关联形状的碰撞检测信息。

```
cpFloat cpBodyGetAngVel(const cpBody *body)
void cpBodySetAngVel(cpBody *body, const cpFloat value)
```

刚体的角速度，弧度/秒，

```
cpFloat cpBodyGetTorque(const cpBody *body)
void cpBodySetTorque(cpBody *body, const cpFloat value)
```

施加到刚体的扭矩。

```
cpVect cpBodyGetRot(const cpBody *body)
```

刚体的旋转向量。可通过cpvrotate()或者cpvunrotate()进行快速旋转。

```
cpFloat cpBodyGetVelLimit(const cpBody *body)
void cpBodySetVelLimit(cpBody *body, const cpFloat value)
```

刚体的速度极限。、默认为INFINITY（无限大），除非你专门设置它。可以被用来限制下落速度等。

```
cpFloat cpBodyGetAngVelLimit(const cpBody *body)
void cpBodySetAngVelLimit(cpBody *body, const cpFloat value)
```

刚体以弧度/秒的角速度限制。默认为INFINITY，除非你专门设置它。

```
cpSpace* cpBodyGetSpace(const cpBody *body)
```

获取body所添加进去的cpSpace。

```
cpDataPointer cpBodyGetUserData(const cpBody *body)
void cpBodySetUserData(cpBody *body, const cpDataPointer value)
```

使用数据指针。使用该指针从回调中获取拥有该刚体的游戏对象的引用。

## 转动惯量和面积帮助函数

使用以下函数来近似计算出刚体的转动惯量，如果想得到多个，那就将结果相加在一起。

- cpFloat cpMomentForCircle(cpFloat m, cpFloat r1, cpFloat r2, cpVect offset) – 计算空心圆的转动惯性，r1和r2是在任何特定顺序下的内径和外径。（实心圆圈的内径为0）
- cpFloat cpMomentForSegment(cpFloat m, cpVect a, cpVect b) – 计算线段的转动惯量。端点a和b相对于刚体。
- cpFloat cpMomentForPoly(cpFloat m, int numVerts, const cpVect *verts, cpVect offset) – 计算固定多边形的转动惯量，假设它的中心在质心上。offset偏移值被加到每个顶点。
- cpFloat cpMomentForBox(cpFloat m, cpFloat width, cpFloat height) – 计算居中于刚体的实心矩形的转动惯量。

转动惯量例子

```
// 质量为2，半径为5的实心圆的转动惯量
cpFloat circle1 = cpMomentForCircle(2, 0, 5, cpvzero);

// 质量为1，内径为1，外径为6的空心圆的转动惯量
cpFloat circle2 = cpMomentForCircle(1, 2, 6, cpvzero);

// 质量为1，半径为3，x轴方向偏离重心量为3的实心圆的转动惯量
cpFloat circle3 = cpMomentForCircle(2, 0, 5, cpv(3, 0));

// 复合对象。居中于重心的1x4的矩形和y轴偏移重心量为3，半径为1的实心圆
// 只需将转动惯量相加到一起
cpFloat composite = cpMomentForBox(boxMass, 1, 4) +
cpMomentForCircle(circleMass, 0, 1, cpv(0, 3));
```

如果你想近似计算诸如质量或密度此类的东西，可以使用下列函数来获取Chipmunk形状区域。

- cpFloat cpAreaForCircle(cpFloat r1, cpFloat r2) – 空心圆形状面积
- cpFloat cpAreaForSegment(cpVect a, cpVect b, cpFloat r) – 斜线段面积。（如果半径为0的话永远为0）
- cpFloat cpAreaForPoly(const int numVerts, const cpVect *verts) – 多边形形状的面积。多边形为凹多边形时返回一个负值。

## 坐标系转换函数

许多事情被定义在刚体的局部坐标，也就意味着（0,0）是刚体的重心和轴线旋转中心。

- cpVect cpBodyLocal2World(const cpBody *body, const cpVect v) – 从刚体局部坐标系转换到世界

坐标系

- cpVect cpBodyWorld2Local(const cpBody *body, const cpVect v) – 从世界坐标系转换到刚体的局部坐标系

## 施加力和力矩

人们有时候容易混淆力和冲力之间的区别。冲力基本上是一个在非常短的时间内施加的一个非常大的力，就像一个球击中一堵墙或者大炮射击一样。Chipmunk的冲力会在一瞬间直接施加在物体的速度上。无论是力还是冲力都受到物体质量的影响。物体质量翻倍，则效果减半。

- void cpBodyResetForces(cpBody *body) – 对刚体施加0值的力和扭矩
- void cpBodyApplyForce(cpBody *body, const cpVect f, const cpVect r) – 在离重心相对偏移量为r的位置施加`f`的力于`body`上
- void cpBodyApplyImpulse(cpBody *body, const cpVect j, const cpVect r) – 在离重心相对偏移量为r的位置施加`j`的冲力于`body`上。

注: `cpBodyApplyForce()`和`cpBodyApplyImpulse()`两者都是在绝对坐标系中施加力或者冲力，并在绝对坐标系中产生相对的偏移。（偏移量相对于重心位置，但不随刚体旋转）

## 休眠函数

Chipmunk支持休眠功能，以便其停止使用CPU时间来模拟移动的对象组。更多信息请查阅cpSpace部分。

- cpBool cpBodyIsSleeping(const cpBody *body) – 如果刚体在休眠则返回`true`。
- void cpBodyActivate(cpBody *body) – 重设刚体的闲置时间。如果在休眠，则会唤醒它以及和它接触的任何其他刚体。
- void cpBodySleep(cpBody *body) – 强制一个刚体立即进入休眠，即使它在半空中。不能从回调中被调用。
- void cpBodyActivateStatic(cpBody *body, cpShape* filter) – 和`cpBodyActivate()`功能类似。激活刚体接触的所有刚体。如果`filter`不为`NULL`，那么只有通过筛选过滤的刚体才会被唤醒。

```
void cpBodySleepWithGroup(cpBody *body, cpBody *group)
```

当对象在Chipmunk中处于休眠时，和它接触或连接在一起的所有刚体都会作为一组进入休眠。当对象被唤醒时，和它一组的所有对象都会被唤醒。

`cpBodySleepWithGroup()`允许你将群组中的对象一起休眠。如果你通过一个新的组给`groups`传递`NULL`值，则它和`cpBodySleep()`功能一样。如果你为`groups`传入一个休眠的刚体，那么当`group`是唤醒状态时，`body`也会被唤醒。你可以通过这来初始化关卡并开始堆对象的预休眠状态。

休眠例子

```
// 构建一堆箱子
// 强制它们进入休眠直到他们第一次被接触
// 将它们放进一组以便接触它们任意一个都会唤醒他们
cpFloat size = 20;
cpFloat mass = 1;
cpFloat moment = cpMomentForBox(mass, size, size);

cpBody *lastBody = NULL;

for(int i=0; i&lt;5; i++){
  cpBody *body = cpSpaceAddBody(space, cpBodyNew(mass, moment));
  cpBodySetPos(body, cpv(0, i*size));

  cpShape *shape = cpSpaceAddShape(space, cpBoxShapeNew(body, size, size));
  cpShapeSetFriction(shape, 0.7);

  // 你可以使用任意休眠刚体作为组别的标识符
  // 这里我们只保存了我们初始化的最后一个刚体的引用
  // 传入NULL值作为组别将启动一个新的休眠组
  // 你必须在完全初始化对象后这么做
  // 添加形状或调用setter函数将会唤醒刚体
  cpBodySleepWithGroup(body, lastBody);
  lastBody = body;
}
```

迭代器

```
typedef void (*cpBodyShapeIteratorFunc)(cpBody *body, cpShape *shape, void
*data)
void cpBodyEachShape(cpBody *body, cpBodyShapeIteratorFunc func, void
*data)
```

对于关联到body且被加入到空间的每个形状调用func函数。data作为上下文值传递。使用这些回调来删除形状是安全的。

```
typedef void (*cpBodyConstraintIteratorFunc)(cpBody *body, cpConstraint
*constraint, void *data)
void cpBodyEachConstraint(cpBody *body, cpBodyConstraintIteratorFunc func,
void *data)
```

对于关联到body且被加入到空间的每个约束调用func函数。data作为上下文值传递。使用这些回调来删除约束是安全的。

```
typedef void (*cpBodyArbiterIteratorFunc)(cpBody *body, cpArbiter *arbiter,
void *data)
void cpBodyEachArbiter(cpBody *body, cpBodyArbiterIteratorFunc func, void
*data)
```

这个更有趣。对于刚体参与碰撞的每个碰撞对调用func函数。调用cpArbiterGet[Bodies|Shapes]()或者CP_ARBITER_GET_[BODIES|SHAPES]()将会返回刚体或者形状作为第一个参数。你可以用它来检查各种碰撞信息。比如，接触地面，接触另一特定的对象，施加到对象上的碰撞力等。被碰撞处理回调或者cpArbiterIngnore()的传感器形状和仲裁者将不被接触图形跟踪。

注：如果你的编译器支持闭包（如Clang），还有另外一组函数可以调用，如cpBodyEachShape_b()等。更多信息见chipmunk.h。

Crushing例子

```
struct CrushingContext {
  cpFloat magnitudeSum;
  cpVect vectorSum;
};

static void
EstimateCrushingHelper(cpBody *body, cpArbiter *arb, struct CrushingContext
*context)
{
  cpVect j = cpArbiterTotalImpulseWithFriction(arb);
  context->magnitudeSum += cpvlength(j);
  context->vectorSum = cpvadd(context->vectorSum, j);
}

cpFloat
EstimateCrushForce(cpBody *body, cpFloat dt)
{
  struct CrushingContext crush = {0.0f, cpvzero};
  cpBodyEachArbiter(body,
(cpBodyArbiterIteratorFunc)EstimateCrushingHelper, &amp;crush);
```

```
  // 通过比较向量和以及幅度和来查看碰撞的力量彼此相对有多大
  cpFloat crushForce = (crush.magnitudeSum -
cpvlength(crush.vectorSum))*dt;
}
```

### 嵌入回调

这部分是残留。现在你可以看看星球演示这个例子，看如何使用嵌入回调来实现的行星重力。

### 杂项函数

- `cpBool cpBodyIsStatic(const cpBody *body)` – 如果body是静态刚体的话，返回`true`。无论是`cpSpace.staticBody`，还是由`cpBodyNewStatic()`或者`cpBodyInitStatic()`创建的刚体。
- `cpBool cpBodyIsRogue(const cpBody *body)`- 如果刚体从来没有被加入到空间的话返回`true`。

### 札记

- 如果可能的话使用力来修正刚体。这样是最稳定的。
- 修正刚体的速度是不可避免的，但是在每帧对刚体的速度做巨大的变化会造成一些奇怪的模拟。你可以自由实验，但别说我没警告你哦。
- 不要在单步中修正刚体的位置除非你确实知道你在干什么。否则你得到的位置、速度则会不同步。
- 如果在调用`cpSpaceRemoveShape()`之前你就要释放一个刚体，那么会引起崩溃。

## Chipmunk碰撞形状：cpShape

当前有三种类型的碰撞形状：

1. 圆形：快速简单的碰撞形状
2. 线段：主要作为静态形状。可以倾斜以便给之一个厚度。
3. 凸多边形：最慢，但却为最灵活的碰撞形状。

如果你愿意，你可以在一个刚体上添加任意数量的形状。这就是为什么两种类型（形状和刚体）是分离开的。这将会让你足够灵活的来给相同对象的不同区域提供不同的摩擦力、弹性以及回调值。

当创建不同类型的形状的时候，你将永远得到一个`cpShape*`指针返回。这是因为Chipmunk的形状是不透明的类型。想象具体的碰撞形状类型如`cpCircleShape`, `cpSegmentShape`和`cpPolyShape`, 他们都是`cpShape`的私有子类。你仍然可以使用`getter`函数来获取他们的属性，但不要将`cpShape`指针转成他们的特定类型指针。

### 札记

Chipmunk直到 6.1.2 版本才支持线段、线段碰撞。由于兼容性的原因，你必须明确地全局调用 `cpEnableSegmentToSegmentCollisions（）`来启用它们。（感谢LegoCylon对此的帮助）

### 属性

Chipmunk为一些碰撞形状属性提供了`getter/ setter`函数。如果形状关联的刚体在休眠，设置多数属性都会自动唤醒它们。如果你想的话，也可以直接设置`cpShape`结构的某些字段。他们在头文件中都记录有。

```
cpBody * cpShapeGetBody(const cpShape *shape)
void cpShapeSetBody(cpShape *shape, cpBody *body)
```

只有当形状尚未添加进空间的时候才能关联到一个刚体。

```
cpBB cpShapeGetBB(const cpShape *shape)
```

上面得到的是形状的碰撞包围盒。只能保证在`cpShapeCacheBB()`或`cpSpaceStep()`调用后是有效的。移动形状所连接到刚体并不更新它的包围盒。对于没有关联到刚体的用于查询的形状，也可以使用`cpShapeUpdate()`。

```
cpBool cpShapeGetSensor(const cpShape *shape)
void cpShapeSetSensor(cpShape *shape, cpBool value)
```

用来标识形状是否是一个感应器的布尔值。感应器只调用碰撞回调，但却不产生真实的碰撞。

```
cpFloat cpShapeGetElasticity(const cpShape *shape)
void cpShapeSetElasticity(cpShape *shape, cpFloat value)
```

上面说的是形状的弹性。值0.0表示没有反弹，而值为1.0将提供一个“完美”的反弹。然而由于使用1.0或更高的值会导致模拟不精确，所以不推荐。碰撞的弹性是由单个形状的弹性相乘得到。

```
cpFloat cpShapeGetFriction(const cpShape *shape)
void cpShapeSetFriction(cpShape *shape, cpFloat value)
```

上面说的是摩擦系数。Chipmunk使用的是库仑摩擦力模型，0.0值表示无摩擦。碰撞间的摩擦是由单个形状的摩擦相乘找到。摩擦系数表

```
cpVect cpShapeGetSurfaceVelocity(const cpShape *shape)
void cpShapeSetSurfaceVelocity(cpShape *shape, cpVect value)
```

上面说的是物体的表面速度。可用于创建传送带或走动的玩家。此值在计算摩擦时才会使用，而不是用于解决碰撞。

```
cpCollisionType cpShapeGetCollisionType(const cpShape *shape)
```

```
void cpShapeSetCollisionType(cpShape *shape, cpCollisionType value)
```

您可以为Chipmunk的碰撞形状指定类型从而在接触特定类型物体的时候触发回调。更多信息请参见回调部分。

```
cpGroup cpShapeGetGroup(const cpShape *shape)
void cpShapeSetGroup(cpShape *shape, cpGroup value)
```

在相同的非零组中，形状间不产生碰撞。在创建了一个许多形状组成的物体，但却不想自身与自身之间发生碰撞，这会很有用。默认值为`CP_NO_GROUP`。

```
cpLayers cpShapeGetLayers(const cpShape *shape)
void cpShapeSetLayers(cpShape *shape, cpLayers value)
```

只有在相同的位平面内形状间才发生碰撞。比如`(a->layers & b->layers) != 0`。默认情况下，一个形状占据所有的位平面。如果你不熟悉如何使用它们，维基百科有篇很好的文章介绍了位掩码的相关知识你可以阅读下。默认值为CP_ALL_LAYERS。

```
cpSpace* cpShapeGetSpace(const cpShape *shape)
```

得到形状所添加进去的空间。

```
cpDataPointer cpShapeGetUserData(const cpShape *shape)
void cpShapeSetUserData(cpShape *shape, cpDataPointer value)
```

上面说的是用户定义的数据指针。如果你设置将其指向形状关联的游戏对象，那么你可以从Chipmunk回调中访问你的的游戏对象。

### 碰撞过滤

Chipmunk 有两种主要的途径来忽略碰撞: 群组和层。

群组是为了忽略一个复杂对象部分之间的碰撞。玩偶是一个很好的例子。当联合手臂和躯干的时候，他们可以重叠。群组允许这样做。相同群组间的形状不产生碰撞。所以通过将一个布娃娃的所有形状放在同一群组中，就会阻止其碰撞自身的其它部件。

层允许你将碰撞的形状分离在相互排斥的位面。形状不止可以在一个层上，形状与形状发生碰撞，而两者必须至少在一个相同的层上。举一个简单的例子，比如说形状A是在第1层，形状B是在第2层和形状C是在层1和2上。形状A和B不会互相碰撞，但形状C将与这两个A和B发生碰撞

层也可以用于建立基于碰撞的规则。比如说在你的游戏中有四种类型的形状。玩家，敌人，玩家子弹，敌人子弹。玩家应该和敌人发生碰撞，但子弹却不应该和发射者碰撞。图表类似下图:

| | Player | Enemy | Player Bullet | Enemy Bullet |
|---|---|---|---|---|
| Player | – | (1) | | (2) |
| Enemy | – | – | (3) | |
| Player Bullet | – | – | – | |
| Enemy Bullet | – | – | – | – |

图表中'-'是多余的斑点，数字的地方应该发生碰撞。您可以使用层要定义每个规则。然后将层添加到每个类型：玩家应该在层1和2，敌人应该是在层1和3中，玩家的子弹应该是在层3中，以及敌人的子弹应该是在层2中。这种处理层作为为规则的方式，可以定义多达32个规则。默认cpLayers类型为unsigned int其中在大多数系统是32位的。如果你需要更多的比特来完成工作，你可以在chipmunk_types.h中重新定义cpLayers类型。

还有最后一个方法通过碰撞处理函数来过滤碰撞。见回调的部分来获取更多信息。碰撞处理程序可以更灵活，但它们也是最慢的方法。所以，你要优先尝试使用群组或层。

### 内存管理函数

```
void cpShapeDestroy(cpShape *shape)
void cpShapeFree(cpShape *shape)
```

Destroy和Free函数由所有形状类型共享。分配和初始化函数特定于每一个形状。见下文。

### 其他函数

- cpBB cpShapeCacheBB(cpShape *shape) – 同步形状与形状关联的刚体
- cpBB cpShapeUpdate(cpShape *shape, cpVect pos, cpVect rot) – 设置形状的位置和旋转角度
- void cpResetShapeIdCounter(void) – Chipmunk使用了一个计数器，以便每一个新的形状是在空间索引中使用唯一的哈希值。因为这会影响空间中碰撞被发现和处理的顺序，你可以在每次空间中添加新的形状时重置形状计数器。如果你不这样做，有可能模拟（非常）略有不同。

### 圆形形状

```
cpCircleShape *cpCircleShapeAlloc(void)
cpCircleShape *cpCircleShapeInit(cpCircleShape *circle, cpBody *body,
cpFloat radius, cpVect offset)
```

```
cpShape *cpCircleShapeNew(cpBody *body, cpFloat radius, cpVect offset)
```

body 是圆形形状关联的刚体。offset 是在刚体局部坐标系内，与刚体中心的偏移量。

```
cpVect cpCircleShapeGetOffset(cpShape *circleShape)
cpFloat cpCircleShapeGetRadius(cpShape *circleShape)
```

圆形形状属性的getter函数。传一个非圆形形状将会抛出一个异常。

### 线段形状

```
cpSegmentShape* cpSegmentShapeAlloc(void)
cpSegmentShape* cpSegmentShapeInit(cpSegmentShape *seg, cpBody *body,
cpVect a, cpVect b, cpFloat radius)
cpShape* cpSegmentShapeNew(cpBody *body, cpVect a, cpVect b, cpFloat
radius)
```

body是线段形状关联的刚体，a和b是端点，radius是线段的厚度。

```
cpVect cpSegmentShapeGetA(cpShape *shape)
cpVect cpSegmentShapeGetA(cpShape *shape)
cpVect cpSegmentShapeGetNormal(cpShape *shape)
cpFloat cpSegmentShapeGetRadius(cpShape *shape)
```

线段属性的getter函数。传入一个非线段形状会抛出一个断言。

```
void cpSegmentShapeSetNeighbors(cpShape *shape, cpVect prev, cpVect next)
```

当你有一些连接在一起的线段形状时，线段仍然可以与线段间的“裂缝”碰撞。通过设置相邻线段的端点，你告诉Chipmunk来避免裂缝内部碰撞。

### 多边形形状

```
cpPolyShape *cpPolyShapeAlloc(void)
cpPolyShape *cpPolyShapeInit(cpPolyShape *poly, cpBody *body, int numVerts,
const cpVect *verts, cpVect offset)
cpShape *cpPolyShapeNew(cpBody *body, int numVerts, const cpVect *verts,
cpVect offset)
```

body是多边形关联的刚体，verts是一个cpVect结构体数组，定义了顺时针方向的凸多边形顶点，offset是在刚体局部坐标系中与刚体重心的偏移量。当顶点没形成凸多边形或者不是顺时针顺序的时候会抛出一个断言。

```
cpPolyShape *cpPolyShapeInit2(cpPolyShape *poly, cpBody *body, int
```

```
numVerts, const cpVect *verts, cpVect offset, cpFloat radius)
cpShape *cpPolyShapeNew2(cpBody *body, int numVerts, cpVect *verts, cpVect
offset, cpFloat radius)
```

和上面的一样，但允许你创建一个带有半径的多边形形状。（我知道名字有点糊涂，在Chipmunk7中将会清理）

```
int cpPolyShapeGetNumVerts(cpShape *shape)
cpVect cpPolyShapeGetVert(cpShape *shape, int index)
cpFloat cpPolyShapeGetRadius()
```

多边形形状属性的getter函数。传递一个非多边形形状或者不存在的index将会抛出一个断言。

### 修改cpShpaes

简短的回答是，你不能因为这些更改将只拿起一个改变形状的面的位置，而不是它的速度。长的答案是，你可以使用“不安全”的API，只要你认识到这样做不会导致真实的物理行为。这些额外的功能都在单独的头文件chipmunk_unsafe.h中定义。

### 札记

- 你可以将多个碰撞形状关联到刚体上。这样你就可以创建几乎任何形状。
- 关联在同一个刚体上的形状不会产生冲突。你不必担心同个刚体上的形状的重叠问题。
- 确保刚体和刚体的碰撞形状都被添加进了空间。有个例外，就是当你如果有一个外部刚体或你嵌入自身到刚体。在这种情况下，只需把形状添加进空间。

## Chipmunk空间：cpSpace

Chipmunk的空间是模拟的基本单元。你将刚体、形状和约束添加进去然后通过时间来步进更新模拟。

### 什么是迭代？为什么我要关心？

Chipmunk使用一个迭代求解器来计算出空间刚体之间的力。也就是说它建立了刚体间的所有碰撞、关节和约束的一个列表，并在列表中逐个考虑每一个刚体的若干条件。遍数这些条件便得到迭代次数，且每次迭代会使求解更准确。如果你使用太多的迭代，物理效果看起来应该不错并且坚实稳定，但可能消耗太多的CPU时间。如果你使用过少的迭代，模拟仿真似乎看起来有些糊状或弹性，而物体应该是坚硬的。设置迭代次数可以让你在CPU使用率和物理精度上做出平衡。 Chipmunk中默认的迭代值是10，足以满足大多数简单的游戏。

### 休眠

休眠是Chipmunk5.3新功能，是指空间停用已停止移动的整个对象群组，以节省CPU时间和电池寿命的能力。为了使用此功能，你必须做两件事情。第一个是，你必须将你的所有静态几何关联到静态刚体。

如果对象接触的是非静态流氓体，则它们不能进入休眠，即使它的形状是作为静态形状添加的。第二个是，你必须通过cpSpace.sleepTimeThreshold设置一个时间阈值来显式启用休眠。如果你没有明确设置cpSpace.idleSpeedThreshold，那么Chipmunk会基于当前重力自动产生一个休眠阈值。

### 属性

```
int cpSpaceGetIterations(const cpSpace *space)
void cpSpaceSetIterations(cpSpace *space, int value)
```

迭代次数允许你控制求解器计算的精度。默认值为10。更多信息见上面。

```
cpVect cpSpaceGetGravity(const cpSpace *space)
void cpSpaceSetGravity(cpSpace *space, cpVect value)
```

施加到空间的全局重力。默认是cpvzero。可以通过编写自定义积分函数来重写每个刚体。

```
cpFloat cpSpaceGetDamping(const cpSpace *space)
void cpSpaceSetDamping(cpSpace *space, cpFloat value)
```

施加到空间的简单的阻尼值。数值0.9意味着每个刚体每秒会损失速度会损失掉10%。默认值为1。像重力一样，阻尼值也可以在每个刚体上重写。

```
cpFloat cpSpaceGetIdleSpeedThreshold(const cpSpace *space)
void cpSpaceSetIdleSpeedThreshold(cpSpace *space, cpFloat value)
```

刚体被考虑为静止限制的速度阈值。默认值为0，意味着让空间来估算猜测基于重力的良好的阈值。

```
cpFloat cpSpaceGetSleepTimeThreshold(const cpSpace *space)
void cpSpaceSetSleepTimeThreshold(cpSpace *space, cpFloat value)
```

一组刚体休眠需要保持静止闲置的时间阈值。默认值为INFINITY, 禁用了休眠功能。

```
cpFloat cpSpaceGetCollisionSlop(const cpSpace *space)
void cpSpaceSetCollisionSlop(cpSpace *space, cpFloat value)
```

支持形状间的重叠量。鼓励将这个值设置高点而不必在意重叠，因为它提高了稳定性。它默认值为0.1。

```
cpFloat cpSpaceGetCollisionBias(const cpSpace *space)
void cpSpaceSetCollisionBias(cpSpace *space, cpFloat value)
```

Chipmunk让快速移动的物体重叠，然后修复重叠。即使横扫碰撞被支持，重叠对象也不可避免，并且这是一个高效，稳定的方式来处理重叠的对象。控制重叠百分比的偏置值在1秒后仍然是不固定的，默认~0.2%。有效值是在0到1的范围内，但由于稳定的原因不推荐使用0。默认值的计算公式为cpfpow（1.0F - 0.1F，60.0f），这意味着Chipmunk试图在1/60s内纠正10%的错误。注：非常非

常少的游戏需要更改此值。

```
cpTimestamp cpSpaceGetCollisionPersistence(const cpSpace *space)
void cpSpaceSetCollisionPersistence(cpSpace *space, cpTimestamp value)
```

空间保持碰撞的帧数量。有助于防止抖动接触恶化。默认值为3，非常非常非常少的游戏需要更改此值。

```
cpFloat cpSpaceGetCurrentTimeStep(const cpSpace *space)
```

检索当前(如果你是从cpSpaceStep()回调)或最近(在cpSpaceStep()之外调用)的时间步长。

```
cpFloat cpSpaceIsLocked(const cpSpace *space)
```

在回调中返回true时，意味着你不能从空间添加/删除对象。可以选择创建一个post-step回调来替代。

```
cpDataPointer cpSpaceGetUserData(const cpSpace *space)
void cpSpaceSetUserData(cpSpace *space, cpDataPointer value)
```

用户定义的数据指针。这点在游戏状态对象或拥有空间的场景管理对象上是很有用的。

```
cpBody * cpSpaceGetStaticBody(const cpSpace *space)
```

空间中专用的静态刚体。你不必使用它，而是因为它的内存由空间自动管理，非常方便。如果你想要做回调的话，你可以将它的数据指针指向一些有用的东西。

### 内存管理函数

```
cpSpace* cpSpaceAlloc(void)
cpSpace* cpSpaceInit(cpSpace *space)
cpSpace* cpSpaceNew()


void cpSpaceDestroy(cpSpace *space)
void cpSpaceFree(cpSpace *space)
```

更多标准的Chipmunk内存函数。

```
void cpSpaceFreeChildren(cpSpace *space)
```

这个函数将释放所有已添加到空间中的的形状、刚体和关节。不要释放space空间。你仍然需要自己调用cpSpaceFree()。在一个真正的游戏中你可能永远不会使用这个，因为你的游戏状态或者游戏控制器应该会管理从空间移除并释放对象。

### 操作运算

```
cpShape *cpSpaceAddShape(cpSpace *space, cpShape *shape)
cpShape *cpSpaceAddStaticShape(cpSpace *space, cpShape *shape)
cpBody *cpSpaceAddBody(cpSpace *space, cpBody *body)
cpConstraint *cpSpaceAddConstraint(cpSpace *space, cpConstraint
*constraint)

void cpSpaceRemoveShape(cpSpace *space, cpShape *shape)
void cpSpaceRemoveBody(cpSpace *space, cpBody *body)
void cpSpaceRemoveConstraint(cpSpace *space, cpConstraint *constraint)

cpBool cpSpaceContainsShape(cpSpace *space, cpShape *shape)
cpBool cpSpaceContainsBody(cpSpace *space, cpBody *body)
cpBool cpSpaceContainsConstraint(cpSpace *space, cpConstraint *constraint)
```

这些函数是从空间中添加和删除形状、刚体和约束。添加/删除函数不能在`postStep()`回调之外的回调内调用(这和postSolve()回调是不同的!)。当`cpSpaceStep()`仍然在执行时，试图从空间添加或删除对象会抛出一个断言。更多信息请参见回调部分。添加函数会返回被添加的对象以便你可以在一行中创建和添加一些东西。注意在移除关联到刚体的形状和约束之前不要去释放刚体，否则会造成崩溃。`contains`函数允许你检查一个对象有没有被添加到空间中。

### 静态动态转换函数

```
void cpSpaceConvertBodyToStatic(cpSpace *space, cpBody *body)
```

将刚体转换为静态刚体。它的质量和力矩将被设置为无穷大，并且速度为0。旧的质量和力矩以及速度都不会被保存。这将有效地将一个刚体和它的形状冻结到一个位置。这不能被一个激活的刚体调用，所以你可能需要先调用`cpSpaceRemoveBody()`。此外，因为它修改了碰撞检测的数据结构，如果你想从另外一个回调函数或迭代器使用你必须使用后一步的回调。

### 空间索引

Chipmunk6正式支持2个空间索引。默认是轴对齐包围盒树，该灵感来自于Bullet物理库中使用的包围盒树，但是我将它与我自己的碰撞对缓存一起做了扩展以便为树实现非常好的时间相干性。树无需调整优化，而且在大多数游戏中会发现使用它能获得更好的性能。另外一个可用的索引是空间哈希，当你有着非常多数量且相同尺寸的物体时，它会更快。

有时，你可能需要更新形状的碰撞检测数据。如果你移动静态形状或者刚体，你必须这样做来让Chipmunk知道它需要更新碰撞数据。你可能还希望手动为移动过的普通形状更新碰撞数据，并且仍然想进行查询。

- void cpSpaceReindexShape(cpSpace *space, cpShape* shape) – 重新索引一个指定的形状
- void cpSpaceReindexShapesForBody(cpSpace *space, cpBody* body) – 重新索引指定刚体上的所

有形状

- void cpSpaceReindexStatic(cpSpace *space) – 重新索引所有静态形状。一般只更新改变的形状会比较快

迭代器

```
typedef void (*cpSpaceBodyIteratorFunc)(cpBody *body, void *data)
void cpSpaceEachBody(cpSpace *space, cpSpaceBodyIteratorFunc func, void
*data)
```

为空间中的每个刚体调用func函数，同时传递data指针。休眠中的刚体包括在内，但是静态和流氓刚体不包括在内，因为他们没有被添加进空间。

cpSpaceEachBody例子:

```
// 检测空间中是否所有刚体都在休眠的代码片段

// 这个函数被空间中的每个刚体调用
static void EachBody(cpBody *body, cpBool *allSleeping){
  if(!cpBodyIsSleeping(body)) *allSleeping = cpFalse;
}

// 然后在你的更新函数中这样做
cpBool allSleeping = true;
cpSpaceEachBody(space, (cpSpaceBodyIteratorFunc)EachBody,
&amp;allSleeping);
printf("All are sleeping: %s\n", allSleeping ? "true" : "false");
```

```
typedef void (*cpSpaceShapeIteratorFunc)(cpShape *shape, void *data)
void cpSpaceEachShape(cpSpace *space, cpSpaceShapeIteratorFunc func, void
*data)
```

为空间中的每个形状调用func函数，同时传递data指针。休眠和静态形状被包括在内。

```
typedef void (*cpSpaceConstraintIteratorFunc)(cpConstraint *constraint,
void *data)
void cpSpaceEachConstraint(cpSpace *space, cpSpaceConstraintIteratorFunc
func, void *data)
```

为空间中的每个约束调用func函数同时传递data指针。

注意：如果你的编译器支持闭包(如Clang), 那么有另外一组函数你可以调用。cpSpaceEachBody_b()

等等。更多信息请查看chipmunk.h。

## 空间模拟

```
void cpSpaceStep(cpSpace *space, cpFloat dt)
```

通过给定的时间步来更新空间。强烈推荐使用一个固定的时间步长。这样做能大大提高模拟的质量。实现固定的时间步，最简单的方法就是简单的每个帧频步进1/60s（或任何你的目标帧率），而无论花去了多少渲染时间。在许多游戏中这样很有效，但是将物理时间步进和渲染分离是一个更好的方式。这是一篇介绍如何做的好文章。

## 启用和调优空间哈希（散列）

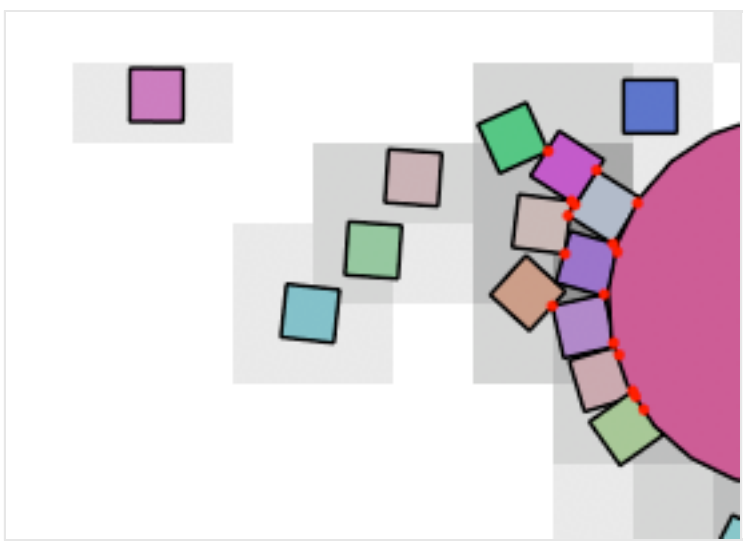
如果你有成千上万个大小大致相同的物体，空间哈希可能会很适合你。

```
void cpSpaceUseSpatialHash(cpSpace *space, cpFloat dim, int count)
```

使空间从碰撞包围盒树切换到空间哈希。 空间哈希数据对大小相当敏感。dim是哈希单元的尺寸。设置dim为碰撞形状大小的平均尺寸可能会得到最好的性能。设置dim太小会导致形状填充进去很多哈希单元，太低会造成过多的物体插入同一个哈希槽。

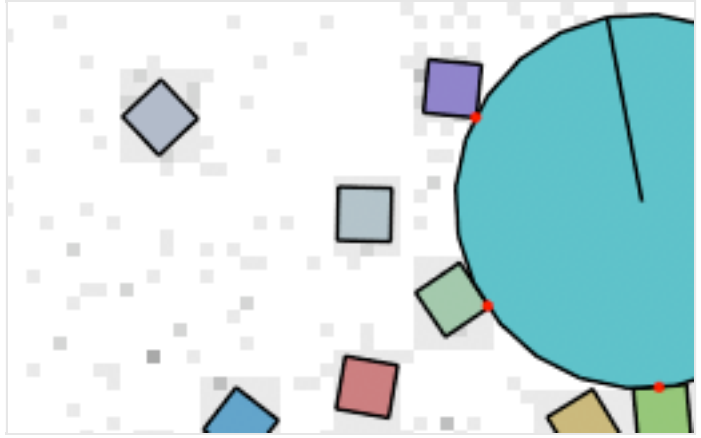
count是在哈希表中建议的最小的单元数量。如果单元太少，空间哈希会产生很多误报。过多的单元将难以做高速缓存并且浪费内存。将count设置成10倍于空间物体的个数可能是一个很好的起点。如果必要的话从那里调优。

关于使用空间哈希有个可视化的演示程序，通过它你可以明白我的意思。灰色正方形达标空间哈希单元。单元颜色越深，就意味着越多的物体被映射到那个单元。一个好的dim尺寸也就是你的物体能够很好的融入格子中。



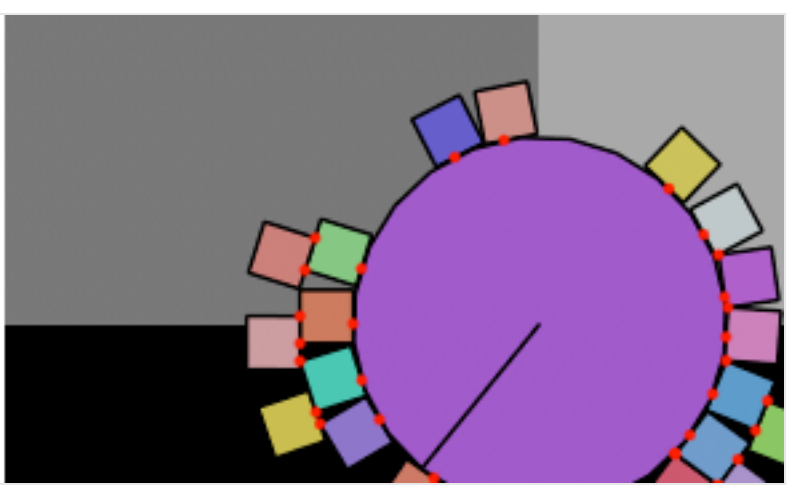
注意，浅色的灰色意味着每个单元没有太多的物体映射到它。

当你使用太小的尺寸，Chipmunk不得不在每个物体上插入很多哈希单元。这个代价有些昂贵。



注意到灰色的单元和碰撞形状相比是非常小的。

当你使用过大的尺寸，就会有很多形状填充进每个单元。每个形状不得不和单元中的其他形状进行检查，所以这会造成许多不必要的碰撞检测。



注意深灰色的单元意味着很多物体映射到了他们。

Chipmunk6也有一个实验性的单轴排序和范围实现。在移动游戏中如果你的世界是很长且扁就像赛车游戏，它是非常高效。如果你想尝试启用它，可以查阅`cpSpaceUseSpatialHash()`的代码。

### 札记

- 当从空间中删除对象时，请确保你已经删除了任何引用它的其他对象。例如，当你删除一个刚体时，要先删除掉关联到刚体的关节和形状。
- 迭代次数和时间步长的大小决定了模拟的质量。越多的迭代次数，或者更小的时间步会提高模拟的质量。请记住，更高质量的同时也意味着更高的CPU使用率。
- 因为静态形状只有当你需要的时候才重新哈希，所以可能会使用一个更大的`count`参数来`cpHashResizeStaticHash()`而不是`cpSpaceResizeActiveHash()`。如果你有大量静态形状的话，这样做会使用更多的内存但是会提升性能。

## Chipmunk约束：cpConstraint

约束是用来描述两个刚体如何相互作用的（他们是如何约束彼此的）。约束可以是允许刚体像我们身体的骨头一样轴转动的简单关节，也可以是更抽象的比如齿轮关节或马达关节。

### 约束是什么，不是什么

在Chipmunk中，约束都是基于速度的约束。这意味着他们主要通过同步两个刚体的速度进行作用。一个轴关节将两个独立刚体的两个锚点连接起来，公式定义要求两个锚点的速度必须相同并且计算施加在刚体上的冲量以便试图保持这个状态。约束将速度视为主要的输入并且产生一个速度变化作为它的输出。一些约束（尤其是关节）通过改变速度来修正位置的差异。更多详情见下一节。

连接两个刚体的弹簧不是一个约束。它很像约束因为它会创建一个力来影响两个刚体的速度，但是弹簧将距离作为输入，将力作为输出。如果弹簧不是一个约束，你会问为什么还会有两种类型的弹簧约束。原因是他们是阻尼弹簧。弹簧关联的阻尼才是真正的约束，这个约束会根据关联的两个刚体的相对速度来创建速度的变化。因为大部分情况将一个阻尼器和一个弹簧放在一起很方便，我想我还不如将弹簧力作为约束的一部分，而不是用一个阻尼器约束然后让用户单独计算和施加弹簧力。

### 属性

- 得到约束关联的两个刚体

```
cpBody * cpConstraintGetA(const cpConstraint *constraint)
cpBody * cpConstraintGetB(const cpConstraint *constraint)
```

- 约束能够作用于两个刚体的最大力。默认为INFINITY。

```
cpFloat cpConstraintGetMaxForce(const cpConstraint *constraint)
void cpConstraintSetMaxForce(cpConstraint *constraint, cpFloat value)
```

- 关节误差百分比一秒钟后仍然没得到修正。这和碰撞偏差机制完全一样，但是这会修正关节的误差而不是重叠碰撞。

```
cpFloat cpConstraintGetErrorBias(const cpConstraint *constraint)
void cpConstraintSetErrorBias(cpConstraint *constraint, cpFloat value)
```

- 约束可以纠错的最大速度。默认为INFINITY。

```
cpFloat cpConstraintGetMaxBias(const cpConstraint *constraint)
void cpConstraintSetMaxBias(cpConstraint *constraint, cpFloat value)
```

- 得到约束所添加进去的空间

```
cpSpace* cpConstraintGetSpace(const cpConstraint *constraint)
```

- 使用数据指针。使用指针来从回调中得到拥有该约束的游戏对象的一个引用。

```
cpDataPointer cpConstraintGetUserData(const cpConstraint *constraint)
void cpConstraintSetUserData(cpConstraint *constraint, cpDataPointer value)
```

- 约束被施加的最新的冲量。为了转化成力，除以cpSpaceStep()传进的时间步。你可以使用这

点来检查施加的力是否超过了一定的阈值从而实现可断裂的关节。

- 断裂关节例子

```
// 创建关节且设置最大力属性
breakableJoint = cpSpaceAddConstraint(space, cpPinJointNew(body1, body2,
cpv(15,0), cpv(-15,0)));
cpConstraintSetMaxForce(breakableJoint, 4000);



// 在update函数中，正常步进模拟空间...
cpFloat dt = 1.0/60.0;
cpSpaceStep(space, dt);

if(breakableJoint){
  // 将冲量除以时间步得到施加的力
  cpFloat force = cpConstraintGetImpulse(breakableJoint)/dt;
  cpFloat maxForce = cpConstraintGetMaxForce(breakableJoint);

  // 如果该力大于设定的阈值则断裂关节
  if(force > 0.9*maxForce){
    cpSpaceRemoveConstraint(space, breakableJoint);
    breakableJoint = NULL;
  }
}
```

要访问特定关节类型的属性，使用提供的getter和setter函数（如cpPinJointGetAnchr1()）。更多信息请查看属性列表。

## 反馈纠错

Chipmunk的关节并不完美。销关节并不能维系两个锚点之间确切的距离，枢轴关节同样也不能保持关联的锚点完全在一起。他们通过自纠错来处理这个问题。在Chipmunk5中，你有很多额外的控制来实现关节对自身的纠错，甚至可以使用这个特性，以独特的方式使用关节来创建一些物理效果。

- 伺服马达：如 打开/关闭门或者旋转物件，无需用最大的力
- 起货机：朝着另外一个物体拉一个物体无需用最大的力
- 鼠标操作：以粗暴、摇晃的鼠标输入方式自如的与物体交互

cpConstraint结构体有3个属性控制着误差纠正，maxForce,maxBias以及biasCoef.maxForce。关节或者约束在不超过该数值大小的力时才能发挥作用。如果它需要更多的力来维系自己，它将会散

架。maxBias是误差纠正可以应用的最大速度了。如果你改变了一个关节的属性，这个关节将不得不自行纠正，一般情况下很快会这么做。通过设置最大速度，你可以像伺服一样使得关节工作，在一段较长的时间以恒定的速率校正自身。最后，biasCoef是在钳位最大值速度前每一步误差纠正的百分比。你可以使用它来使得关节平滑的纠正自身而不是以一个恒定的速度，但可能是三个属性中迄今为止最没用的。

```
// 在一个顶视角的游戏中，采用这种配置的枢轴关节将会计算与地面之间的摩擦
// 因为关节纠正被禁用，所以关节不会重新摆正自身并只会影响速度。
// 当速度改变时，关节施加的力会被最大力钳位
// 这样它就会像摩擦一样工作
cpConstraint *pivot = cpSpaceAddConstraint(space,
cpPivotJointNew2(staticBody, body, cpvzero, cpvzero));
pivot->maxBias = 0.0f; // disable joint correction
pivot->maxForce = 1000.0f;

// The pivot joint doesn't apply rotational forces, use a gear joint with a
ratio of 1.0 for that.
cpConstraint *gear = cpSpaceAddConstraint(space, cpGearJointNew(staticBody,
body, 0.0f, 1.0f));
gear->maxBias = 0.0f; // disable joint correction
gear->maxForce = 5000.0f;

// 另外，你可以将关节连接到一个无限大质量的流氓刚体上来取代连接到一个静态刚体上
// 你可以使用流氓刚体作为控制刚体来连接。可以查看`Tank`演示例子。
```

### 约束和碰撞形状

约束和碰撞形状互不了解双方信息。当为刚体连接关节时，锚点不必处于刚体形状的内部，这么做通常是有意义的。同样的，为两个刚体添加约束并不能阻止刚体形状碰撞。事实上，这便是碰撞组属性存在的主要原因。

### 现有关节类型视频演示

- Youtube地址
- 优酷地址

### 共享内存管理函数

Destroy和Free函数由所有关节类型共享。Allocation和init函数对于每种关节类型都是特定的。

## 约束类型

关节

```
cpPinJoint *cpPinJointAlloc(void)
cpPinJoint *cpPinJointInit(cpPinJoint *joint, cpBody *a, cpBody *b, cpVect anchr1, cpVect anchr2)
cpConstraint *cpPinJointNew(cpBody *a, cpBody *b, cpVect anchr1, cpVect anchr2)
```

a和b是被连接的两个刚体，anchr1和anchr2是这两个刚体的锚点。当关节被创建的时候距离便被确定，如果你想要设定一个特定的距离，使用setter函数来重写它。

属性

- cpVect cpPinJointGetAnchr1(const cpConstraint *constraint)
- void cpPinJointSetAnchr1(cpConstraint *constraint, cpVect value)
- cpVect cpPinJointGetAnchr2(const cpConstraint *constraint)
- void cpPinJointSetAnchr2(cpConstraint *constraint, cpVect value)
- cpFloat cpPinJointGetDist(const cpConstraint *constraint)
- void cpPinJointSetDist(cpConstraint *constraint, cpFloat value)

滑动关节

```
cpSlideJoint *cpSlideJointAlloc(void)

cpSlideJoint *cpSlideJointInit(
    cpSlideJoint *joint, cpBody *a, cpBody *b,
    cpVect anchr1, cpVect anchr2, cpFloat min, cpFloat max
)

cpConstraint *cpSlideJointNew(cpBody *a, cpBody *b, cpVect anchr1, cpVect anchr2, cpFloat min, cpFloat max)
```

a和b是被连接的两个刚体，anchr1和anchr2是这两个刚体的锚点， min和max定义了两个锚点间的最小最大距离。

属性

- cpVect cpSlideJointGetAnchr1(const cpConstraint *constraint)
- void cpSlideJointSetAnchr1(cpConstraint *constraint, cpVect value)
- cpVect cpSlideJointGetAnchr2(const cpConstraint *constraint)
- void cpSlideJointSetAnchr2(cpConstraint *constraint, cpVect value)
- cpFloat cpSlideJointGetMin(const cpConstraint *constraint)

- void cpSlideJointSetMin(cpConstraint *constraint, cpFloat value)
- cpFloat cpSlideJointGetMax(const cpConstraint *constraint)
- void cpSlideJointSetMax(cpConstraint *constraint, cpFloat value)

枢轴关节

```
cpPivotJoint *cpPivotJointAlloc(void)
cpPivotJoint *cpPivotJointInit(cpPivotJoint *joint, cpBody *a, cpBody *b,
cpVect pivot)
cpConstraint *cpPivotJointNew(cpBody *a, cpBody *b, cpVect pivot)
cpConstraint *cpPivotJointNew2(cpBody *a, cpBody *b, cpVect anchr1, cpVect
anchr2)
```

a和b是关节连接的两个刚体，pivot是世界坐标系下的枢轴点。因为枢轴点位置是在世界坐标系下，所以你必须确保两个刚体已经处于正确的位置上。另外你可以指定基于一对锚点的轴关节，但是要确保刚体处于正确的位置上因为一旦你空间开启了模拟，关节将会修正它自身。

属性

- cpVect cpPivotJointGetAnchr1(const cpConstraint *constraint)
- void cpPivotJointSetAnchr1(cpConstraint *constraint, cpVect value)
- cpVect cpPivotJointGetAnchr2(const cpConstraint *constraint)
- void cpPivotJointSetAnchr2(cpConstraint *constraint, cpVect value)

沟槽关节

```
cpGrooveJoint *cpGrooveJointAlloc(void)

cpGrooveJoint *cpGrooveJointInit(
    cpGrooveJoint *joint, cpBody *a, cpBody *b,
    cpVect groove_a, cpVect groove_b, cpVect anchr2
)

cpConstraint *cpGrooveJointNew(cpBody *a, cpBody *b, cpVect groove_a,
cpVect groove_b, cpVect anchr2)
```

沟槽在刚体a上从groov_a到groov_b，枢轴被附加在刚体b的anchr2锚点上。所有的坐标都是刚体局部坐标。

属性

- cpVect cpGrooveJointGetGrooveA(const cpConstraint *constraint)

- void cpGrooveJointSetGrooveA(cpConstraint *constraint, cpVect value)
- cpVect cpGrooveJointGetGrooveB(const cpConstraint *constraint)
- void cpGrooveJointSetGrooveB(cpConstraint *constraint, cpVect value)
- cpVect cpGrooveJointGetAnchr2(const cpConstraint *constraint)
- void cpGrooveJointSetAnchr2(cpConstraint *constraint, cpVect value)

阻尼弹簧

```
cpDampedSpring *cpDampedSpringAlloc(void)

cpDampedSpring *cpDampedSpringInit(
    cpDampedSpring *joint, cpBody *a, cpBody *b, cpVect anchr1, cpVect
anchr2,
    cpFloat restLength, cpFloat stiffness, cpFloat damping
)

cpConstraint *cpDampedSpringNew(
    cpBody *a, cpBody *b, cpVect anchr1, cpVect anchr2,
    cpFloat restLength, cpFloat stiffness, cpFloat damping
)
```

和滑动关节的定义很类似。`restLength`是弹簧想要的长度，`stiffness`是弹簧系数（Young's modulus），`dampling`用来描述弹簧阻尼的柔软度。

属性

- cpVect cpDampedSpringGetAnchr1(const cpConstraint *constraint)
- void cpDampedSpringSetAnchr1(cpConstraint *constraint, cpVect value)
- cpVect cpDampedSpringGetAnchr2(const cpConstraint *constraint)
- void cpDampedSpringSetAnchr2(cpConstraint *constraint, cpVect value)
- cpFloat cpDampedSpringGetRestLength(const cpConstraint *constraint)
- void cpDampedSpringSetRestLength(cpConstraint *constraint, cpFloat value)
- cpFloat cpDampedSpringGetStiffness(const cpConstraint *constraint)
- void cpDampedSpringSetStiffness(cpConstraint *constraint, cpFloat value)
- cpFloat cpDampedSpringGetDamping(const cpConstraint *constraint)
- void cpDampedSpringSetDamping(cpConstraint *constraint, cpFloat value)

阻尼旋转弹簧

```
cpDampedRotarySpring *cpDampedRotarySpringAlloc(void)
```

```
cpDampedRotarySpring *cpDampedRotarySpringInit(
    cpDampedRotarySpring *joint, cpBody *a, cpBody *b,
    cpFloat restAngle, cpFloat stiffness, cpFloat damping
)

cpConstraint *cpDampedRotarySpringNew(cpBody *a, cpBody *b, cpFloat
restAngle, cpFloat stiffness, cpFloat damping)
```

犹如阻尼弹簧，但却在角度层面起作用。restAngle是刚体间想要的相对角度，stiffness和dampling和阻尼弹簧的基本一样。

属性

- cpFloat cpDampedRotarySpringGetRestAngle(const cpConstraint *constraint)
- void cpDampedRotarySpringSetRestAngle(cpConstraint *constraint, cpFloat value)
- cpFloat cpDampedRotarySpringGetStiffness(const cpConstraint *constraint)
- void cpDampedRotarySpringSetStiffness(cpConstraint *constraint, cpFloat value)
- cpFloat cpDampedRotarySpringGetDamping(const cpConstraint *constraint)
- void cpDampedRotarySpringSetDamping(cpConstraint *constraint, cpFloat value)

旋转限位关节

```
cpRotaryLimitJoint *cpRotaryLimitJointAlloc(void)
cpRotaryLimitJoint *cpRotaryLimitJointInit(cpRotaryLimitJoint *joint,
cpBody *a, cpBody *b, cpFloat min, cpFloat max)
cpConstraint *cpRotaryLimitJointNew(cpBody *a, cpBody *b, cpFloat min,
cpFloat max)
```

旋转限位关节约束着两个刚体间的相对角度。min和max就是最小和最大的相对角度，单位为弧度。它被实现以便可能使范围大于一整圈。

属性

- cpFloat cpRotaryLimitJointGetMin(const cpConstraint *constraint)
- void cpRotaryLimitJointSetMin(cpConstraint *constraint, cpFloat value)
- cpFloat cpRotaryLimitJointGetMax(const cpConstraint *constraint)
- void cpRotaryLimitJointSetMax(cpConstraint *constraint, cpFloat value)

棘轮关节

```
cpRatchetJoint *cpRatchetJointAlloc(void);
cpRatchetJoint *cpRatchetJointInit(cpRatchetJoint *joint, cpBody *a, cpBody
```

```
*b, cpFloat phase, cpFloat ratchet);
cpConstraint *cpRatchetJointNew(cpBody *a, cpBody *b, cpFloat phase,
cpFloat ratchet);
```

工作起来像套筒扳手。ratchet是”clicks”间的距离，phase是当决定棘轮角度的时候的初始位移。

属性

- cpFloat cpRatchetJointGetAngle(const cpConstraint *constraint)
- void cpRatchetJointSetAngle(cpConstraint *constraint, cpFloat value)
- cpFloat cpRatchetJointGetPhase(const cpConstraint *constraint)
- void cpRatchetJointSetPhase(cpConstraint *constraint, cpFloat value)
- cpFloat cpRatchetJointGetRatchet(const cpConstraint *constraint)
- void cpRatchetJointSetRatchet(cpConstraint *constraint, cpFloat value)

齿轮关节

```
cpGearJoint *cpGearJointAlloc(void);
cpGearJoint *cpGearJointInit(cpGearJoint *joint, cpBody *a, cpBody *b,
cpFloat phase, cpFloat ratio);
cpConstraint *cpGearJointNew(cpBody *a, cpBody *b, cpFloat phase, cpFloat
ratio);
```

齿轮关节保持着一对刚体恒定的角速度比。ratio总是测量绝对值，目前无法设定相对于第三个刚体的角速度。phase是两个刚体的初始角度偏移量。

属性

- cpFloat cpGearJointGetPhase(const cpConstraint *constraint)
- void cpGearJointSetPhase(cpConstraint *constraint, cpFloat value)
- cpFloat cpGearJointGetRatio(const cpConstraint *constraint)
- void cpGearJointSetRatio(cpConstraint *constraint, cpFloat value)

简单马达

```
cpSimpleMotor *cpSimpleMotorAlloc(void);
cpSimpleMotor *cpSimpleMotorInit(cpSimpleMotor *joint, cpBody *a, cpBody
*b, cpFloat rate);
cpConstraint *cpSimpleMotorNew(cpBody *a, cpBody *b, cpFloat rate);
```

简单马达保持着一对刚体恒定的角速度比。rate是所需的相对角速度。通常你会给马达设定一个最大力（扭矩）否则他们会申请一个无限大的扭矩来使得刚体移动。

### 属性

- cpFloat cpSimpleMotorGetRate(const cpConstraint *constraint)
- void cpSimpleMotorSetRate(cpConstraint *constraint, cpFloat value)

### 札记

- 你可以为两个刚体添加多个关节，但要确保他们彼此不会冲突。否则会引起刚体抖动或者剧烈的旋转。

## Chipmunk碰撞检测概述

Chipmunk为了使得碰撞检测尽可能快，将处理过程分成了若干阶段。虽然我一直试图保持它概念简单，但实现却有点让人生畏。幸运的是作为Chipmunk库的使用者，你并不需要了解一切关于它是如何工作的。但如果你在尝试发挥Chipmunk的极致，理解这一部分会有所帮助。

### 空间索引

在场景中用一个for循环来检查每一个对象是否与其他对象发生碰撞会很慢。所以碰撞检测的第一步(或者就像通常称作的阶段)，就是使用高层次空间算法来找出哪些对象应该被检查碰撞。目前Chipmunk支持两种空间索引，轴对齐包围盒树和空间散列。这些空间索引能够快速识别哪些形状彼此靠近，并应做碰撞检查。

### 碰撞过滤（筛选）

在空间索引找出彼此靠近的形状对后，将它们传给space，然后再执行一些额外的筛选。在进行任何操作前，Chipmunk会执行几个简单的测试来检测形状是否会发生碰撞。

- 包围盒测试：如果形状的包围盒没有重叠，那么形状便没发生碰撞。对象如对角线线段会引发许多误报，但你不应该担心。
- 层测试：如果形状不在同一层内则不会发生碰撞。（他们的层掩码按位与运算结果为0）
- 群组测试：在相同的非零群组中的形状不会发生碰撞。

### 基本形状与形状间的碰撞检测

最昂贵的测试其实就是检测基于几何形状的重叠。圆与圆，圆与线之间的碰撞检测相当快，多边形和多边形的碰撞检测随着顶点数的增加而更加昂贵。形状越简单，碰撞检测就越快（更重要的是求解器检测的碰撞点就越少）。Chipmunk使用了一个分发表来描述应该使用哪个函数来检测形状是否重叠。

### 碰撞处理函数过滤

在检测到两个形状间重叠之后，Chipmunk会查看你是否为该碰撞形状的类型定义了一个碰撞处理函数。对于游戏这样去处理碰撞事件是至关重要的，同时也为你提供了一个非常灵活的方式来过滤掉碰撞。`begin()`和`preSolve()`回调函数的返回值决定了碰撞的形状对是否该舍弃掉。返回`true`会保留

形状对，false则会舍弃。在begin()回调中中止一个碰撞是永久性的，在preSolve()回调中中止只是应用于当前所处的时间步。如果你没有为碰撞类型定义一个处理函数，Chipmunk将会调用space的默认处理函数，默认会简单的接受所有碰撞。

使用回调过滤碰撞是最灵活的方式，记住，到那时候所有最昂贵的碰撞检测通过你的回调都已经完成。对于每帧有大量碰撞对象的模拟，寻找碰撞所消耗的时间和解决碰撞所消耗的时间相比要小很多，所以这不是一个大问题。不过，如果可以的话先使用层或者群组。

## 碰撞回调

没有任何事件或反馈的物理库对游戏而言帮助并不大。你怎么知道当玩家碰到了一个敌人，以便你扣除一些生命点数？你怎么知道汽车撞击一个东西的力度，这样你就不会在石子击中它的时候播放一个巨响轰隆音？如果你需要决定在特定条件下的碰撞是否应该被忽略，比如要实现单向平台？Chipmunk拥有一套强大的回调系统，你可以实现他们来完成这一切。

### 碰撞处理

碰撞处理函数是Chipmunk能够识别的不同碰撞事件的一组4个回调函数。事件类型是：

- begin():该步中两个形状刚开始第一次接触。回调返回true则会处理正常碰撞，返回falseChipmunk会完全忽略碰撞。如果返回false，则preSolve()和postSolve()回调将永远不会被执行，但你仍然会在形状停止重叠的时候接收到一个单独的事件。
- preSolve():该步中两个形状相互接触。回调返回falseChipmunk在这一步会忽略碰撞，返回true来正常处理它。此外，你可以使用cpArbiterSetFriction()，cpArbiterSetElasticity()或cpArbiterSetSurfaceVelocity()来提供自定义的摩擦，弹性，或表面速度值来覆盖碰撞值。更多信息请查看cpArbiter。
- postSolve():两种形状相互接触并且它们的碰撞响应已被处理。如果你想使用它来计算音量或者伤害值，这时你可以检索碰撞冲力或动能。更多信息请查看cpArbiter。
- separate():该步中两个形状刚第一次停止接触。确保begin()/separate()总是被成对调用，当删除接触中的形状时或者析构space时它也会被调用。

碰撞回调都与cpArbiter结构紧密相关。你应该熟悉那些为好。

注：标记为传感器的形状（cpShape.sensor == true）从来不会得到碰撞处理，所以传感器形状和其他形状间永远不会调用postSolve()回调。它们仍然会调用begin()和separate()回调，而preSolve()仍然会在每帧调用回调，即使这里不存在真正的碰撞。

注2：preSolve()回调在休眠算法运行之前被调用。如果一个对象进入休眠状态，postSolve()回调将不会被调用，直到它被唤醒。

### 碰撞处理API

```
typedef int (*cpCollisionBeginFunc)(cpArbiter *arb, struct cpSpace *space,
void *data)
typedef int (*cpCollisionPreSolveFunc)(cpArbiter *arb, cpSpace *space, void
*data)
typedef void (*cpCollisionPostSolveFunc)(cpArbiter *arb, cpSpace *space,
void *data)
typedef void (*cpCollisionSeparateFunc)(cpArbiter *arb, cpSpace *space,
void *data)
```

碰撞处理函数类型。所有这些函数都附带一个arbiter，space和用户data指针，只有begin()和preSolve()回调会返回值。更多信息请查看上方碰撞处理。

```
void cpSpaceAddCollisionHandler(
    cpSpace *space,
    cpCollisionType a, cpCollisionType b,
    cpCollisionBeginFunc begin,
    cpCollisionPreSolveFunc preSolve,
    cpCollisionPostSolveFunc postSolve,
    cpCollisionSeparateFunc separate,
    void *data
)
```

为指定的碰撞类型对添加一个碰撞处理函数。每当碰撞类型（cpShape.collision_type）为a的形状与碰撞类型为b的形状碰撞时，这些回调就会被调用来处理碰撞。data是用户定义的上下文指针，用来传递到每个回调中。你不想实现的话可以使用NULL，然而Chipmunk会调用它自身的默认版本而不是你为space设置的默认值。如果你需要依赖space的默认回调，你必须单独为每个处理函数定义提供实现。

```
void cpSpaceRemoveCollisionHandler(cpSpace *space, cpCollisionType a,
cpCollisionType b)
```

移除指定碰撞类型对的碰撞处理函数。

```
void cpSpaceSetDefaultCollisionHandler(
    cpSpace *space,
    cpCollisionBeginFunc begin,
    cpCollisionPreSolveFunc preSolve,
    cpCollisionPostSolveFunc postSolve,
    cpCollisionSeparateFunc separate,
    void *data
)
```

当没有具体的碰撞处理时Chipmunk会使用一个默认的注册碰撞处理函数。space在创建时被指定了一个默认的处理函数，该函数在`begin()`和`preSolve()`回调中返回`true`，在`postSolve()`和`separate()`回调中不做任何事情。

### 后步回调

后步回调允许你打破在一个回调内增加或删除对象的规则。事实上，它们的主要功能就是帮助你安全的从你想要禁用/破坏一个碰撞/查询回调的空间中移除对象。

后步回调被注册为一个函数和用作键的一个指针。你只能为每个键注册一个`postStep()`回调。这可以防止你不小心多次删除对象。例如，假设你有子弹和对象A之间的碰撞回调。你想摧毁子弹和对象A，因此你注册一个`postStep()`回调来从游戏中安全地移除它们。然后，你得到子弹和对象B之间的碰撞回调，你注册一个`postStep()`回调来删除对象B，第二次`postStep()`回调来移除子弹。因为你只能为每个键注册一次回调， `postStep()`回调对于子弹而言只会被调用一次，你不可能意外删除两次。

### 例子

更多信息请查看碰撞回调范例。

## Chipmunk碰撞对：cpArbiter

Chipmunk的`cpArbiter`结构封装了一对碰撞的形状和关于它们的所有碰撞数据。

为什么称之为仲裁者？简短来说，我一直用的是“仲裁”来形容碰撞解决的方式，然后早在2006年当我在看Box2D的求解器的时候看到了Box2D居然叫它们仲裁者。仲裁者就像是一个法官，有权力来解决两个人之间的纠纷。这是有趣的，使用了合适的名字并且输入比我以前用的`CollisionPair`要短。它最初只是被设定为一个私有的内部结构，但却在回调中很有用。

### 内存管理

你永远不需要创建或释放一个仲裁者。更重要的是，因为它们完全由空间管理，所以你永远不应该存储一个仲裁者的引用，因为你不知道它们什么时候会被释放或重新使用。在回调中使用它们，然后忘记它们或复制出你需要的信息。

### 属性

```
cpFloat cpArbiterGetElasticity(const cpArbiter *arb)
void cpArbiterSetElasticity(cpArbiter *arb, cpFloat value)
```

计算碰撞对的弹性。在`preSolve()`回调中设定该值将会覆盖由空间计算的值。默认计算会将两个形状的弹性相乘。

```
cpFloat cpArbiterGetFriction(const cpArbiter *arb)
void cpArbiterSetFriction(cpArbiter *arb, cpFloat value)
```

计算碰撞对的摩擦力。在preSolve()回调中设定该值将会覆盖由空间计算的值。默认计算会将两个形状的摩擦力相乘。

```
cpVect cpArbiterGetSurfaceVelocity(const cpArbiter *arb)
void cpArbiterSetSurfaceVelocity(cpArbiter *arb, cpVect value)
```

计算碰撞对的表面速度。在preSolve()回调中设定该值将会覆盖由空间计算的值。默认计算会将第二个形状的表面速度从第一个形状的表面速度中减去，然后投射到碰撞的切线上。这使得只有摩擦力受到默认计算的影响。使用自定义计算，你可以使得响应就像一个弹球保险杠一样，或使得表面速度依赖于接触点的位置。

注：不幸的是，有一个老的bug会让表面速度计算逆向（负值）。我真的很久没有注意到这点了。这将在Chipmunk7中得到修正，但现在由于向后兼容的原因我已经先不管它了。

```
cpDataPointer cpArbiterGetUserData(const cpArbiter *arb)
void cpArbiterSetUserData(cpArbiter *arb, cpDataPointer data)
```

用户自定义指针。该值将维持形状对直到separate()回调被调用。

注：如果你需要清理这个指针，你应该实现separate()回调来这么做。同时在摧毁空间的时候要小心，因为仍然有可能有激活的碰撞存在。为了触发separate()回调，在处置它之前你需要先移除空间中的所有形状。这正是我建议的方式。见
ChipmunkDemo.c：ChipmunkDemoFreeSpaceChildren()演示了如何轻松做到这一点。

```
int cpArbiterGetCount(const cpArbiter *arb)
cpVect cpArbiterGetNormal(const cpArbiter *arb, int i)
cpVect cpArbiterGetPoint(const cpArbiter *arb, int i)
cpFloat cpArbiterGetDepth(const cpArbiter *arb, int i)
```

得到由这仲裁者或特定碰撞点，碰撞点的法向量或深度穿透跟踪的触点的数目。

```
cpBool cpArbiterIsFirstContact(const cpArbiter *arb)
```

如果这是两个形状开始接触的第一步则返回true。举例来说这对于声音效果很有用。如果这是特定碰撞的第一帧，在postStep()回调中检测碰撞能量，并用它来确定播放的声效音量。

```
void cpArbiterGetShapes(const cpArbiter *arb, cpShape **a, cpShape **b)
void cpArbiterGetBodies(const cpArbiter *arb, cpBody **a, cpBody **b)
```

按照形状或者刚体在该仲裁者关联的碰撞对中定义的顺序一样得到它们。如果你像cpSpaceAddCollisionHandler(space, 1, 2, …)定义了个函数，你会发现a->collision_type == 1且b->collision_type == 2。

碰撞回调例子

```
static void
postStepRemove(cpSpace *space, cpShape *shape, void *unused)
{
  cpSpaceRemoveShape(space, shape);
  cpSpaceRemoveBody(space, shape->body);

  cpShapeFree(shape);
  cpBodyFree(shape->body);
}

static int
begin(cpArbiter *arb, cpSpace *space, void *unused)
{
  // 得到参与碰撞的形状
  // 顺序和你在函数定义中的顺序一致
  // a->collision_type将是BULLET_TYPE, b->collision_type将是MONSTER_TYPE
  CP_ARBITER_GET_SHAPES(arb, a, b);

  // 宏展开后和下面输入一样
  // cpShape *a, *b; cpArbiterGetShapes(arb, &amp;a, &amp;b);

  // 添加一个后步回调来安全从空间中移除和刚体
  // 直接从碰撞处理函数回调中调用 cpSpaceRemove() 会引起崩溃
  cpSpaceAddPostStepCallback(space, (cpPostStepFunc)postStepRemove, b,
NULL);

  // 物体死亡，不再处理碰撞
  return 0;
}

#define BULLET_TYPE 1
#define MONSTER_TYPE 2

// 为子弹和怪物定义一个碰撞处理函数
// 一旦怪物被子弹击中则通过移除它的形状和刚体来立马杀死怪物
cpSpaceAddCollisionHandler(space, BULLET_TYPE, MONSTER_TYPE, begin, NULL,
NULL, NULL, NULL);
```

### 触点集

通过触点集我们得到接触信息变得更为容易。

```
cpContactPointSet cpArbiterGetContactPointSet(const cpArbiter *arb)
```

从仲裁者中得到的触点集结构域。

你可能通过下面的方式来得到并且处理一个触点集:

```
cpContactPointSet set = cpArbiterGetContactPointSet(arbiter);
for(int i=0; i&lt;set.count; i++){
    // 得到并使用触点的法向量和穿透距离
    set.points[i].point
    set.points[i].normal
    set.points[i].dist
}
```

```
void cpArbiterSetContactPointSet(cpArbiter *arb, cpContactPointSet *set)
```

替换仲裁者的触点集。你不能改变触点的数目，但是可以改变它们的位置，法向量或穿透距离。Sticky演示使用它来使得物体能够获得额外量的重叠。你也可以在乒乓式风格游戏中使用它来修改基于碰撞x轴的碰撞法向量，即使板子是扁平形状。

### 帮助函数

```
void cpArbiterGetShapes(cpArbiter *arb, cpShape **a, cpShape **b)
void cpArbiterGetBodies(const cpArbiter *arb, cpBody **a, cpBody **b)
```

得到在仲裁者关联的碰撞处理中所定义的形状（或者刚体）。如果你定义了一个处理函数如cpSpaceAddCollisionHandler(space, 1, 2, …)，你会发现a->collision_type == 1并且b->collision_type == 2。便捷的宏为你定义并且初始化了两个形状变量。默认的碰撞处理函数不会使用碰撞类型，所以顺序是未定义的。

```
#define CP_ARBITER_GET_SHAPES(arb, a, b) cpShape *a, *b;
cpArbiterGetShapes(arb, &amp;a, &amp;b)
#define CP_ARBITER_GET_BODIES(arb, a, b) cpBody *a, *b;
cpArbiterGetBodies(arb, &amp;a, &amp;b);
```

定义变量并且从仲裁者中检索形状/刚体所用的缩略宏。

```
cpVect cpArbiterTotalImpulseWithFriction(cpArbiter *arb);
cpVect cpArbiterTotalImpulse(cpArbiter *arb);
```

返回为解决碰撞而施加于此步骤的冲量。这些函数应该只在postStep()或cpBodyEachArbiter()回调中被调用，否则结果将不可确定。如有疑问不知道该使用哪个函数，就使用cpArbiterTotalImpulseWithFriction()。

```
cpFloat cpArbiterTotalKE(const cpArbiter *arb);
```

计算在碰撞中的能量损失值，包括静摩擦不包括动摩擦。这个函数应该在postSolve(),postStep()或者cpBodyEachArbiter()回调中被调用。

## 查询

Chipmunk空间支持4种空间查询，包括最近点查询、线段查询、形状查询和快速包围盒查询。任何一种类型都可在空间里有效运行，并且点和线段查询可以针对单个形状来进行。所有类型的查询需要一个碰撞组和层，并使用和过滤形状间碰撞一样的规则来过滤出匹配。如果你不希望过滤掉任何匹配，使用CP_ALL_LAYERS作为层，CP_NO_GROUP作为组。

### 最近点查询

点查询对于像鼠标拾取和简单的感应器来说非常有用。它允许你检查离给定点一定距离内是否存在着形状，找到形状上离给定点最近的点或者找到离给定点最近的形状。

```
typedef struct cpNearestPointQueryInfo {
    /// 最近的形状。如果在范围内没有形状返回NULL。
    cpShape *shape;
    /// 形状表面上最近点（世界坐标系）
    cpVect p;
    /// 离给定点的距离。如果点在形状内部距离则为负值
    cpFloat d;
    /// 距离函数的梯度
    /// 和info.p/info.d相同，即使当info.d是非常小的值时，仍然精确
    cpVect g;
} cpNearestPointQueryInfo;
```

### 线段查询

线段查询就像射线投射一样，但由于并非所有的空间索引都允许处理无限长的射线查询所以它仅限于线段。在实践中这仍然非常快速，你不用过多的担心过长的线段查询会影响到性能。

```
typedef struct cpSegmentQueryInfo {
    //碰撞的形状，如果没有碰撞发生则为NULL
    cpShape *shape;
    // 线段查询的归一化距离，在[0,1]范围内
```

```
    cpFloat t;
    // 表面命中点的法向量
    cpVect n;
} cpSegmentQueryInfo;
```

分类查询返回的信息不只是一个简单的是或否，它们也会返回形状被击中的位置以及被击中位置的表面的法向量。t是该查询的开始点和结束点之间的百分比。如果你需要世界空间中的击中点或者到开始点的绝对距离，可以查看下面的线段查询帮助函数。如果线段查询的开始点在形状内部，则t = 0并且n = cpvzero。

```
cpBool cpShapeSegmentQuery(cpShape *shape, cpVect a, cpVect b,
cpSegmentQueryInfo *info)
```

执行从a到b的线段与单一形状shape的线段查询。info必须是一个指向cpSegmentQueryInfo结构体的有效的指针，该结构体会被光线投射信息（raycast info）初始化。

```
typedef void (*cpSpaceSegmentQueryFunc)(cpShape *shape, cpFloat t, cpVect
n, void *data)


void cpSpaceSegmentQuery(
    cpSpace *space, cpVect start, cpVect end,
    cpLayers layers, cpGroup group,
    cpSpaceSegmentQueryFunc func, void *data
)
```

沿着线段的start到end使用给定的layers和groups来查询space过滤筛选出匹配。func函数被调用，附带着线段和任何被发现的形状表面的法向量之间的归一化距离，还有传递给cpSpacePointQuery()的data参数。传感器类形状也被包括在内。

```
cpShape *cpSpaceSegmentQueryFirst(
    cpSpace *space, cpVect start, cpVect end,
    cpLayers layers, cpGroup group,
    cpSegmentQueryInfo *info
)
```

沿着线段的start到end使用给定的layers和groups来查询space过滤筛选出匹配。只有遇到的第一个形状会被返回并结束搜索，如果没有发现形状则返回NULL。info指向的结构体将会被光线投射信息初始化，除非info是NULL。传感器类形状将被忽略。

**线段查询辅助函数:**

```
cpVect cpSegmentQueryHitPoint(cpVect start, cpVect end, cpSegmentQueryInfo
```

```
info)
```

返回在世界坐标系内线段与形状相交的第一个相交点。

```
cpFloat cpSegmentQueryHitDist(cpVect start, cpVect end, cpSegmentQueryInfo
info)
```

返回线段与形状第一个相交点的绝对距离。

### AABB查询

AABB查询提供一个快速的方式来粗略检测一个范围内存在的形状。

```
typedef void (*cpSpaceBBQueryFunc)(cpShape *shape, void *data)

void cpSpaceBBQuery(
    cpSpace *space, cpBB bb,
    cpLayers layers, cpGroup group,
    cpSpaceBBQueryFunc func, void *data
)
```

查询space找到bb附近并筛选出符合给定层和组的所有形状。每个包围盒和bb有重叠的形状，都会调用func, 并将data参数传给cpSpaceBBQuery()。传感器类形状也包括在内。

### 形状查询

形状查询允许你检测空间中的形状是否和一个指定的区域发生了重叠。如果你想在该位置添加另外一个形状，又或者在AI中使用它进行感应查询的话。你可以通过形状查询来检测物体是否已经存在于一个位置上。

在查询前，你可以创建一个刚体对或者形状对，或者你创建一个shape值为NULL的刚体，通过调用cpShapeUpdate()函数来设置形状的位置和旋转角度。

```
typedef void (*cpSpaceShapeQueryFunc)(cpShape *shape, cpContactPointSet
*points, void *data);

cpBool cpSpaceShapeQuery(cpSpace *space, cpShape *shape,
cpSpaceShapeQueryFunc func, void *data);
```

查询space来找到和shape重叠的所有形状。使用shape的层和群组来过滤筛选得到匹配。func函数由每个重叠的形状调用，附带一个临时的cpContactPointSet的一个指针和传递给cpSpaceBBQuery()的data参数。传感器类形状也包括在内。

## 闭包

如果你的编译器支持闭包(如Clang), 还有另外一组函数可以调用，如`cpSpaceNearestPointQuery_b()`等。详情请参考`chipmunk.h`。

## 例子

更多信息见查询例子