Box2D v2.3.0 用户手册

Copyright © 2007-2013 Erin Catto 著

Copyright © 2015 Antkillerfarm译

基于

Aman JIANG(江超宇)的v2.0.1用户手册

complex\_ok的2.1.0用户手册

Chapter 1 导言 7

1.1 关于 7

1.2 先决条件 7

1.3 关于手册 7

1.4 反馈和报告BUG 8

1.5 核心概念 8

形状(shape) 8

刚体(rigid body) 8

夹具(fixture) 8

约束(constraint) 8

接触约束(contact constraint) 9

关节(joint) 9

关节限制(joint limit) 9

关节马达(joint motor) 9

世界(world) 9

求解器(solver) 9

连续碰撞(continuous collision) 9

1.6 模块 10

1.7 单位 11

1.8 工厂和定义 11

Chapter 2 Hello Box2D 13

2.1 创建世界 13

2.2 创建地面盒 13

2.3 创建动态物体 14

2.4 模拟(Box2D的)世界 15

2.5 清理 17

2.6 The Testbed 17

Chapter 3 通用模块 19

3.1 关于 19

3.2 配置 19

类型 19

常数 19

分配器包装 19

版本号 19

3.3 内存管理 19

3.4 数学 20

Chapter 4 碰撞模块 21

4.1 关于 21

4.2 形状 21

圆形 21

多边形 22

边框形状（Edge shapes） 24

链接形状（Chain Shapes） 25

4.3 单元几何查询（Unary Geometric Queries） 26

形状点测试（Shape Point Test） 26

形状的光线投射(Shape Ray Cast) 27

4.4 对等函数 27

重叠 27

接触形（Contact Manifolds） 28

距离 29

撞击时间 29

4.5 动态树 30

4.6 Broad-phase 31

Chapter 5 力学模块 33

5.1 概述 33

Chapter 6 物体 34

6.1 关于 34

b2\_staticBody 34

b2\_kinematicBody 34

b2\_dynamicBody 34

6.2 物体定义 34

物体类型 35

位置和角度 35

阻尼 35

重力因子 36

休眠参数 36

固定旋转 36

子弹 36

活动状态 37

用户数据 37

6.3 物体工厂（Body Factory） 37

6.4 使用物体 38

质量数据 38

状态信息 39

位置和速度 39

Chapter 7 夹具 40

7.1 关于 40

7.2 创建夹具 40

密度 40

摩擦 41

恢复 41

筛选 41

7.3 传感器 43

Chapter 8 关节 44

8.1 关于 44

8.2 关节定义 44

8.3 关节工厂 44

8.4 使用关节 45

8.5 距离关节 46

8.6 旋转关节 46

8.7 移动关节 48

8.8 滑轮关节 49

8.9 齿轮关节 50

8.10 鼠标关节 51

8.11 轮子关节 51

8.12 焊接关节 52

8.13 绳子关节 52

8.14 摩擦关节 52

8.15 马达关节 52

Chapter 9 接触 53

9.1 关于 53

接触点 53

接触法线 53

接触分隔 53

接触流形 53

法向冲量 53

切向冲量 53

接触标识 54

9.2 接触类 54

9.3 访问接触 55

9.4 接触监听器 55

开始接触事件 56

结束接触事件 56

求解前事件 56

求解后事件 57

9.5 接触筛选 59

Chapter 10 世界类 60

关于 60

创建和摧毁世界 60

使用世界 60

模拟 60

探测世界 61

AABB查询 62

光线投射 63

力与冲量 64

坐标转换 64

列表 64

Chapter 11 杂项 66

11.1 用户数据 66

11.2 隐式摧毁 67

11.3 像素和坐标系统 68

Chapter 12 调试绘图 70

Chapter 13 限制 71

Chapter 14 参考 72

# 导言

## 关于

Box2D 是一个用于游戏的 2D 刚体仿真库。程序员可以在他们的游戏里使用它，它可以使物体的运动更加真实，并让游戏世界看起来更具交互性。从游戏引擎的视角来看，物理引擎就是一个程序性动画 (proceduralanimation)的系统。

(译注: 做动画常有两种方法, 一种是预先准备好动画所需的数据，比如图片，再一帧一帧地播放。另一种是以一定方法，动态计算出动画所需的数据，根据数据再进行绘图。   
从这种角度看，预先准备的，可称为数据性动画，动态计算的可称为程序性动画。  
这个区别，就类似以前我们做历史题和数学题，做历史题，记忆很重要，也就是答案需要预先准备好的。做数学题，方法就很重要，答案是需要用方法推导出来的。   
Box2D就是用物理学的方法，推导出那游戏世界物体的位置，角度等数据。而Box2D也仅仅推导出数据，至于得到数据之后怎么处理就是程序员自己的事情了。)

Box2D 是用可移植的 C++ 写成的。引擎中大部分类型的定义都有 b2 前缀,希望这能有效的消除它和你的游戏引擎之间的名字冲突。

## 先决条件

在此,我假定你已经熟悉了基本的物理学概念，例如质量、力、扭矩和冲量。如果没有，请先查询一下Google搜索和维基百科。

Box2D是游戏开发者大会(Game Developer Conference，GDC)的物理学教程的一部分。你可以从box2d.org的下载页面获得这些教程。

因为 Box2D 是用 C++ 写成的，所以你应该具备 C++ 程序设计的经验。Box2D 不应该成为你的第一个 C++ 程序项目。你应该已经能熟练地编译，链接和调试了。

**注意**

Box2D 不应该成为你的第一个 C++ 程序项目。你应该已经能熟练地编译，链接和调试了。网络上有很多关于C++的资料。

## 关于手册

这个手册包含了主要的Box2D的API，但并不是每一个都包含了 。你可以通过阅读Box2D自带的testbed程序的代码来学习更多的东西。而且Box2D代码的注释已经按照Doxygen格式编写，可以很容易的创建超链接形式的API文档。

这个手册只在新版本发布的时候更新。因此相比版本库中的代码版本，它可能已经过时了。

## 反馈和报告BUG

如果你有关于Box2D的问题或者反馈意见，请在论坛留言。这也是社区讨论的好地方。

Box2D使用 Google code project来跟踪问题。这是个很好的方式，它可以确保你的问题不会淹没在论坛之中。

请在这里列出BUG和功能需求: <http://code.google.com/p/box2d/>

如果你能提供更有效的细节的话，就能确保你的问题得到解决。一个用于复现问题的testbed用例是很有意义的。你可以在随后章节读到和testbed相关的内容。

## 核心概念

Box2D 中有一些基本的概念和对象，这里我们先做一个简要的定义,在随后的章节里会有更详细的描述。

#### 形状(shape)

形状是一个2D的几何对象。例如圆或多边形。

#### 刚体(rigid body)

一块十分坚硬的物质,它上面的任何两点之间的距离都是完全不变的。它们就像钻石那样坚硬。在后面的讨论中,我们用物体(body)来指代刚体。

#### 夹具(fixture)

夹具将形状绑定到物体上，并添加密度(density)、摩擦(friction)和恢复(restitution)等材料特性。夹具还将形状放入碰撞系统(碰撞检测(Broad Phase))中，以使之能与其他形状相碰撞。

(译注:一个物体和另一物体碰撞, 碰撞后速度和碰撞前速度的比值会保持不变，这比值就叫恢复系数。)

(译注: Broad Phase是碰撞检测的一个子阶段, 将空间分割, 每个空间对应一个子树, 物体就放到树中, 不同子树内的物体不可能相交不用去计算, 在同一个子树由对应的算法再计算出接触点等信息。因为这是远距碰撞检测，就叫Broad Phase, 接下来还有Narrow Phase。)

#### 约束(constraint)

约束(constraint)就是消除物体自由度的物理连接。一个2D物体有3个自由度（两个平移坐标和一个旋转坐标）。如果我们把一个物体钉在墙上(像摆锤那样)，那我们就把它约束到了墙上。这样，此物体就只能绕着这个钉子旋转，因此这个约束消除了它 2 个自由度。

#### 接触约束(contact constraint)

一种防止刚体穿透，并模拟摩擦和恢复的特殊约束。你不必创建接触约束，它们会自动被 Box2D 创建。

#### 关节(joint)

它是一种用于把两个或更多的物体固定到一起的约束。Box2D 支持若干种关节类型: 旋转、棱柱、距离等等。有些关节拥有限制(limits)和马达(motors)。

#### 关节限制(joint limit)

关节限制限定了关节的运动范围。例如，人类的胳膊肘只能做某一范围角度的运动。

#### 关节马达(joint motor)

关节马达能依照关节的自由度来驱动所连接的物体。例如，你可以使用马达来驱动胳膊肘的

旋转。

#### 世界(world)

物理世界就是相互作用的物体，夹具和约束的集合。Box2D 支持创建多个世界，但这通常是不必要或不推荐的。

#### 求解器(solver)

物理世界使用求解器来推算时间，求解接触和关节约束。Box2D的求解器是一种高性能的迭代求解器，它会顺序执行N次，这里的N是约束的个数。

(译注:即算法的复杂度为O（N）。)

#### 连续碰撞(continuous collision)

求解器使用时域上的离散时间步来推算物体状态。如果没有特殊处理的话，这会导致隧穿效应。

(译注：假设我们采用1s的固定时间间隔来推算一个物理系统的运动。那么如果这个系统中有两个物体在某一秒的0.5s的时刻，发生碰撞的话。死板的采用固定时间间隔计算的方法，就会导致物体实际上越过了碰撞点的现象发生，这就是隧穿效应。解决的办法显然是要估算出碰撞发生的时刻，并做相应的处理，这也是下一段提到的TOI的含义。)



Box2D拥有特殊的算法来处理隧穿效应。首先，碰撞算法能够在两个物体的运动过程中进行插值运算，以找到首次碰撞时间 (the first time of impact,TOI)。接着，一个分步求解器将物体移动到它们的TOI时刻，并对碰撞求解。

## 模块

Box2D由三个模块组成: 通用模块(Common)，碰撞模块(Collision) 和力学模块(Dynamics). 通用模块包含了内存分配、数学和配置的代码。碰撞模块定义形状、碰撞检测和碰撞的函数或队列。最终力学模块提供对世界、物体、夹具和关节的模拟。



## 单位

Box2D使用浮点数，所以必须使用公差来保证它正常工作。这些公差已经被调谐得适合米-千克-秒(MKS)单位制。尤其是，Box2D已被调谐得能良好地处理0.1到10米之间的移动物体。这意味着从罐头盒到公共汽车大小的对象都能良好地工作。静态的物体就算大到50米都没有问题。

作为一个2D物理引擎，使用像素作为单位是很诱人的。但很不幸，那将导致不良的模拟，也可能会造成古怪的行为。一个200像素长的物体在Box2D看来就有45层建筑那么大。

**注意**

Box2D 已被调谐至 MKS 单位。移动物体的尺寸应该保持在大约 0.1 到 10 米之间。当你渲染场景和角色时, 可能要用到一些比例缩放系统。Box2D自带的testbed例子，使用了OpenGL的视口变换。不要使用像素！！！

最好把Box2D中的物体看作是被贴上了你的艺术创作品的移动广告板。这个广告板在一个以米为单位的系统里运动，但你可以利用简单的比例因子把它转换为像素坐标。之后就可以使用这些像素坐标去确定你的精灵(sprites)的位置，等等。你也可以将它的坐标轴翻转过来。

(译注:坐标轴翻转的含义是比例因子可以为负数。)

Box2D里的角使用弧度制。物体的旋转角度以弧度方式存储，并可以无限增大。如角度变得太大，可考虑将角度进行规范化。（使用b2Body::SetAngle）

**注意**

Box2D使用弧度，而不是度。

## 工厂和定义

快速内存管理在 Box2D API 的设计中担当了一个中心角色。所以当你创建一个 b2Body 或一个 b2Joint时，你需要调用 b2World 的工厂函数(factory functions)。你不应以别的方式为这些类型分配内存。

这些是创建函数:

b2Body\* b2World::CreateBody(const b2BodyDef\* def)

b2Joint\* b2World::CreateJoint(const b2JointDef\* def)

这些是对应的销毁函数:

void b2World::DestroyBody(b2Body\* body)

void b2World::DestroyJoint(b2Joint\* joint)

当你创建物体或关节时，需要提供定义(definition)。这些定义包含了创建物体或关节时所需的所有信息。使用这样的方法，我们能够预防构造错误，保持较少的函数参数数量，提供有意义的默认值，并减少访问子(accessor)的个数。

因为fixture必须有父body，所以要使用b2Body的工厂方法来创建并销毁它们。

b2Fixture\* b2Body::CreateFixture(const b2FixtureDef\* def)

void b2Body::DestroyFixture(b2Fixture\* fixture)

也有个简便的方法直接用形状和密度来创建fixture。

b2Fixture\* b2Body::CreateFixture(const b2Shape\* shape, float32 density)

工厂并不保留定义的引用，因此你可以在栈上创建定义，并在临时资源中保存它们。

# Hello Box2D

Box2D的发布包中有个Hello World程序。程序创建了一个大大的地面盒(ground box)和一个小小的动态盒(dynamic box)。代码没有涉及到图形界面，你只能在控制台中看到随时间变化的盒子位置的文字输出。

这是个很好的例子, 展示了怎么学习和使用Box2D。

## 创建世界

每个Box2D程序开始时都会创建一个b2World对象。b2World是个物理枢纽(physics hub)，用于管理内存、对象和模拟。你可以在栈, 堆或数据区中创建出world。

创建Box2D的world很简单。首先，我们定义重力矢量。

b2Vec2 gravity(0.0f, -10.0f);

现在可以创建world对象了。注意，我们是在栈中创建world, 所以world不能离开它的作用域。

b2World world(gravity);

现在我们已经有了自己的物理世界，开始向里面加东西了。

## 创建地面盒

body用以下步骤来创建：

1. 用位置(position), 阻尼(damping)等来定义body。
2. 用world对象来创建body。
3. 用形状(shape), 摩擦(friction), 密度(density)等来定义fixture。
4. 在body上来创建fixture。

第一步，创建ground body。为此我们需要一个body定义。在定义中，我们指定ground body的初始位置。

b2BodyDef groundBodyDef;

groundBodyDef.position.Set(0.0f, -10.0f);

第二步，将body定义传给world对象，用以创建ground body。world对象并不保留body定义的引用。body默认是静态的。静态物体和其它静态物体之间并没有碰撞，它们是固定的。

b2Body\* groundBody = world.CreateBody(&groundBodyDef);

第三步，创建地面多边形。我们用简便函数SetAsBox使得地面多边形构成一个盒子形状，盒子的中心点就是父body的原点。

b2PolygonShape groundBox;

groundBox.SetAsBox(50.0f, 10.0f);

SetAsBox函数接收半个宽度和半个高度作为参数。因此在这种情况下，地面盒就是100个单位宽(x轴),20个单位高(y轴)。Box2D已被调谐 到使用米，千克和秒做单位。你可以认为长度单位就是米。当物体的大小跟真实世界一样时，Box2D通常工作良好。例如，一个桶约1米高。由于浮点算法的局 限性，使用Box2D模拟冰川或沙尘的运动并不是一个好主意。

第四步，我们创建shape fixture，以完成ground body。在这步中，我们有个简便方法。我们并不需要修改fixture默认的材质属性，可以直接将形状传给body而不需要创建fixture的定义。随后的教程中，我们将会看到如何使用fixture定义来定制材质属性。第二个参数是形状密度，单位是千克/平方米。静态物体的质量定义为0，因此密度对它们是没有用的。

groundBody->CreateFixture(&groundBox, 0.0f);

Box2D并不保存shape的引用。它把数据复制到一个新的b2Shape对象中。

注意，每个fixture都必须有一个父body，即使fixture是静态的。然而，你可以把所有的静态fixture都依附在单个静态body之上。

当你使用fixture 向body添加shape的时候，shape的坐标对于body来说就变成本地的了。因此当body移动的时候，shape也一起移动。fixture的世界变换继承自它的父body。fixture没有独立于body的变换。所以我们不需要移动body上的shape。不支持移动或修改body上的shape。原因很简单：形状发生改变的物体不是刚体，而Box2D只是个刚体引擎。Box2D所做的很多假设都是基于刚体模型的。如果这一条被改变的话，很多事情都会出错。

## 创建动态物体

现在我们已经有了一个地面body，我们可以使用同样的方法来创建一个动态body。除尺寸之外的主要区别是，我们必须为动态body设置质量属性。

首先我们用CreateBody创建body。默认情况下，body是静态的，所以在构造时候应该设置b2BodyType，使得body成为动态的。

b2BodyDef bodyDef;

bodyDef.type = b2\_dynamicBody;

bodyDef.position.Set(0.0f, 4.0f);

b2Body\* body = world.CreateBody(&bodyDef);

注意

如果你想让body受力的影响而运动, 你必须将body的类型设为b2\_dynamicBody。

然后，我们创建一个多边形shape, 并将它附加到fixture定义上。我们先创建一个box shape：

b2PolygonShape dynamicBox;

dynamicBox.SetAsBox(1.0f, 1.0f);

接下来，我们使用box创建一个fixture定义。注意, 我们把密度值设置为1，而密度值默认是0。并且，shape的摩擦系数设置为0.3。

b2FixtureDef fixtureDef;

fixtureDef.shape = &dynamicBox;

fixtureDef.density = 1.0f;

fixtureDef.friction = 0.3f;

注意

一个动态body至少有一个密度不为0的fixture。否则会出现一些奇怪的行为。

使用fixture定义，我们现在就可以创建fixture。这会自动更新body的质量。要是你喜欢，你可以为body添加许多不同的fixture。每个fixture都会增加物体的总质量。

body->CreateFixture(&fixtureDef);

这就是初始化过程。现在我们已经做好准备，可以开始模拟了。

## 模拟(Box2D的)世界

我们已经初始化了地面box和一个动态box。该让牛顿来接手了。我们只有少数几个问题需要考虑。

Box2D使用了一种名叫积分器(integrator)的数值算法。 积分器在离散的时间点上模拟物理方程。 它与传统的游戏动画循环一同运行。我们需要为Box2D选取一个时间步（time step）。通常来说用于游戏的物理引擎需要至少 60Hz 的速度，也就是 1/60 秒的时间步。你可以使用更大的时间步，但是你必须更加小心地为你的世界调整定义。我们也不喜欢时间步变化得太大，一个变化的时间步会导致变化的结果，这会给调试带来困难。所以不要把时间步关联到帧频(除非你真的必须这样做)。直截了当地,这个就是时间步。

float32 timeStep = 1.0f / 60.0f;

除积分器外,Box2D代码还使用了约束求解器(constraint solver)。约束求解器用于解决模拟中的所有约束，一次一个。单个的约束会被完美的求解，然而当我们求解一个约束的时候，我们就会稍微干扰另一个约束。要得到良好的解，我们需要多次迭代所有约束。

约束求解有两个阶段：速度阶段和位置阶段。在速度阶段，求解器会计算必要的冲量，使得物体正确运动。而在位置阶段，求解器会调整物体的位置，减少物体之间的重叠和关节的脱节。每个阶段都有自己的迭代计数。此外，如果误差已足够小的话，位置阶段的迭代可能会提前退出。

Box2D建议的迭代次数，对于速度是8次，对于位置是3次。你可以按自己的喜好去调整这个数字，但要记得它是性能与精度之间的折中。更少的迭代会增加性能但降低精度，同样地，更多的迭代会降低性能但能提高模拟的质量。对于这个简单示例，我们不需要很多的迭代。这里是我们选择的迭代次数。

int32 velocityIterations = 6;

int32 positionIterations = 2;

注意，时间步和迭代数是完全无关的。一个迭代并不是一个子步。一次迭代就是在时间步之中的单次遍历所有约束，你可以在单个时间步内多次遍历约束。

现在我们可以开始模拟循环了，在你的游戏中，模拟循环和游戏循环可以合并起来。每次游戏循环你都应该调用b2World::Step，通常调用一次就够了，这取决于帧频以及物理时间步。

这个Hello World程序设计得非常简单，它没有图形输出。代码会打印出动态body的位置以及旋转角。这就是模拟 1 秒钟内 60 个时间步的循环。

for (int32 i = 0; i < 60; ++i)

{

world.Step(timeStep, velocityIterations, positionIterations);

b2Vec2 position = body->GetPosition();

float32 angle = body->GetAngle();

printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);

}

输出展示了box下降并降落到地面的情况。你的输出看起来应当是这样的：

0.00 4.00 0.00

0.00 3.99 0.00

0.00 3.98 0.00

...

0.00 1.25 0.00

0.00 1.13 0.00

0.00 1.01 0.00

## 清理

当world对象超出它的作用域，或通过指针将其 delete 时，分配给body、 fixture和 joint使用的内存都会被释放。这能提升性能，并使你的生活变得更简单。然而，你应该将body、fixture或joint的指针都清零，因为它们已经无效了。

## The Testbed

一旦你征服了 HelloWorld 例子，你应该开始看 Box2D 的 testbed 了。testbed 是个单元测试框架，也是个演示环境，这是它的一些特点：

* 可移动和缩放的摄像机。
* 可用鼠标选中依附在动态物体上的形状。
* 可扩展的测试集。
* 通过图形界面选择测试,调整参数,以及设置调试绘图。
* 暂停和单步模拟。
* 文字渲染。



在 testbed 中有许多 Box2D 的测试用例，以及框架本身的实例。我鼓励你通过研究和修改它来学习 Box2D。

注意:testbed 是使用 freeglut 和 GLUI 写成的，testbed 本身并不是 Box2D 库的一部分。Box2D本身并不知道如何渲染。就像 HelloWorld 例子一样，使用 Box2D 并不一定需要渲染。

# 通用模块

## 关于

通用模块包含了配置(Settings)，内存管理(memory management)和矢量数学(vector math)。

## 配置

头文件 b2Settings.h 包含：

* 类型，比如int32和float32
* 常数
* 分配器包装(Allocation wrappers)
* 版本号

### 类型

Box2D定义了不同的类型，比如float32、 int8等，以方便确定结构的大小。

### 常数

Box2D定义了若干常数，这些都记录在b2Settings.h中。通常情况下，你不需要调整这些常数。

Box2D的碰撞计算和物体模拟使用了浮点数学。考虑到有舍入方面的错误，所以要定义一些数值公差的，一些是绝对公差，另一些是相对公差。绝对公差使用MKS单位。

### 分配器包装

配置文件定义了b2Alloc和b2Free，用于大内存的分配。你可以让b2Alloc和b2Free调用你自己的内存管理系统。

### 版本号

b2Version结构保存了当前的版本信息，你可以在运行时(run-time)查询。

## 内存管理

Box2D 设计上的很多决定，都是为了能快速有效地使用内存。在本节我将论述 Box2D 如何及为什么要这样分配内存。

Box2D 倾向于分配大量的小型对象(50-300 字节左右)。在系统的堆(heap)上，通过 malloc 或 new分配内存既低效，又容易产生内存碎片。大多数小型对象的生命期都很短暂，例如触点 (contact)，可能只会维持几个时间步。所以我们需要为这些对象提供一个有效的堆分配器。

Box2D的解决方案是使用名为b2BlockAllocator的小型对象分配器(SOA)。SOA维护了一些不定尺寸并可扩展的内存池。当有内存分配请求时，SOA 会返回一块大小最匹配的内存。当内存块释放之后，它会被回收到池中。这些操作都十分快速，只有很小的堆流量。

因为 Box2D 使用了SOA，所以你永远也不应该去 new 或 malloc 一个body、fixture或joint。你只需分配一个 b2World，它为你提供了创建body、fixture和joint的工厂(factory)。这使得Box2D可以使用 SOA 并且将具体的细节隐藏起来。同样，绝对不要去 delete 或 free 一个body、fixture或joint。

当执行一个时间步的时候，Box2D 会需要一些临时的内存。为此，它使用了一个栈分配器来消除单步的堆分配，这个分配器命名为b2StackAllocator。你不需要关心栈分配器，但对此有所了解还是不错的。

## 数学

Box2D包含了一个简单精细的矢量和矩阵模块，来满足Box2D内部和API接口的需要。所有的类都是公开的，你可以在自己的应用程序中自由使用它们。

数学库保持得尽量简单, 使得Box2D容易移植和维护。

# 碰撞模块

## 关于

碰撞模块包含了形状和操作形状的函数。该模块还包含了动态树(dynamic tree)和broad-phase，用于加快大型系统的碰撞处理速度。

碰撞模块被设计为可用于动态系统之外的地方。例如，你可以将动态树用于你的游戏中，除了物理之外的目的。

然而，Box2D 的主要目标是提供一个刚体物理引擎。因此对于一些应用来说，使用碰撞模块会感觉受到限制。同样的，我也不是很想将之写成文档，并形成API。

## 形状

形状描述了可相互碰撞的几何对象，它的使用独立于物理模拟。最起码，你应该知道如何创建shape，并将之附加到刚体上。

b2Shape是个基类，Box2D的各种形状都实现了这个基类。此基类定义了几个函数：

* 判断一个点与形状是否有重叠。
* 在形状上执行光线投射(ray cast)。
* 计算形状的AABB。
* 计算形状的质量。

另外,，每个形状都有成员变量：类型(type)和半径(radius)。 对于多边形，半径也是有意义的，下面会进行讨论。

需要注意的是shape并不知道body，也与力学系统无关。Shape采用一种紧凑格式来进行存储，这种格式经过尺寸和性能的优化。因此，shape并不方便移动，你必须通过手动的设置形状顶点来移动shape。然而，当使用fixture将shape添加到body上之后，shape就会和他的宿主body一起移动。总之：

* 当一个shape**没有**添加到body上时，它的顶点用世界坐标系来表示。
* 当一个shape添加到body上时，它的顶点用局部坐标系来表示。

### 圆形

圆形有位置和半径。圆形是实心的，你没有办法使圆形变成空心。

b2CircleShape circle;

circle.m\_p.Set(2.0f, 3.0f);

circle.m\_radius = 0.5f;

### 多边形

Box2D的多边形是实心的凸(Convex)多边形。在多边形内部任意选择两点，作一线段，如果所有的线段跟多边形的边都不相交，这个多边形就是凸多边形。多边形是实心的，而不是空心的。一个多边形必须有3个或以上的顶点。



多边形的顶点以逆时针（counter clockwise winding，CCW）的顺序存储。我们必须很小心，逆时针是相对于右手坐标系统来说的，这坐标系下，Z轴指向平面外面。有可能相对于你的屏幕，就变成顺时针了，这取决于你自己的坐标系统是怎么规定的。



多边形的成员变量具有public访问权限，但是你应该使用初始化函数来创建多边形。初始化函数会创建法向量(normal vectors)，并检查参数的合法性。

创建多边形时，你可以传递一个包含顶点的数组。数组大小最多是b2\_maxPolygonVertices，这数值默认是8。这已足够描述大多数的凸多边形了。

b2PolygonShape::Set 函数会自动计算凸包（convex hull），并建立时针序。这个函数在顶点数少的时候，是非常快的。但如果你增大b2\_maxPolygonVertices ，凸包的计算就会变慢。此外，凸包函数会消除你提供的顶点，或者对其重排序。距离小于b2\_linearSlop 的顶点会被合并。

// This defines a triangle in CCW order.

b2Vec2 vertices[3];

vertices[0].Set(0.0f, 0.0f);

vertices[1].Set(1.0f, 0.0f);

vertices[2].Set(0.0f, 1.0f);

int32 count = 3;

b2PolygonShape polygon;

polygon.Set(vertices, count);

多边形有一些方便的函数来创建box。

void SetAsBox(float32 hx, float32 hy);

void SetAsBox(float32 hx, float32 hy, const b2Vec2& center, float32 angle);

多边形从b2Shape中继承了半径。通过半径，在多边形的周围创建了一个保护层(skin)。堆叠的情况下，此保护层让多边形之间保持稍微分开。这使得可以在核心多边形上执行连续碰撞。



多边形保护层通过保持多边形的分离来防止隧穿效应。这会导致形状之间有小空隙。你的显示可以比多边形大些，来隐藏这些空隙。



### 边框形状（Edge shapes）

边框形状由一些线段组成。它们可辅助为你的游戏创建一个形状自由的静态环境。边框形状的主要限制在于它们能够与圆形和多边形碰撞，但它们之间却不会碰撞。Box2D使用的碰撞算法要求两个碰撞物体中至少有一个有体积。边框形状没有体积。所以边框形状之间的碰撞是不可能的。

// This an edge shape.

b2Vec2 v1(0.0f, 0.0f);

b2Vec2 v2(1.0f, 0.0f);

b2EdgeShape edge;

edge.Set(v1, v2);

在很多情况下，游戏环境是若干个边框形状首尾相连构成的。当多边形沿着边框链滑动的时候，会导致一个异常的行为。在下图中，我们可以看到一个box和一个内部顶点之间的碰撞。当多边形和一个内部顶点碰撞时，会产生一个内部碰撞法线，这会导致“幽灵”碰撞现象。



如果edge1不存在的话，这个碰撞看起来还算正常。edge1存在的话，这个内部碰撞就像是Bug了。但是通常当Box2D处理两个形状的碰撞时，会单独处理他们的。

幸运的是，边框形状提供了一种机制来消除幽灵碰撞——存储用于调整的幽灵顶点。Box2D使用这些幽灵顶点来阻止内部碰撞。



// This is an edge shape with ghost vertices.

b2Vec2 v0(1.7f, 0.0f);

b2Vec2 v1(1.0f, 0.25f);

b2Vec2 v2(0.0f, 0.0f);

b2Vec2 v3(-1.7f, 0.4f);

b2EdgeShape edge;

edge.Set(v1, v2);

edge.m\_hasVertex0 = true;

edge.m\_hasVertex3 = true;

edge.m\_vertex0 = v0;

edge.m\_vertex3 = v3;

通常情况下，用这种方式将边框缝合到一起是有些浪费和无聊的。这就为我们引入了链接形状（chain shapes）的概念。

### 链接形状（Chain Shapes）

链接形状提供了一种有效的方式，来将许多边框连接在一起，用以构建你的静态游戏世界。链接形状自动消除幽灵碰撞，并提供两侧的碰撞。



// This a chain shape with isolated vertices

b2Vec2 vs[4];

vs[0].Set(1.7f, 0.0f);

vs[1].Set(1.0f, 0.25f);

vs[2].Set(0.0f, 0.0f);

vs[3].Set(-1.7f, 0.4f);

b2ChainShape chain;

chain.CreateChain(vs, 4);

你可能会有一个滚动的游戏世界，需要将若干个链接连接到一起。你可以用幽灵顶点来连接链接，就像我们在b2EdgeShape 中所做的那样。

// Install ghost vertices

chain.SetPrevVertex(b2Vec2(3.0f, 1.0f));

chain.SetNextVertex(b2Vec2(-2.0f, 0.0f));

你也可以自动创建一个环。

// Create a loop. The first and last vertices are connected.

b2ChainShape chain;

chain.CreateLoop(vs, 4);

不支持自相交的链接形状。它可能正常工作，也可能不会。防止幽灵碰撞的代码假定链接中没有自相交存在。同样的，非常接近的顶点也会导致问题。需要确保你的所有边框都比b2\_linearSlop (5mm)长。



链接中的每个边框都被认为是一个子形状，并能用索引来访问。当一个链接形状被连到body上时，每个边框在碰撞检测树上都会有它自己的包围盒（bounding box）。

// Visit each child edge.

for (int32 i = 0; i < chain.GetChildCount(); ++i)

{

b2EdgeShape edge;

chain.GetChildEdge(&edge, i);

…

}

## 单元几何查询（Unary Geometric Queries）

你可以在一个单独的形状上，执行一系列的几何查询。

### 形状点测试（Shape Point Test）

你可以测试一个点是否与形状有所重叠。你需要提供一个形状的变换以及世界坐标上的一个点。

b2Transfrom transform;

transform.SetIdentity();

b2Vec2 point(5.0f, 2.0f);

bool hit = shape->TestPoint(transform, point);

边框和链接形状总是返回false，即使链接是一个环。

### 形状的光线投射(Shape Ray Cast)

你可以用光线射向形状，得到它们之间的第一个交点和法向量。如果在形状内部开始投射，就当成没有交点。链接形状包含儿子索引，是因为光线投射一次只会检测一个边框。

b2Transfrom transform;

transform.SetIdentity();

b2RayCastInput input;

input.p1.Set(0.0f, 0.0f, 0.0f);

input.p2.Set(1.0f, 0.0f, 0.0f);

input.maxFraction = 1.0f;

int32 childIndex = 0;

b2RayCastOutput output;

bool hit = shape->RayCast(&output, input, transform, childIndex);

if (hit)

{

b2Vec2 hitPoint = input.p1 + output.fraction \* (input.p2 – input.p1);

…

}

## 对等函数

碰撞模块包含一些对等函数，它们接受一对形状参数，并计算出结果。这些函数包括：

* 重叠
* 接触形
* 距离
* 撞击时间

### 重叠

你可以用这个函数测试两个形状是否重叠。

b2Transform xfA = …, xfB = …;

bool overlap = b2TestOverlap(shapeA, indexA, shapeB, indexB, xfA, xfB);

如果是链接形状的话，你还必须提供子形状的索引。

### 接触形（Contact Manifolds）

Box2D有一些用来计算重合形状之间的接触点的函数。考虑一下圆与圆，圆与多边形的碰撞，我们只会得到一个接触点和一个向量。多边形与多边形的碰撞，我们可 以得到两个接触点。这些接触点具有相同的法向量，所以Box2D将它们归成一组，构成manifold结构。接触求解器将利用这个结构，以改善堆叠的稳定性。



通常你不需要直接计算接触形，但你可能会使用在模拟过程中已处理好的结果。

b2Manifold结构含有一个法向量和最多两个接触点。向量和接触点都是相对于局部坐标系。为方便接触求解器处理，每个接触点都存储了法向冲量和切向(摩擦）冲量。

存储在b2WorldManifold结构中的数据为内部使用做了优化。如果你需要这些数据，最好的方法是使用b2WorldManifold结构生成世界坐标下的接触向量和点。你需要提供b2Manifold结构和形状的转换及半径。

b2WorldManifold worldManifold;

worldManifold.Initialize(&manifold, transformA, shapeA.m\_radius,

transformB, shapeB.m\_radius);

for (int32 i = 0; i < manifold.pointCount; ++i)

{

b2Vec2 point = worldManifold.points[i];

…

}

注意worldmanifold使用的点数量来自于manifold.

模拟过程中，形状会移动而manifold可能会改变。接触点有可能会添加或移除。你可以使用b2GetPointStates来检查状态。

b2PointState state1[2], state2[2];

b2GetPointStates(state1, state2, &manifold1, &manifold2);

if (state1[0] == b2\_removeState)

{

// process event

}

### 距离

b2Distance函数可以用来计算两个形状之间的距离。距离函数需要两个形状都被转成b2DistanceProxy。重复调用距离函数的时候，Box2D会使用缓冲的方式使之热启动。你可以在b2Distance.h 中看到实现的细节。

(译注：热启动是相对于冷启动而言的。一般情况下，机器热启动时，由于初始化的东西较少，而具有更快的速度。这里将机器热启动的概念，拓展推广到函数热启动，含义都是类似的。)



### 撞击时间

如果两个形状快速移动，它们可能会在一个时间步内穿过对方。



b2TimeOfImpact函数用于确定两个形状运动时碰撞的时间。这称为撞击时间(time of impact, TOI)。b2TimeOfImpact的主要目的是防止隧穿效应。特别是，它设计来防止运动的物体隧穿过静态的几何形状。

这个函数考虑了形状的旋转和平移，但如果旋转足够大，这函数还是会错过碰撞。函数仍然会报告一个非重叠的时间，并捕捉到所有的平移碰撞。

撞击时间函数定义了一条初始的分离轴，并确保形状没有越过这条轴。这可能会在最终位置错过一些碰撞。尽管如此，这种方法在防止隧穿方面已经快速并足够适用了。





很难去限定旋转角的范围，有些情况下，就算是很小的旋转角也会导致错过碰撞。通常，就算错过了一些碰撞，也不会影响到游戏的好玩性。游戏往往会忽略这些碰撞。

这函数需要两个形状(转换成b2DistanceProxy)和两个b2Sweep结构。b2Sweep结构定义了形状的开始和结束时的转换。

你可以在固定旋转角的情况下去执行这个计算撞击时间的函数，这样就不会错过任何碰撞。

## 动态树

Box2D使用b2DynamicTree来高效地组织大量的形状。这个类并不知道形状的存在。取而代之，它通过用户数据指针来操作轴对齐包围框(AABB)。

动态树是分层的AABB树。树的每个内部节点都有两个子节点。叶子节点是用户的AABB。这个树使用旋转来保持树的平衡，即使是在退化的输入（degenerate input）的情况下。

(译注：degenerate input在算法上是指最坏情况的输入。例如要对一个数组进行升序排列。通常情况下的输入，一般是随机的数字序列。而一个降序的数字序列在这里就算是degenerate input了。)

这种树结构支持高效的光线投射(ray casts)和区域查询(region queries)。比如，场景中有数百个形状，你想对场景执行光线投射，如果采用蛮力，就需要对每个形状都进行投射。这是很低效的，并没有利用到形状的分布信息。替代方法是，你维护一棵动态树，并对树进行光线投射。在遍历树的时候，可以跳过大量的形状。

区域查询使用树来查找跟需查询的AABB有重叠的所有叶节点。这比蛮力算法高效得多，因为很多形状会被直接跳过。





通常你并不会直接用到动态树。你会通过b2World类来执行光线投射和区域查询。如果你想创建自己的动态树，你可以去看看Box2D是怎么使用动态树的。

## Broad-phase

物理步内的碰撞处理可以分成两个阶段: narrow-phase和broad-phase。narrow-phase时，我们去计算两个形状之间的接触点。假设有N个形状，使用蛮力算法的话，就需要执行 N\*N/2次narrow-phase。

Tb2BroadPhase类使用了动态树来减少管理数据方面的开销。这可以大幅度减少narrow-phase的调用次数。

通常你不会直接和broad-phase交互。Box2D自己会在内部创建并管理broad-phase。另外要注意，b2BroadPhase是设计用于Box2D中的物理模拟，它可能不适合处理其它情况。

# 力学模块

## 概述

力学模块是Box2D中最复杂的部分，也是与你交互最多的部分。力学模块构建在通用和碰撞模块的基础上，到现在你对这两个模块也应该有所了解了。

力学模块包括下面这些类：

* 夹具
* 刚体
* 接触
* 关节
* 世界
* 监听者

这些类相互依赖，很难在不提及其它类的情况下单独描述一个类。在接下来的章节中，你会看到一些类是之前没有提及过的。你可以先快速浏览一下对应的章节，之后才去细读。

力学模块包括接下来的章节。

# 物体

## 关于

物体具有位置和速度。你可以将力(forces)、扭矩(torques)、冲量(impulses)应用到物体上。 物体可以是静态的(static)、运动但不受力的(kinematic)或动态的(dynamic)。这是物体的类型定义：

#### b2\_staticBody

static物体在模拟时不会运动，就好像它具有无穷大的质量。在Box2D内部，会将static物体的质量和质量的倒数存储为零。static物体可以让用户手动移动。它的速度为零，另外也不会和其它static或kinematic物体相互碰撞。

#### b2\_kinematicBody

kinematic物体在模拟时以一定的速度运动，但不受力的作用。它们可以让用户手动移动，但通常的做法是设置一定的速度来移动它。kinematic物体的行为表现就好像它具有无穷大的质量，Box2D将它的质量和质量的倒数存储为零。

#### b2\_dynamicBody

dynamic物体被完全模拟。它们可以让用户手动移动，但通常它们都是受力的作用而运动。dynamic物体可以和其它所有类型的物体相互碰撞。dynamic物体的质量总是有限大的，非零的。如果你试图将它的质量设置为零，它会自动地将质量修改成一千克，并且它不会转动。

物体是fixtures的骨架，带着fixture在世界中运动。Box2D中的物体总是刚体(rigid body)。也就是说，同一物体上的两个fixture，永远不会相对移动，也不会碰撞。

fixture有可碰撞的几何形状和密度(density)。物体通常从它的fixture中获得质量属性。当物体构建之后，你也可以改写它的质量属性。

通常你会保存所有你所创建物体的指针，这样你就能查询物体的位置，用于更新图形实体的位置。另外在不需要它们的时候，你也可以使用指针去摧毁它们。

## 物体定义

在创建物体之前你需要先创建物体定义(b2BodyDef)。物体定义含有创建并初始化物体所需的数据。

Box2D会从物体定义中复制数据，并不会保存它的指针。这意味着你可以重复使用同一个物体定义去创建多个物体。

让我们看一些物体定义的关键成员。

### 物体类型

本章开始已经说过，有三种物体类型: static、kinematic和dynamic。 你应该在创建时就确定好物体类型，因为以后再修改的话，代价会很高。

bodyDef.type = b2\_dynamicBody;

物体类型是一定要设置的。

### 位置和角度

物体定义为你提供了一个在创建时初始化位置的机会。这比在world原点下创建物体后再移动到某个位置更高效。

**注意**

不要在原点创建物体后再移动它。如果你在原点上同时创建了几个物体，性能会很差。

物体上主要有两个让人感兴趣的点。第一个是物体的原点。fixture和关节都是相对于原点而依附到物体上面的。第二个是物体的质心。质心由形状的质量分布决定，或显式地由b2MassData设置。Box2D内部许多计算都要使用物体的质心, 例如b2Body会存储质心的线速度。

当你构造物体定义的时候,可能你并不知道质心在哪里。你可以指定物体的原点，也可以以弧度指定物体的角度，角度并不受质心位置的影响。如果随后你改变了物体的质量属性，那么质心也会随之移动，但是原点不会改变，物体上的形状和关节也不会移动。

bodyDef.position.Set(0.0f, 2.0f); // the body's origin position.

bodyDef.angle = 0.25f \* b2\_pi; // the body's angle in radians.

刚体也是个参考框架。你可以在这个框架内定义fixture和joint。fixture和joint的锚点不会在框架内移动。

### 阻尼

阻尼用于减小物体在世界中的速度。阻尼跟摩擦有所不同，摩擦仅在物体有接触的时候才会发生。阻尼并不能取代摩擦，往往这两个效果需要同时使用。

阻尼参数的范围可以在0到无穷大之间，0表示没有阻尼，无穷大表示满阻尼。通常来说，阻尼的值应 该在0到0.1之间。通常我不使用线性阻尼， 因为它会使物体看起来有点漂浮。

bodyDef.linearDamping = 0.0f;

bodyDef.angularDamping = 0.01f;

阻尼类似稳定性与性能， 在值较小的时候阻尼效应几乎不依赖于时间步，值较大的时候阻尼效应将随着时间步而变化。如果你使用固定的时间步(推荐)这就不是问题了。

### 重力因子

你可以使用重力因子来调整单个物体上的重力。这需要足够的细心，增加的重力会降低稳定性。

// Set the gravity scale to zero so this body will float

bodyDef.gravityScale = 0.0f;

### 休眠参数

休眠是什么意思？模拟物体的成本是高昂的，所以物体越少，那模拟的效果就越好。当物体停止了运动时，我们会希望停止模拟它。

当Box2D确定一个物体(或一组物体)已停止移动时，物体就会进入休眠状态。休眠物体只消耗很小的 CPU开销。如果一个醒着的物体接触到了一个休眠中的物体，那么休眠中的物体就会醒过来。当物体上的关节或触点被摧毁的时候，它们同样会醒过来。你也可以手动地唤醒物体。

通过物体定义,你可以指定一个物体是否可以休眠，或者创建一个休眠的物体。

bodyDef.allowSleep = true;

bodyDef.awake = true;

### 固定旋转

你可能想让一个刚体，比如某个角色，具有固定的旋转角。这样物体即使在负载下，也不会旋转。 你可以设置fixedRotation来达到这个目的：

bodyDef.fixedRotation = true;

固定旋转标记使得转动惯量和它的倒数被设置成零。

### 子弹

游戏模拟通常以一定帧率(frame rate)产生一系列的图片。这就是所谓的离散模拟。在离散模拟中，在一个时间步内刚体可能移动较大距离。如果一个物理引擎没有处理好大幅度的运动，你就可能会看见一些物体错误地穿过了彼此。这被称为隧穿效应(tunneling)。

默认情况下，Box2D会通过连续碰撞检测(CCD)来防止动态物体穿越静态物体。这是通过扫描形状从旧位置到新位置的过程来完成的。引擎会查找扫描中的新碰撞，并为这些碰撞计算碰撞时间 (TOI)。物体会先被移动到它们的第一个TOI，然后求解器执行一个子步（sub-step）计算以完成整个时间步的计算。在子步中，可能还有更多的TOI事件发生。

一般情况下，dynamic物体之间不会应用CCD，这是为了保持合理的性能。在一些游戏场景中，你需要在动态物体上也使用CCD。比如，你可能想用一颗高速的子弹去射击一块动态的砖头。没有CCD，子弹就可能会隧穿砖头。

在Box2D中，高速移动的物体可以标记成子弹(bullet)。子弹跟static或者dynamic物体之间都会执行CCD。你需要按照游戏的设计来决定哪些物体是子弹。如果你决定一个物体应该按照子弹去处理，可使用下面的设置。

bodyDef.bullet = true;

子弹标记只影响dynamic物体。

### 活动状态

你可能希望创建一个物体并不参与碰撞和动态模拟。这状态跟休眠有点类似，但并不会被其它物体唤醒，它上面的fixture也不会 被放到broad-phase中。也就是说，物体不会参于碰撞检测，光线投射(ray casts)等等。

你可以创建一个非活动的物体，之后再激活它。

bodyDef.active = true;

关节也可以连接到非活动的物体。但这些关节并不会被模拟。你要小心，当激活物体时，它的关节不会被扭曲(distorted)。

注意，激活一个物体和重新创建一个物体的开销差不多。因此你不应该在流世界（streaming worlds）中使用激活，而应该用创建和销毁来节省内存。

(译注：streaming worlds是指该世界中的大多数物体是动态创建的，而不是一开始就有的。)

### 用户数据

用户数据是个void指针。它让你将物体和你的应用程序关联起来。你应该保持一致性，所有物体的用户数据都指向相同的对象类型。

b2BodyDef bodyDef;

bodyDef.userData = &myActor;

## 物体工厂（Body Factory）

world类提供物体工厂来创建和摧毁物体。这让world可以通过一个高效的分配器来创建物体，并且把物体添加到world的数据结构中。

b2Body\* dynamicBody = myWorld->CreateBody(&bodyDef);

... do stuff ...

myWorld->DestroyBody(dynamicBody);

dynamicBody = NULL;

**注意**

永远不要使用new或malloc来创建物体，否则世界不会知道这个物体的存在，并且物体也不会被适当地初始化。

Box2D并不保存物体定义的引用，也不保存其任何数据(除了用户数据指针)。所以你可以创建临时的物体定义，并重复利用它。

Box2D允许你通过删除b2World对象来摧毁物体，它会为你做所有的清理工作。然而，你必须小心地将保存在游戏引擎中的body指针清零。

当你摧毁物体时，依附其上的fixture和joint都会自动被摧毁。了解这点，对你如何管理shape 和joint指针有重要意义。

## 使用物体

在创建完一个物体之后，你可以对它进行许多操作。其中包括设置质量属性，访问其位置和速度，施加力，以及转换点和向量。

### 质量数据

每个物体都有质量（标量）、质心（二维向量）和转动惯性（标量）。对于static物体，它的质量和转动惯性都被设为零。当物体设置成固定旋转(fixed rotation)，它的转动惯性也是零。

通常情况下，当fixture添加到物体上时，物体的质量属性会自动地确定。你也可以在运行时(run-time)调整物体的质量。当你有特殊的游戏方案需要改变质量时，可以这样做。

void SetMassData(const b2MassData\* data);

直接设置物体的质量后，你可能希望再次使用fixture所展示的质量。你可以这样做：

void ResetMassData();

要得到物体的质量数据，可以通过下面的函数：

float32 GetMass() const;

float32 GetInertia() const;

const b2Vec2& GetLocalCenter() const;

void GetMassData(b2MassData\* data) const;

### 状态信息

物体的有多个方面状态。你可以通过下面的函数高效地访问状态数据：

void SetType(b2BodyType type);

b2BodyType GetType();

void SetBullet(bool flag);

bool IsBullet() const;

void SetSleepingAllowed(bool flag);

bool IsSleepingAllowed() const;

void SetAwake(bool flag);

bool IsAwake() const;

void SetActive(bool flag);

bool IsActive() const;

void SetFixedRotation(bool flag);

bool IsFixedRotation() const;

### 位置和速度

你可以访问一个物体的位置和旋转角，这在你渲染相关游戏角色时很常用。通常情况下，你都是使用Box2D来模拟运动，但你也可以设置位置，尽管并不怎么常用。

bool SetTransform(const b2Vec2& position, float32 angle);

const b2Transform& GetTransform() const;

const b2Vec2& GetPosition() const;

float32 GetAngle() const;

你可以访问本地坐标系及世界坐标下的质心。许多Box2D的内部模拟都使用质心。然而，通常你并不需要访问质心。取而代之，你一般应该关心物体变换。比如，你有个正方形的物体。物体的原点可能在正方形的一个角点，而质心却位于正方形的中心点。

const b2Vec2& GetWorldCenter() const;

const b2Vec2& GetLocalCenter() const;

你可以访问线速度和角速度。线速度是对于质心所言的。所以质量属性改变了，线速度有可能也会改变。

# 夹具

## 关于

回想一下，形状不知道物体的存在，并可独立于物理模拟而被使用。因此Box2D提供b2Fixture类，用于将形状附加到物体上。一个物体可以有零个或多个fixture。拥有多个fixture的物体有时被叫做组合物体。

fixture具有下列属性：

* 关联的形状
* broad-phase代理
* 密度(density)、摩擦(friction)和恢复(restitution)
* 碰撞筛选标记(collision filtering flags)
* 指向父物体的指针
* 用户数据
* 传感器标记(sensor flag)

这些都会在接下来的小节中描述。

## 创建夹具

要创建fixture，先要创始化一个fixture定义，并将定义传到父物体中。

b2FixtureDef fixtureDef;

fixtureDef.shape = &myShape;

fixtureDef.density = 1.0f;

b2Fixture\* myFixture = myBody->CreateFixture(&fixtureDef);

这会创建fixture，并将它附加到物体之上。你不需要保存fixture的指针，因为当它的父物体被摧毁时，fixture也会自动被摧毁。 你可以在单个物体上创建多个fixture。

你可以摧毁父物体上的fixture，来模拟一个可分裂开的物体。你也可以不理会fixture，让物体的析构函数来摧毁附加其上的fixture。

myBody->DestroyFixture(myFixture);

### 密度

fixture的密度用来计算父物体的质量属性。密度值可以为零或者是正数。你所有的fixture都应该使用相似的密度，这样做可以改善堆叠稳定性。

当设置密度的时候，物体的质量不会立即改变。你必须调用ResetMassData ，使之生效。

fixture->SetDensity(5.0f);

body->ResetMassData();

### 摩擦

摩擦可以使对象逼真地沿其它对象滑动。Box2D支持静摩擦和动摩擦，两者都使用相同的参数。摩擦在Box2D中会被精确地模拟，摩擦力的强度与正交力 (称之为库仑摩擦)成正比。摩擦参数通常会设置在0到1之间，但也可是任意的非负数，0意味着没有摩擦，1会产生强摩擦。当计算两个形状之间的摩擦力时，Box2D必须组合两个形状的摩擦参数。这是通过以下公式完成的：

float32 friction;

friction = sqrtf(fixtureA->friction \* fixtureB->friction);

所以当其中一个fixture的摩擦参数为0时，接触的摩擦就为0。

你可以用b2Contact::SetFriction 重载默认的合成摩擦的方法。这通常是在b2ContactListener 的回调中执行的。

### 恢复

恢复可以使对象弹起。恢复的值通常设置在0到1之间。想象一个小球掉落到桌子上，值为0表示着小球不会弹起， 这称为非弹性碰撞。值为1表示小球的速度跟原来一样，只是方向相反，这称为完全弹性碰撞。恢复是通过下面的公式合成的：

float32 restitution;

restitution = b2Max(fixtureA->restitution, fixtureB->restitution);

因为恢复是以这样的方式合成，因此你可以指定一个有弹性的球，而不用指定一个有弹性的地板。

你可以用b2Contact:: SetRestitution 重载默认的合成恢复的方法。这通常是在b2ContactListener 的回调中执行的。

当一个形状多次碰撞时，恢复会被近似地模拟，这是因为Box2D使用了迭代求解器。当冲撞速度很小时，Box2D也会使用非弹性碰撞，这是为了防止抖动。参见b2Settings.h中的b2\_velocityThreshold 。

### 筛选

碰撞筛选是为了防止某些fixture之间发生碰撞。比如,你创造了一个骑自行车的角色。你希望自行车与地形之间有碰撞,角色与地形有碰撞,但你不希望角色和自行车之间发生碰撞 (因为它们必须重叠)。Box2D通过种群和分组来支持这样的碰撞筛选。

Box2D支持16个种群。任意fixture你都可以指定它属于哪个种群。你还可以指定这个fixture可以和其它哪些种群发生碰撞。例如,你可以在一个多人游戏中指定玩家之间不会碰撞，怪物之间也不会碰撞，但是玩家和怪物会发生碰撞。这是通过掩码来完成的，例如：

playerFixtureDef.filter.categoryBits = 0x0002;

monsterFixtureDef.filter.categoryBits = 0x0004;

playerFixtureDef.filter.maskBits = 0x0004;

monsterFixtureDef.filter.maskBits = 0x0002;

下面是产生碰撞的规则：

uint16 catA = fixtureA.filter.categoryBits;

uint16 maskA = fixtureA.filter.maskBits;

uint16 catB = fixtureB.filter.categoryBits;

uint16 maskB = fixtureB.filter.maskBits;

if ((catA & maskB) != 0 && (catB & maskA) != 0)

{

// fixtures can collide

}

碰撞分组让你指定一个整数的组索引。你可以让同一个组的所有fixture总是相互碰撞(正索引)或永远不碰撞(负索引)。组索引通常用于一些以某种方式关联的事物，就像自行车的那些部件。在下面的例子中，fixture1 和 fixture2 总是碰撞，而 fixture3 和 fixture4 永远不会碰撞。

fixture1Def.filter.groupIndex = 2;

fixture2Def.filter.groupIndex = 2;

fixture3Def.filter.groupIndex = -8;

fixture4Def.filter.groupIndex = -8;

如果组索引不同，碰撞筛选就会按照种群和掩码来进行。换句话说，分组筛选与种群筛选相比，具有更高的优选级。

注意在 Box2D 中还有其它的碰撞筛选，这里是一个列表：

* static上的fixture只会与dynamic物体上的fixture发生碰撞。
* kinematic物体 只会和dynamic物体碰撞。
* 同一个物体上的fixture永远不会相互碰撞。
* 如果两个物体用关节连接起来，物体上面的fixture可以选择启用或禁止它们之间相互碰撞。

有时你可能希望在形状创建之后去改变其碰撞筛选。 你可以使用b2Shape::GetFilterData 和b2Shape::SetFilterData来访问和设置已存在的fixture的b2FilterData结构。注意就算修改了筛选数据，到下一个时间步 为止，现在的接触并不会被增加或删除（参见world类）。

## 传感器

有时候游戏逻辑需要判断两个fixture是否相交，而不想有碰撞反应。这可以通过传感器(sensor)来完成。传感器也是个fixture，但它只会侦测碰撞，而不产生其它反应。

你可以将任意fixture标记为传感器。传感器可以是static、kinematic或dynamic的。记住，每个物体上可以有多个fixture，传感器和实体fixture是可以混合存在的。而且，只有至少一个物体是dynamic的，传感器才会产生接触事件，而kinematic 与kinematic 、kinematic 与static ，或者static 与static 之间都不会产生接触事件。

传感器不会生成接触点。这里有两种方法得到传感器的状态：

1. b2Contact::IsTouching
2. b2ContactListener::BeginContact 和 EndContact

# 关节

## 关于

关节用于把物体约束到世界，或约束到其它物体上。在游戏中，典型例子有木偶，跷跷板和滑轮。用不同的方式将关节结合起来使用，可以创造出有趣的运动。

有些关节提供了限制(limit)，使你可以控制运动的范围。有些关节还提供了马达(motor)，它可以以指定的速度驱动关节一直运动，直到你指定了更大的力或扭矩来抵消这种运动。

关节马达有许多不同的用途。你可以使用关节来控制位置，只要提供一个与目标之距离成正比例的关节速度即可。你还可以模拟关节摩擦：将关节速度置零，并且提供一个小的、但有效的最大力或扭矩；那么马达就会努力保持关节不动，直到负载变得过大为止。

## 关节定义

每种关节类型都有各自的定义(definition)，它们都派生自b2JointDef。所有的关节都连接两个不同的物体，其中一个物体有可能是静态的。关节也可以连接两个static或者kinematic类型的物体，但这没有任何实际用途，只会浪费处理器时间。

你可以为任何一种关节类型指定用户数据。你还可以提供一个标记，用于防止用关节相连的物体之间发生碰撞。实际上, 这是默认行为。你也可以通过设置 collideConnected，来允许相连的物体之间发生碰撞。

很多关节定义需要你提供一些几何数据。一个关节常常需要一个锚点(anchor point)来定义，这是固定于相接物体中的点。Box2D要求这些点在局部坐标系中指定，这样，即便当前物体的变化违反了关节约束(joint constraint)，关节还是可以被指定——这通常会发生在游戏保存或载入进度时。另外，有些关节定义需要知道物体之间默认的相对角度。这样才能正确地约束旋转。

初始化几何数据可能有些乏味。所以很多关节提供了初始化函数，使用当前的物体的形状，来消除大部分工作。然而，这些初始化函数通常只应用于原型，在产品代码中应该直接地定义几何数据。这能使关节行为更具健壮性。

其余的关节定义数据依赖于关节的类型。下面我们来介绍它们。

## 关节工厂

关节使用world的工厂方法来创建和摧毁。这引入一个老问题：

**注意**

不要使用new或malloc在栈(stack)或堆(heap)中创建关节。你想创建或摧毁物体和关节，必须使用b2World类中对应的创建或摧毁函数。

这里有个例子，展示了旋转关节(revolute joint)的生命周期：

b2RevoluteJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.anchorPoint = myBodyA->GetCenterPosition();

b2RevoluteJoint\* joint = (b2RevoluteJoint\*)myWorld->CreateJoint(&jointDef);

... do stuff ...

myWorld->DestroyJoint(joint);

joint = NULL;

一个很好的习惯：当对象摧毁后，就将对应的指针清零。这样的话，当你试图再次使用这个指针时，程序会以一种可控的方式崩溃。

(译注：野指针虽然不一定会使程序崩溃，但它所带来的问题非常难以调试。空指针虽然一定会让程序崩溃，但是调试起来很简单。)

关节的生命周期并不简单，要特别留心下面的警告。

**注意**

物体被摧毁时，依附其上的关节也会被摧毁。

上面的注意并非时时必要。你可以组织好自己的游戏引擎，保证物体被摧毁前，依附其上的关节已经先被摧毁。在这种情况下，你没有必要实现监听类 (listener class)，去监听物体被摧毁的事件。更多细节请看隐式摧毁(Implicit Destruction)那小节。

## 使用关节

在许多模拟中，关节被创建之后，直到摧毁也不会再被访问。然而，关节中包含着很多有用的数据，使你可以创建出丰富的模拟。

首先，你可以在关节上得到物体、锚点和用户数据。

b2Body\* GetBodyA();

b2Body\* GetBodyB();

b2Vec2 GetAnchorA();

b2Vec2 GetAnchorB();

void\* GetUserData();

所有的关节都有反作用力和反扭矩，这个反作用力应用于 body2 的锚点之上。你可以用反作用力来折断关节(break joints)，或引发其它游戏事件。这些函数可能需要做一些计算，所以没有必要就不要去调用它们。

b2Vec2 GetReactionForce();

float32 GetReactionTorque();

## 距离关节

距离关节是最简单的关节之一, 它是说, 两个物体上面各自有一点，两点之间的距离必须固定不变。当你指定一个距离关节时，两个物体必须已在应有的位置上。之后，你指定世界坐标中的两个锚点。第一个锚点连接 到物体1，第二个锚点连接到物体2。这两点隐含了距离约束的长度。



这是一个距离关节定义的例子。这种情况下, 我们允许物体碰撞。

b2DistanceJointDef jointDef;

jointDef.Initialize(myBodyA, myBodyB, worldAnchorOnBodyA, worldAnchorOnBodyB);

jointDef.collideConnected = true;

距离关节也可以是软的，就像用橡皮筋来连接。看看testbed中的Web例子，就可以知道它有什么样的行为。

要使关节有弹性，可以调节一下定义中的两个常数：频率(frequency)和阻尼率(damping ratio)。将频率想象成谐振子(harmonic oscillator，比如吉他弦)振动的快慢。频率使用单位赫兹(Hertz)来指定。典型情况下，关节频率要小于时间步(time step)频率的一半。比如每秒执行60次时间步， 距离关节的频率就要小于30赫兹。这样做的理由可以参考Nyquist频率理论。

阻尼率无单位，典型是在0到1之间，也可以更大。1是阻尼率的临界值，当阻尼率为1时，没有振动。

jointDef.frequencyHz = 4.0f;

jointDef.dampingRatio = 0.5f;

## 旋转关节

旋转关节会强制两个物体共享一个锚点，即所谓铰接点。旋转关节只有一个自由度：两个物体的相对旋转。这称之为关节角。



要指定一个旋转关节，你需要提供两个物体以及世界坐标的一个锚点。初始化函数会假定物体已经在正确的位置了。

在此例中，两个物体被旋转关节连接起来，其中铰接点为第一个物体的质心。

b2RevoluteJointDef jointDef;

jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter());

在 body2 逆时针旋转时，关节角为正。像所有 Box2D 中的角度一样，旋转角也是弧度制的。按规定，使用Initialize() 创建关节时，无论两个物体当前的角度怎样，旋转关节角都为0。

有时候，你可能需要控制关节角。为此，旋转关节可以随意地模拟关节限制和马达。

关节限制(joint limit)会强制关节角度保持在一定范围内，它会应用足够的扭矩来保持这个范围。0应该在范围内，否则在开始模拟时关节会有点倾斜。

关节马达允许你指定关节的角速度(角度的时间导数)，速度可正可负。马达可以有产生无限大的力，但这通常是没有必要的。想想那个经典问题：

"当一个不可抵抗的力作用在一个不可移动的物体上，会发生什么?"

我可以告诉你这并不有趣。所以你应该为关节马达提供一个最大扭矩。关节马达会维持在指定的速度，除非其所需的扭矩超出了最大扭矩。当超出最大扭矩时，关节会慢下来，甚至会反向运动。

你还可以使用关节马达来模拟关节摩擦。只要把关节速度设为0，并将最大扭矩设为很小但有效的值。这样马达会试图阻止关节旋转，除非有过大的负载。

这里是对上面旋转关节定义的修订; 这次，关节拥有一个限制以及一个马达，后者用于模拟摩擦。

b2RevoluteJointDef jointDef;

jointDef.Initialize(bodyA, bodyB, myBodyA->GetWorldCenter());

jointDef.lowerAngle = -0.5f \* b2\_pi; // -90 degrees

jointDef.upperAngle = 0.25f \* b2\_pi; // 45 degrees

jointDef.enableLimit = true;

jointDef.maxMotorTorque = 10.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

你可以访问旋转关节的角度，速度和马达扭矩。

float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;

每次执行step后，你也可以更新马达的参数。

void SetMotorSpeed(float32 speed);

void SetMaxMotorTorque(float32 torque);

关节马达有些有趣的功能。你可以在每个时间步中更新关节的速度，使得它像正弦波或者任意一个你想要的函数那样前后摆动。

... Game Loop Begin ...

myJoint->SetMotorSpeed(cosf(0.5f \* time));

... Game Loop End ...

你也可以用关节马达来跟踪你想要的关节角。比如：

... Game Loop Begin ...

float32 angleError = myJoint->GetJointAngle() - angleTarget;

float32 gain = 0.1f;

myJoint->SetMotorSpeed(-gain \* angleError);

... Game Loop End ...

通常你的增益参数不应太大，不然关节会变得不稳定。

## 移动关节

移动关节(prismatic joint)允许两个物体沿指定轴相对移动，它会阻止相对旋转。因此，移动关节只有一个自由度。



移动关节的定义有些类似于旋转关节；只是转动角度换成了平移，扭矩换成了力。以这样的类比，我们来看一个带有关节限制以及马达摩擦的移动关节定义：

b2PrismaticJointDef jointDef;

b2Vec2 worldAxis(1.0f, 0.0f);

jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter(), worldAxis);

jointDef.lowerTranslation = -5.0f;

jointDef.upperTranslation = 2.5f;

jointDef.enableLimit = true;

jointDef.maxMotorForce = 1.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

旋转关节隐含着一个从屏幕射出的轴，而移动关节明确地需要一个平行于屏幕的轴。这个轴会固定于两个物体之上，沿着它们的运动方向。

就像旋转关节一样，当使用 Initialize() 创建移动关节时，移动为0。所以要确保0在你的移动限制范围内。

移动关节的用法跟旋转关节类似。这是相应的成员函数：

float32 GetJointTranslation() const;

float32 GetJointSpeed() const;

float32 GetMotorForce() const;

void SetMotorSpeed(float32 speed);

void SetMotorForce(float32 force);

## 滑轮关节

滑轮关节用于创建理想的滑轮，它将两个物体接地(ground)并彼此连接。这样，当一个物体上升，另一个物体就会下降。滑轮的绳子长度取决于初始配置。

length1 + length2 == constant

你还可以提供一个系数(ratio)来模拟滑轮组，这会使滑轮一侧的运动比另一侧要快。同时，一侧的约束力也比另一侧要小。你也可以用这个来创建机械杠杆。

length1 + ratio \* length2 == constant

举个例子，如果系数是2，那么 length1 的变化会是 length2 的两倍。另外连接body1的绳子的约束力将会是连接body2绳子的一半。



滑轮的一侧完全展开时,另一侧的绳子长度为零，这可能会出问题。此时，约束方程将变得奇异(糟糕)。你应该配置碰撞形状以避免这种情况。

这是一个滑轮定义的例子：

b2Vec2 anchor1 = myBody1->GetWorldCenter();

b2Vec2 anchor2 = myBody2->GetWorldCenter();

b2Vec2 groundAnchor1(p1.x, p1.y + 10.0f);

b2Vec2 groundAnchor2(p2.x, p2.y + 12.0f);

float32 ratio = 1.0f;

b2PulleyJointDef jointDef;

jointDef.Initialize(myBody1, myBody2, groundAnchor1, groundAnchor2, anchor1, anchor2, ratio);

滑轮关节提供函数得到当前长度。

float32 GetLengthA() const;

float32 GetLengthB() const;

## 齿轮关节

如果你想创建复杂的机械装置，可能需要齿轮。原则上，在 Box2D 中你可以用复杂的形状来模拟轮齿，但这并不十分高效，而且可能有些乏味。另外，你还得小心地排列齿轮，保证轮齿能平稳地啮合。Box2D 提供了一个创建齿轮的更简单的方法：齿轮关节。



齿轮关节只能连接旋转关节和移动关节。

类似于滑轮系数，你可以指定一个齿轮系数(ratio)，齿轮系数可以为负。另外值得注意的是，当一个是旋转关节(有角度的)而另一个是移动关节(平移)时，齿轮系数有长度单位，或者是长度单位的倒数。

coordinate1 + ratio \* coordinate2 == constant

这是一个齿轮关节的例子。物体myBodyA和myBodyB来自于两个关节，并且它们不是同一个物体。

b2GearJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.joint1 = myRevoluteJoint;

jointDef.joint2 = myPrismaticJoint;

jointDef.ratio = 2.0f \* b2\_pi / myLength;

注意，齿轮关节依赖于两个其它关节，这是脆弱的：当其它关节被删除了会发生什么?

**注意**

齿轮关节总应该先于旋转或移动关节被删除。否则你的代码将会因访问齿轮关节的孤儿关节指针而导致崩溃。另外齿轮关节也应该在任何相关物体被删除之前删除。

## 鼠标关节

在testbed例子中，鼠标关节用于通过鼠标来操控物体。它试图将物体拖向当前鼠标光标的位置。而在旋转方面就没有限制。

鼠标关节的定义需要一个目标点(target point)，最大力(maximum force)，频率(frequency)，阻尼率(damping ratio)。目标点最开始与物体的锚点重合。最大力用于防止在多个动态物体相互作用时的激烈反应。你想将最大力设为多大就多大。频率和阻尼率用于创造一种弹性效果，就跟距离关节类似。

许多用户为了游戏的可玩性，会试图修改鼠标关节。用户常常希望鼠标关节有即时反应，精确的去到某个点。这情况下，鼠标关节表现并不好。你可以考虑一下用kinematic物体来替代。

## 轮子关节

轮子关节限制bodyB上的一个点到bodyA的一条线上。轮子关节也提供悬置弹簧的效果。细节参见b2WheelJoint.h和Car.h。



## 焊接关节

焊接关节的用途是使两个物体不能相对运动。看看testbed中的Cantilever例子，可以知道焊接关节有怎么样的表现。

用焊接关节来定义一个可分裂物体，这想法很诱人。但是，由于Box2D的迭代求解，关节焊得有点不稳。因此用焊接关节连接起来的物体会有所摆动。

创建可裂物体的更好方法是使用单个的物体，上面有很多fixture。 当物体分裂时，你可以删掉原物体的一个fixture，并重新在一个新的物体上创建它。参考一下testbed中的Breakable例子。

## 绳子关节

绳子关机限制了两个点之间的最大距离。它能够阻止连接的物体之间的拉伸，即使在很大的负载下。细节参见b2RopeJoint.h和RopeJoint.h。

## 摩擦关节

摩擦关节被用于模拟上下摩擦。关节提供2D的传统摩擦和角度摩擦。细节参见b2FrictionJoint.h 和ApplyForce.h。

## 马达关节

马达关节通过指定目标位置和旋转偏移来控制物体的移动。你能设置最大的马达力和力矩来到达目标位置和旋转。如果物体被阻塞，它将停下来，接触力与最大的马达力和力矩成正比。细节参见b2MotorJoint和MotorJoint.h。

# 接触

## 关于

接触(contact)是由 Box2D 创建的用于管理fixture间碰撞的对象。如果 fixture 有诸如链接形状之类的子 fixture ，那么每个相应的子 fixture 都存在接触。接触有不同的种类，它们都派生自 b2Contact，用于管理不同类型 fixture 之间的接触。例如，有管理多边形之间碰撞的类，有管理圆形之间碰撞的类。

这是与接触有关的术语。

### 接触点

接触点就是两个形状相互接触的点。在Box2D中，近似地认为在少数点处有接触。

### 接触法线

接触法线是一个单位向量，由一个fixture指向另一个fixture。按照惯例，向量由fixtureA指向fixtureB。

### 接触分隔

分隔正好与穿透(penetration)相反。当形状相重叠时，分隔为负。有可能将来的Box2D版本中会以正隔离来创建触点，所以当有触点的报告时你可能需要检查一下符号。

### 接触流形

两个凸多边形相互接触，有可能会产生两个接触点。这些点都有相同的法线，所以它们被分成一组，构成接触流形，这是连续区域接触的一个近似。

### 法向冲量

法向力作用于接触点，用于防止形状相互穿透。为方便起见，Box2D使用冲量(impulses)。法向力与时间步相乘，构成法向冲量。

### 切向冲量

切向力会在接触点生成，用于模拟摩擦。为方便起见，切向作用使用冲量的方式存储。

### 接触标识

Box2D试图复用上一个时间步计算出的接触力，做为下一个时间步的初始估计值。Box2D使用接触标识匹配跨越时间步的触点。标识包含了几何特征索引以便区分接触点。

当两个fixture的AABB重叠时，接触就被创建了。有时碰撞筛选会阻止接触的创建。当AABB 不再重叠，接触就会被摧毁。

也许你会皱起眉头，为了没有发生实际碰撞的形状(只是它们的 AABB)却创建了接触。好吧，的确是这样的，这是一个“鸡或蛋”的问题。我们并不知道是否需要一个接触，除非我们创建一个接触去分析碰撞。如果形状之间没有发生碰撞，我们需要正确地删除接触，或者，我们可以一直等到 AABB 不再重叠。为了提高性能，Box2D选择了后面这个方法，因为它可以使用系统的缓冲信息来提升性能。

## 接触类

之前已经提及过，接触对象是Box2D内部创建和摧毁的，并不是由用户来创建。然而，你还是能够访问接触类并和它交互的。

你可以访问原始的接触流形：

b2Manifold\* GetManifold();

const b2Manifold\* GetManifold() const;

你甚至可以修改流形，一般情况下不提倡你怎样做。修改流形是较高级的用法。

这个是获取b2WorldManifold的帮助函数:

void GetWorldManifold(b2WorldManifold\* worldManifold) const;

这使用了物体的当前位置去计算出接触点在世界坐标下的位置。

传感器(Sensors)并不创建流形，所以要使用：

bool touching = sensorContact->IsTouching();

这函数对于非传感器(non-sensors)也有效。

从接触(contact)中你可以得到fixture，从而再得到body。

b2Fixture\* fixtureA = myContact->GetFixtureA();

b2Body\* bodyA = fixtureA->GetBody();

MyActor\* actorA = (MyActor\*)bodyA->GetUserData();

你可以使一个接触失效。这仅仅在b2ContactListener::PreSolve事件中有效，下面会再进行讨论。

## 访问接触

你有几种方法来访问接触。为了访问接触，你可以直接查询world或者body结构，你还可以实现一个接触监听器(contact listener)。

在world中，你可以遍历所有的接触：

for (b2Contact\* c = myWorld->GetContactList(); c; c = c->GetNext())

{

// process c

}

同样在body中，你也可以遍历所有接触。接触以图的方式存储，使用了接触边数据结构(contact edge structure)。

for (b2ContactEdge\* ce = myBody->GetContactList(); ce; ce = ce->next)

{

b2Contact\* c = ce->contact;

// process c

}

通过下面描述的接触监听器，你也可以访问接触。

**注意**

通过b2World或者b2Body直接访问，有可能会错过一些时间步中产生的临时接触。而使用b2ContactListener 就可以很精确的得到全部结果。

## 接触监听器

通过实现b2ContactListener你就可以收到接触数据。接触监听器支持几种事件：开始(begin)，结束(end)，求解前(pre-solve)和求解后(post-solve)。

class MyContactListener : public b2ContactListener

{

public:

void BeginContact(b2Contact\* contact)

{ /\* handle begin event \*/ }

void EndContact(b2Contact\* contact)

{ /\* handle end event \*/ }

void PreSolve(b2Contact\* contact, const b2Manifold\* oldManifold)

{ /\* handle pre-solve event \*/ }

void PostSolve(b2Contact\* contact, const b2ContactImpulse\* impulse)

{ /\* handle post-solve event \*/ }

};

**注意**

不要保存发送到b2ContactListener的指针。取而代之，用深拷贝的方式将触点数据保存到你自己的缓冲区。下面的例子演示了一种操作方法。

在运行期(run-time)，你可以创建listener的实例对象，并使用b2World::SetContactListener来注册这个对象。 但要保证当listener在作用域中时，world对象是存在的。

### 开始接触事件

当两个fixture开始有重叠时，事件会被触发。传感器和非传感器都会触发这事件。这事件只能在时间步内(译注: 也就是b2World::step函数内部)发生。

### 结束接触事件

当两个fixture不再重叠时，事件会被触发。传感器和非传感器都会触发这事件。当一个body被摧毁时，事件也有可能被触发。所以这事件也有可能发生在时间步之外。

### 求解前事件

在碰撞检测之后，但在碰撞求解之前，事件会被触发。这样可以给你一个机会，根据当前的配置来决定是否使这个接触失效。 举个例子，在回调中使用b2Contact::SetEnabled(false)，你就可以实现单侧平台（译注：类似于半透膜那样的东西，允许一侧的物体无障碍的穿过，而另一侧的物体无法穿过。）的功能。每次碰撞处理时，接触会重新生效，所以你在每一个时间步中都应禁用那个接触。由于连续的碰撞检测，pre-solve事件在单个时间步中有可能发生多次。

void PreSolve(b2Contact\* contact, const b2Manifold\* oldManifold)

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

if (worldManifold.normal.y < -0.5f)

{

contact->SetEnabled(false);

}

}

如果要确认触点状态或得到碰撞前的速度，可以在pre-solve事件中处理。

void PreSolve(b2Contact\* contact, const b2Manifold\* oldManifold)

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

b2PointState state1[2], state2[2];

b2GetPointStates(state1, state2, oldManifold, contact->GetManifold());

if (state2[0] == b2\_addState)

{

const b2Body\* bodyA = contact->GetFixtureA()->GetBody();

const b2Body\* bodyB = contact->GetFixtureB()->GetBody();

b2Vec2 point = worldManifold.points[0];

b2Vec2 vA = bodyA->GetLinearVelocityFromWorldPoint(point);

b2Vec2 vB = bodyB->GetLinearVelocityFromWorldPoint(point);

float32 approachVelocity = b2Dot(vB – vA, worldManifold.normal);

if (approachVelocity > 1.0f)

{

MyPlayCollisionSound();

}

}

}

### 求解后事件

你可以在post-solve事件中，得到碰撞冲量(collision impulse)的结果。 如果你不关心冲量，你可能只需要实现pre-solve事件。

在一个接触回调中去改变物理世界是诱人的。例如，你可能会以碰撞来施加伤害，并试图摧毁关联的角色和它的刚体。然而，Box2D并不允许你在回调中改变物理世界，因为你可能会摧毁 Box2D正在处理的对象，造成野指针。

处理接触点的推荐方法是缓冲所有你关心的接触数据，并在时间步之后处理它们。一般在时间步之后你应该立即处理它们，否则其它客户端代码可能会改变物理世界，使你的接触缓冲失效。当你处理接触缓冲的时候，你可以去改变物理世界，但是你仍然应该小心不要在接触点缓冲区造成无效的指针。在 testbed中有安全处理触点以避免无效指针的例子。

这是一小段CollisionProcessing测试中的代码，它演示了在操作接触缓冲时，如何处理孤立物体。这里是节选，请注意阅读注释。代码假定所有触点都缓冲于b2ContactPoint类型的数组m\_points中。

// 我们打算摧毁和contact指针有关联的物体。

// 我们必须先缓存那些需要摧毁的物体，因为它们有可能被多个触点所共有。

const int32 k\_maxNuke = 6;

b2Body\* nuke[k\_maxNuke];

int32 nukeCount = 0;

// 遍历contact缓存，摧毁那些正在和更重的物体接触的物体。

for (int32 i = 0; i < m\_pointCount; ++i)

{

ContactPoint\* point = m\_points + i;

b2Body\* bodyA = point->fixtureA->GetBody();

b2Body\* bodyB = point->FixtureB->GetBody();

float32 massA = bodyA->GetMass();

float32 massB = bodyB->GetMass();

if (massA > 0.0f && massB > 0.0f)

{

if (massB > massA)

{

nuke[nukeCount++] = bodyA;

}

else

{

nuke[nukeCount++] = bodyB;

}

if (nukeCount == k\_maxNuke)

{

break;

}

}

}

// 将nuke数组排序，使得重复的指针归在一起

std::sort(nuke, nuke + nukeCount);

// 删除body, 忽略重复的

int32 i = 0;

while (i < nukeCount)

{

b2Body\* b = nuke[i++];

while (i < nukeCount && nuke[i] == b)

{

++i;

}

m\_world->DestroyBody(b);

}

## 接触筛选

通常，你不希望游戏中的所有物体都发生碰撞。例如，你可能会创建一个只有特定角色才能通过的门。 这称之为接触筛选，因为一些交互被筛选出了。

通过实现b2ContactFilter类，Box2D允许定制接触筛选。这个类需要你实现一个ShouldCollide 函数，这个函数接收两个b2Shape的指针作为参数。如果形状会碰撞，那么就返回true。

默认的ShouldCollide实现使用了“第06章，夹具(Fixtures)”定义的b2FilterData。

bool b2ContactFilter::ShouldCollide(b2Fixture\* fixtureA, b2Fixture\* fixtureB)

{

const b2Filter& filterA = fixtureA->GetFilterData();

const b2Filter& filterB = fixtureB->GetFilterData();

if (filterA.groupIndex == filterB.groupIndex && filterA.groupIndex != 0)

{

return filterA.groupIndex > 0;

}

bool collide = (filterA.maskBits & filterB.categoryBits) != 0 &&

(filterA.categoryBits & filterB.maskBits) != 0;

return collide;

}

在运行期（run-time)，你可以创建自己的接触筛选实例，并使用b2World::SetContactFilter函数来注册。 你要保证当world存在时，你的filter要保留在作用域中。

MyContactFilter filter;

world->SetContactFilter(&filter);

// filter留在作用域中

# 世界类

### 关于

b2World类包含物体和关节。它管理着模拟的方方面面，并允许异步查询(例如AABB查询和光线投射)。 你与Box2D的大部分交互都将通过 b2World 对象来完成。

### 创建和摧毁世界

创建world十分的简单。你只需提供一个重力矢量，和一个布尔量去指定物体是否可以休眠。 通常你会使用new和delete去创建和摧毁一个world。

b2World\* myWorld = new b2World(gravity, doSleep);

... do stuff ...

delete myWorld;

### 使用世界

world类含有用于创建和摧毁物体与关节的工厂函数,，这些工厂函数已在物体和关节的章节中讨论过。 在这里我们讨论b2World的其它交互。

### 模拟

世界类用于驱动模拟。你需要指定一个时间步和一个速度及位置的迭代次数。例如：

float32 timeStep = 1.0f / 60.f;

int32 velocityIterations = 10;

int32 positionIterations = 8;

myWorld->Step(timeStep, velocityIterations, positionIterations);

在时间步完成之后，你可以调查物体和关节的信息。最经常的情况是你会获取物体的位置，这样你才能更新你的角色并渲染它们。你可以在游戏循环的任何地方执行时间步，但你应该注意事情发生的先后顺序。例如，如果你想要在一帧(frame)中得到新物体的碰撞结果，你必须在时间步之前创建物体。

正如之前我在 HelloWorld 教程中说明的，你需要使用一个固定的时间步。使用大一些的时间步你可以在低帧率的情况下提升性能。但通常情况下你应该使用一个不大于1/30秒的时间步。1/60 的时间步通常会呈现一个高质量的模拟。

迭代次数控制了约束求解器会遍历多少次世界中的接触以及关节。更多的迭代总能产生更好的模拟，但不要使用小频率大迭代数。60Hz和10次迭代远好于30Hz和20次迭代。

时间步之后，你应该清除任何施加到物体之上的力。使用b2World::ClearForces命令可以完成。这会让你在多个子步中使用相同的力。

myWorld->ClearForces();

### 探测世界

世界是物体、接触和关节的容器。你可以获取世界中所有物体、接触和关节并遍历它们。例如，这段代码会唤醒世界中的所有物体：

for (b2Body\* b = myWorld->GetBodyList(); b; b = b->GetNext())

{

b->SetAwake(true);

}

不幸的是，真实的程序可能很复杂。例如，下面的代码是有错误的：

for (b2Body\* b = myWorld->GetBodyList(); b; b = b->GetNext())

{

GameActor\* myActor = (GameActor\*)b->GetUserData();

if (myActor->IsDead())

{

myWorld->DestroyBody(b); // ERROR: now GetNext returns garbage.

}

}

在物体摧毁之前一切都很顺利。一旦物体摧毁了, 它的next指针就变得非法。所以 b2Body::GetNext() 就会返回无用信息。 解决方法是在物体摧毁之前拷贝next指针。

b2Body\* node = myWorld->GetBodyList();

while (node)

{

b2Body\* b = node;

node = node->GetNext();

GameActor\* myActor = (GameActor\*)b->GetUserData();

if (myActor->IsDead())

{

myWorld->DestroyBody(b);

}

}

这能安全地摧毁当前物体。然而，你可能想要调用一个游戏的函数来摧毁多个物体，这时你需要十分小心。 解决方案取决于具体应用，但为求方便，在此我给出一种解决这问题的方法。

b2Body\* node = myWorld->GetBodyList();

while (node)

{

b2Body\* b = node;

node = node->GetNext();

GameActor\* myActor = (GameActor\*)b->GetUserData();

if (myActor->IsDead())

{

bool otherBodiesDestroyed = GameCrazyBodyDestroyer(b);

if (otherBodiesDestroyed)

{

node = myWorld->GetBodyList();

}

}

}

很明显要保证这个能正确工作, GameCrazyBodyDestroyer对它都摧毁了什么必须要诚实。

### AABB查询

有时你需要得出一个区域内的所有fixture。b2World类为此使用了broad-phase数据结构，并提供了一个 log(N) 的快速方法。你提供一个世界坐标的AABB和b2QueryCallback的一个实现。只要fixture的AABB和需查询的AABB有重 合，world类就会调用你的b2QueryCallback类。返回true表示要继续查询，否则就返回false。例如，下面的代码找到所有大致与指 定AABB相交的fixtures并唤醒所有关联的物体。

class MyQueryCallback : public b2QueryCallback

{

public:

bool ReportFixture(b2Fixture\* fixture)

{

b2Body\* body = fixture->GetBody();

body->SetAwake(true);

// Return true to continue the query.

return true;

}

};

...

MyQueryCallback callback;

b2AABB aabb;

aabb.lowerBound.Set(-1.0f, -1.0f);

aabb.upperBound.Set(1.0f, 1.0f);

myWorld->Query(&callback, aabb);

你不能假定回调函数会以固定的顺序执行。

### 光线投射

你可以使用光线投射去做现场(line-of-site)检查，开枪扫射等等。通过实现一个回调类，并提供一个开始点和结束点，你就可以执行光线投射。只要fixture被光线穿过，world就会调用你提供的类。回调时会传递fixture，交点，单位法向量，和光线通过的分数距离(fractional distance along the ray)。你不能假定回调会以固定的顺序执行。

通过返回fraction， 你可以控制光线投射是否继续执行。返回的fraction为0，表示应该结束光线投射。fraction为1，表示投射应该继续执行，并且没有和其它形状 相交。如果你返回参数列表中传进来的fraction，表示光线会被裁剪到当前的和形状的相交点。这样通过返回适当的fraction值，你可以投射任何形状，投射所有形状，或者只投射最接近的形状。

另外你可以返回fraction为-1，去过滤fixture。这样光线投射会继续执行，并表现得似乎fixture根本就存在。

这里是个例子：

// This class captures the closest hit shape.

class MyRayCastCallback : public b2RayCastCallback

{

public:

MyRayCastCallback()

{

m\_fixture = NULL;

}

float32 ReportFixture(b2Fixture\* fixture, const b2Vec2& point,

const b2Vec2& normal, float32 fraction)

{

m\_fixture = fixture;

m\_point = point;

m\_normal = normal;

m\_fraction = fraction;

return fraction;

}

b2Fixture\* m\_fixture;

b2Vec2 m\_point;

b2Vec2 m\_normal;

float32 m\_fraction;

};

MyRayCastCallback callback;

b2Vec2 point1(-1.0f, 0.0f);

b2Vec2 point2(3.0f, 1.0f);

myWorld->RayCast(&callback, point1, point2);

**注意**

由于舍入误差，光线投射可能会通过在静态环境中的多边形之间的细小裂缝。如果这不是您的应用程序中的可接受的，请稍微扩大您的多边形。

void SetLinearVelocity(const b2Vec2& v);

b2Vec2 GetLinearVelocity() const;

void SetAngularVelocity(float32 omega);

float32 GetAngularVelocity() const;

### 力与冲量

你可以将力、扭矩和冲量应用到物体上。当应用一个力或者冲量时，你需要提供一个在世界坐标下的受力点。这经常导致相对于质心，会有个扭矩。

void ApplyForce(const b2Vec2& force, const b2Vec2& point);

void ApplyTorque(float32 torque);

void ApplyLinearImpulse(const b2Vec2& impulse, const b2Vec2& point);

void ApplyAngularImpulse(float32 impulse);

应用力、扭矩或冲量会唤醒物体。有时这是不合需求的。例如，你可能想要应用一个固定的力，并允许物体休眠来提升性能。这时，你可以使用这样的代码：

if (myBody->IsAwake() == true)

{

myBody->ApplyForce(myForce, myPoint);

}

### 坐标转换

body类包含一些工具函数，它们可以帮助你在局部和世界坐标系之间转换点和向量。如果你不了解这些概念，请看 Jim Van Verth 和 Lars Bishop 的“游戏和交互应用的数学基础(Essential Mathematics for Games and Interactive Applications)”。这些函数都很高效(当内联时)。

b2Vec2 GetWorldPoint(const b2Vec2& localPoint);

b2Vec2 GetWorldVector(const b2Vec2& localVector);

b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);

b2Vec2 GetLocalVector(const b2Vec2& worldVector);

### 列表

你可以遍历一个物体的fixture, 主要用途是帮助你访问fixture中的用户数据。

for (b2Fixture\* f = body->GetFixtureList(); f; f = f->GetNext())

{

MyFixtureData\* data = (MyFixtureData\*)f->GetUserData();

... do something with data ...

}

你也可以用类似的方法遍历物体的关节列表。

body也提供了访问相关contact的列表。你可以用来得到当前contact的信息。但使用时请小心，因为前一个时间步存在的contact，可能并不包含在当前列表中。

# 杂项

## 用户数据

b2Fixture, b2Body 和 b2Joint 类都允许你通过一个 void 指针来附加用户数据。当你测试Box2D数据结构，并使其跟自己游戏引擎中的对象结合起来时，这样做是比较方便的。

举个典型的例子，角色上附有物体，并在物体中附加角色的指针，这就构成了一个循环引用。如果你有角色(actor)，你就能得到物体。如果你有物体，你也能得到角色。

GameActor\* actor = GameCreateActor();

b2BodyDef bodyDef;

bodyDef.userData = actor;

actor->body = box2Dworld->CreateBody(&bodyDef);

一些需要用户数据的例子：

* 使用碰撞结果给角色施加伤害效果。
* 当玩家进入一个包围盒(axis-aligned box)时，触发脚本事件。
* 当Box2D通知你关节将要被摧毁时，去访问某个游戏结构。

记住，用户数据是可选的，并且能放入任何东西。然而，你需要确保一致性。例如，如果你想在body中保存actor的指针，那你就应该在所有的 body中都保存actor指针。不要在一个body中保存actor指针，却在另一个body中保存foo指针。将一个actor指针强制转成foo指 针，可能会导致程序崩溃。

用户数据指针默认为NULL。

对于fixture来说，你可以定义一个用户数据结构来存储游戏特定的信息。例如材料类型、特效钩子、音效钩子，等等。

struct FixtureUserData

{

int materialIndex;

…

};

FixtureUserData myData = new FixtureUserData;

myData->materialIndex = 2;

b2FixtureDef fixtureDef;

fixtureDef.shape = &someShape;

fixtureDef.userData = myData;

b2Fixture\* fixture = body->CreateFixture(&fixtureDef);

…

delete fixture->GetUserData();

fixture->SetUserData(NULL);

body->DestroyFixture(fixture);

## 隐式摧毁

Box2D没有使用引用计数。因此你摧毁了body后，它就确实不存在了。访问指向已摧毁body的指针，会导致未定义的行为。 也就是说，你的程序可能会崩溃。以debug方式编译出的程序，Box2D的内存管理器会将已被摧毁实体占用的内存，都填上FDFDFDFD。在某些时候， 这样做可以使你更容易的找到问题的所在，并进而修复问题。

如果你摧毁了Box2D实体，你要确保所有指向这实体的引用都被移除。如果只有实体的单个引用，处理起来就很简单了。但如果有多个引用，你需要考虑是否去实现一个句柄类(handle class)，将原始指针封装起来。

当你使用Box2D时，会很频繁的创建并摧毁很多的物体(bodies)、形状(shapes)和关节(joints)。某种程度上，这些实体是由Box2D自动管理的。当你摧毁了一个body，所有跟它有关联的形状、关节和接触都会自动被摧毁。这称为隐式摧毁。

连接到这些关节或接触上的物体会被唤醒。这个过程是很简便的，但是你必须小心一个关键问题：

**注意**

当body被摧毁时，所有依附其上的形状和关节也会被自动摧毁。你应该将任何指向这些形状和关节的指针清零。否则如果你之后试图访问或者再次摧毁这些形状或关节，你的程序会死得很惨。

为了帮助你清空关节指针，Box2D提供了一个名叫b2DestructionListener的监听类。你可以实现这个类，并将world对象传给它。这样当关节被隐式摧毁时，world对象会通知你。

注意，关节或fixture被显式摧毁时，并没有通知。这种情况下，所有者是清楚的，你可以在合适的地方执行必要的清理工作。如果你喜欢，你也可以调用你自己的b2DestructionListener的实现来保持清理代码的集中。

大多数情况下，隐式摧毁还是很方便的，但也很容易使你的程序崩溃。你可能会将指向shape或关节的指针保存起来，当有关联的body摧毁时，这些指针就变得无效了。当关节是由那些与相关的物体的管理代码无关的代码片段创建时，事情会变得更糟糕。比如，testbed就创建了一个 b2MouseJoint用来交互式操作屏幕上的body。

Box2D提供了一种回调机制，用于在隐式摧毁发生时，通知你的应用程序。这给了你的程序一个机会，将无效指针清零。这个回调机制将在稍后详述。

你可以实现一个b2DestructionListener，这样当一个形状或关节隐式摧毁时，b2World 就能通知你。这将防止你的代码访问野指针。

class MyDestructionListener : public b2DestructionListener

{

void SayGoodbye(b2Joint\* joint)

{

// remove all references to joint.

}

};

你可以将摧毁监听器(destruction listener)注册到world对象中。在world对象初始化时，你就应该这样做了。

myWorld->SetListener(myDestructionListener);

## 像素和坐标系统

重申一下Box2D使用MKS（米、千克和秒）单位制，角度使用弧度为单位。你可能对使用米感到困惑，因为你的游戏是以像素的形式表示的。在testbed中为了解决这个问题，我将整个游戏都使用米，并使用OpenGL的视图转换，将world调整到屏幕空间中。

float lowerX = -25.0f, upperX = 25.0f, lowerY = -5.0f, upperY = 25.0f;

gluOrtho2D(lowerX, upperX, lowerY, upperY);

如果你的游戏必须使用像素为单位，在向Box2D传送值的时候，你应该将你的长度单位由像素转换成米。反之，当你接收Box2D传来的值时，应该将之由米转换成像素。这会提升物理模拟的稳定性。

你必须设置一个合理的转换因子。我建议可以根据你的角色的尺寸来做出选择。假设你每米使用50个像素（因为你的角色有75个像素的高度），则你可以使用下面这些公式将像素转换成米：

xMeters = 0.02f \* xPixels;

yMeters = 0.02f \* yPixels;

相反的：

xPixels = 50.0f \* xMeters;

yPixels = 50.0f \* yMeters;

你应该在你的代码中使用MKS单位制，只在你渲染的时候，将之转换成像素。这会简化你的游戏逻辑，并减少错误的可能。因为渲染时的变换，能被限定在很小的代码中。

如果你使用转换因子，你应该在全局范围内调整它，确保没有错误发生。你也可以调整它来改善稳定度。

# 调试绘图

实现 b2DebugDraw 可得到物理世界的细部图，这里是可用的实体：

* 形状轮廓
* 关节连通性
* broad-phase axis-aligned bounding boxes (AABBs)
* 质心



这是绘制这些物理实体的首选方法，比直接访问数据要好。因为很多的必要信息只能在内部访问，并时有变更。

testbed使用debug drawing和接触监听器来绘制物理实体。它本身就是个好例子，演示了怎样去实现debug drawing以及怎样绘制接触点。

# 限制

Box2D使用了一些数值近似来让模拟更高效。这就带来一些限制。

这是当前的限制：

1. 将重的物体放到相对很轻的物体上面，会不稳定。当质量比到10:1时，稳定性就会降低。
2. 用关节将body链接起来，如果是较轻的物体吊着较重的物体，body链接有可能被拉伸。比如，一条很轻的锁链吊着个很重的球，就可能不稳定。当质量比超过10:1时，稳定性就会降低。
3. 通常还有约0.5cm的间隙，就检测到形状与形状碰撞。
4. 连续碰撞不会处理关节，因此你会看到在快速移动的物体上的关节拉伸。
5. Box2D使用欧拉积分法，它不会导致物体的抛物线运动，并只有一阶导数的精度。然而这种方法足够快并有很好的稳定性。
6. Box2D使用迭代求解器来提供实时计算。你不可能得到绝对准确的刚体碰撞和像素。增加迭代的次数会提升准确性。

# 参考

Erin Catto’s GDC Tutorials: <http://code.google.com/p/box2d/downloads/list>

Collision Detection in Interactive 3D Environments, Gino van den Bergen, 2004

Real-Time Collision Detection, Christer Ericson, 2005