[MongoDB系列（二）：C#应用](http://www.cnblogs.com/Erik_Xu/p/5514804.html)

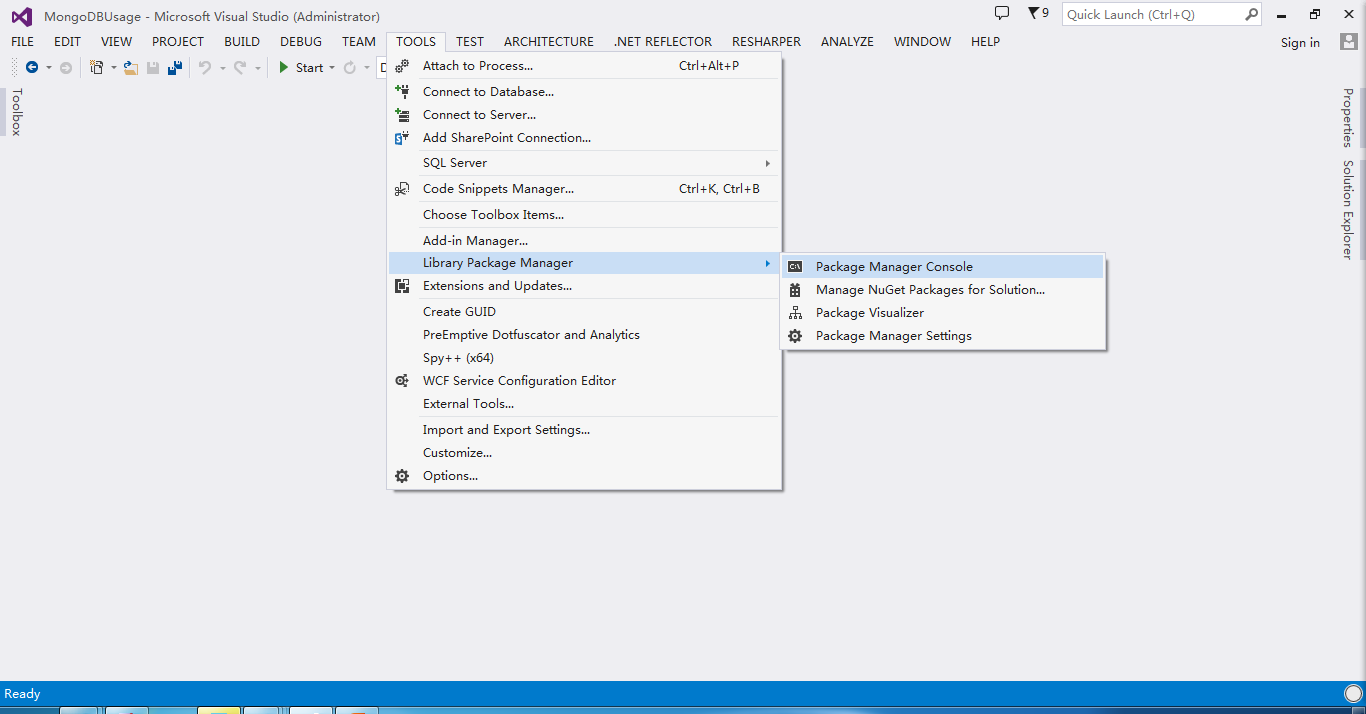
**前言**

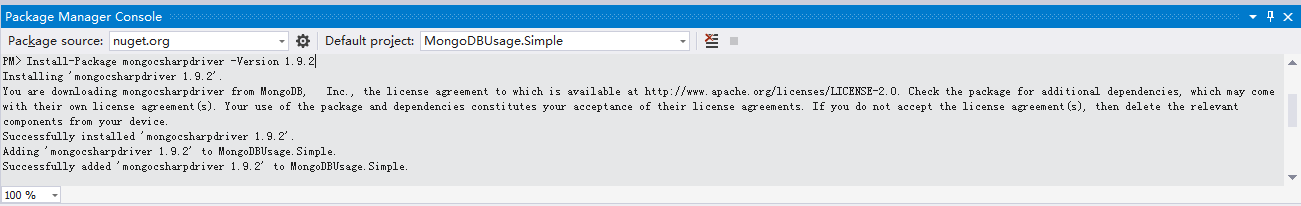
　　上一篇文章[《MongoDB系列（一）：简介及安装》](http://www.cnblogs.com/Erik_Xu/p/5514799.html" \t "_blank)已经介绍了MongoDB以及其在window环境下的安装，这篇文章主要讲讲如何用C#来与MongoDB进行通讯。再次强调一下，我使用的MongoDB版本是2.6，因为2.6是我最熟悉的版本，而且我使用的GUI工具Robomongo目前还不支持3.0版本。

**添加官方驱动**

　　官方驱动可以从Nuget上获取，但是这里我们不使用最新的驱动，而是使用1.9.2这个版本，个人认为该版本对MongoDB2.6的支持最好，而且目前的下载量也是最多。驱动地址：[https://www.nuget.org/packages/mongocsharpdriver/1.9.2](https://www.nuget.org/packages/mongocsharpdriver/1.9.2" \t "_blank)。因此，需要在程序包管理器中获取Nuget。

打开“程序包管理器中”



输入指令Install-Package mongocsharpdriver -Version 1.9.2，下载添加驱动

**连接字符串**

mongodb://[username:password@]host1[:port1][,host2[:port2],…[,hostN[:portN]]][/[database][?options]]

|  |  |
| --- | --- |
| 内容 | 描述 |
| mongodb:// | 是连接字串必须的前缀字串 |
| username:password@ | 可选项，连接到数据库后会尝试验证登陆 |
| host1 | 必须的指定至少一个host |
| :portX | 可选项，默认连接到27017 |
| /database | 如果指定username:password@，连接并验证登陆指定数据库。若不指定，默认打开admin数据库。 |
| ?options | 是连接选项。如果不使用/database，则前面需要加上/。所有连接选项都是键值对name=value，键值对之间通过&或;（分号）隔开 |

**C#驱动提供的常用API**

|  |  |
| --- | --- |
| 方法 | 描述 |
| InsertBatch | 批量插入 |
| Insert | 单条插入 |
| FindOneById | 按Id查询 |
| Save | 保存，如果库中有记录则更新，否则做插入，按Id匹配 |
| Remove | 删除指定文档 |
| AsQueryable | 返回IQueryable<T>对象 |
| Update | 更新一个或多个文档 |
| RemoveAll | 删除所有记录 |
| … | 其它 |

**代码说明**

抽象实体类Entity

[复制代码](javascript:void(0);)

public abstract class EntityWithTypedId<TId>

{

public TId Id { get; set; }

}

public abstract class Entity : EntityWithTypedId<ObjectId>

{

}

[复制代码](javascript:void(0);)

　　MongoDB要求每个集合都需要有一个Id，即使你定义的类中没有Id字段，存数据的时候也会生成一个Id，而且Id的类型默认是使用ObjectId，当然也可以使用其他简单类型作为Id，如int。

核心代码封装DbContext

[复制代码](javascript:void(0);)

public class DbContext

{

private readonly MongoDatabase \_db;

public DbContext()

{

var client = new MongoClient("mongodb://localhost:27017");

var server = client.GetServer();

\_db = server.GetDatabase("Temp");

}

public MongoCollection<T> Collection<T>() where T : Entity

{

var collectionName = InferCollectionNameFrom<T>();

return \_db.GetCollection<T>(collectionName);

}

private static string InferCollectionNameFrom<T>()

{

var type = typeof(T);

return type.Name;

}

}

[复制代码](javascript:void(0);)

 　　1. 通过连接字符串与数据库建立连接。

　　 2. 获取需要操作的Database，这里是Temp。

　　 3. 类名与Collection名一致，作为映射的约束。如果库中没有这个Collection，则创建该Collection，如果有，则操作该Collection。

定义一个股票类Stock，包含股票代码，股票名称，股票价格等简单类型字段以及股票粉丝复杂字段：

[复制代码](javascript:void(0);)

public class Stock : Entity

{

public string Symbol { get; set; }

public string Name { get; set; }

public double Price { get; set; }

public List<Follower> Followers { get; set; }

}

public class Follower

{

public string Name { get; set; }

public int Age { get; set; }

}

[复制代码](javascript:void(0);)

代码调用

[复制代码](javascript:void(0);)

static void Main()

{

SetConvention();

var db = new DbContext();

var collection = db.Collection<Stock>();

var stocks = new List<Stock>

{

new Stock

{

Symbol = "000001",

Name = "股票1",

Price = 100,

Followers = new List<Follower>

{

new Follower{ Name = "张三", Age = 20 },

new Follower{ Name = "李四", Age = 22 },

new Follower{ Name = "王五", Age = 23 }

}

},

new Stock

{

Symbol = "000002",

Name = "股票2",

Price = 200,

Followers = new List<Follower>

{

new Follower{ Name = "张三", Age = 20 },

new Follower{ Name = "李四", Age = 22 }

}

},

new Stock

{

Symbol = "000003",

Name = "股票3",

Price = 300,

Followers = new List<Follower>

{

new Follower{ Name = "张三", Age = 20 }

}

},

new Stock

{

Id = ObjectId.GenerateNewId(), //这里可以自己设定Id，也可以不设，不设的话操作后会自动分配Id

Symbol = "000004",

Name = "股票4",

Price = 400

}

};

Console.WriteLine("批量插入");

var results = collection.InsertBatch(stocks);

Console.WriteLine(results.Count()); //这里返回的是1，挺奇怪的。

Console.WriteLine();

var stock = new Stock

{

Id = ObjectId.GenerateNewId(), //这里可以自己设定Id，也可以不设，不设的话操作后会自动分配Id

Symbol = "000005",

Name = "股票5",

Price = 500

};

Console.WriteLine("单条插入");

var result = collection.Insert(stock);

Console.WriteLine("插入是否成功：{0}", result.Ok);

Console.WriteLine();

Console.WriteLine("通过Id检索");

var findedStock = collection.FindOneById(BsonValue.Create(stock.Id));

Console.WriteLine("Symbol:{0}, Name:{1}, Price:{2}", findedStock.Symbol, findedStock.Name, findedStock.Price);

Console.WriteLine();

Console.WriteLine("保存操作，库里有数据");

stock.Symbol = "000006";

result = collection.Save(stock);

Console.WriteLine("保存是否成功：{0}", result.Ok);

Console.WriteLine();

Console.WriteLine("删除");

result = collection.Remove(Query<Stock>.EQ(n => n.Id, stock.Id));

Console.WriteLine("删除是否成功：{0}", result.Ok);

Console.WriteLine();

Console.WriteLine("保存操作，库里没数据");

result = collection.Save(stock);

Console.WriteLine("保存是否成功：{0}", result.Ok);

Console.WriteLine();

Console.WriteLine("简单查询");

var list = collection.AsQueryable().Where(n => n.Price >= 300).ToList();

Console.WriteLine("查询结果条数：{0}", list.Count);

Console.WriteLine();

Console.WriteLine("复杂类型查询");

list = collection.AsQueryable().Where(n => n.Followers.Any(f => f.Name == "王五")).ToList();

Console.WriteLine("查询结果条数：{0}", list.Count);

Console.WriteLine();

Console.WriteLine("批量更新");

var query = Query<Stock>.Where(n => n.Price >= 300);

var update = Update<Stock>.Set(n => n.Name, "股票300")

.Set(n => n.Price, 299);

result = collection.Update(query, update, UpdateFlags.Multi);

Console.WriteLine("批量更新是否成功：{0}", result.Ok);

Console.WriteLine();

Console.WriteLine("批量删除");

result = collection.Remove(Query<Stock>.Where(n => n.Price >= 299));

Console.WriteLine("批量删除更新是否成功：{0}", result.Ok);

Console.WriteLine();

Console.WriteLine("删除所有记录");

result = collection.RemoveAll();

Console.WriteLine("删除所有记录是否成功：{0}", result.Ok);

Console.WriteLine();

Console.ReadKey();

}

[复制代码](javascript:void(0);)

全局公约设置

private static void SetConvention()

{

var pack = new ConventionPack {new IgnoreExtraElementsConvention(true), new IgnoreIfNullConvention(true)};

ConventionRegistry.Register("IgnoreExtraElements&IgnoreIfNull", pack, type => true);

}

　　1. IgnoreExtraElementsConvention：忽略库中有但是类中没有定义的字段。这个一般用于敏感字段处理，例如密码字段，它会存在用户Collection中，但是这个字段只是登录校验的时候会用到（这时可以用js来查询），其他用户查询（linq查询）基本都不需要用到密码字段。

　　2. IgnoreIfNullConvention：如果字段null，则不存这个字段，简单来说就是省空间，假设一个类中有A，B两个字段，其中A字段为空，如果指定该设置，存为:{B:'B'}，否则，存为{A:null, B:'B'}。

返回值说明

　　为什么MongoDB提供的API基本都有返回值？那如果API中出现的异常怎么处理，被MongoDB吃掉了？

　　这里我查看了MongoDB的驱动源码，它的结果是通过执行getLastError的方式来获取的，这是c++的方式，C#无法捕捉到这些异常，因此，返回值标识着操作是否成功。同时，API中也会抛出C#的异常或者自定义的异常。这就说明了，操作要满足两个条件才算成功：一、无异常，二、返回值标识成功。

**Repository方式**

　　来到这里，应该有很多人会问，为什么还要用Repository？特别是接触过Entity Framework，因为Entity Framework表明已包含了Unit of Work和Repository，或者是看过[《博客园的大牛们，被你们害惨了，Entity Framework从来都不需要去写Repository设计模式》](http://www.cnblogs.com/leotsai/p/entity-framework-doesnt-need-additional-repository.html" \t "_blank)这类文章的童鞋。

　　首先，我需要表明立场，对于不使用Repository的观点，我是百分之八九十赞同的。那为什么还要用呢？不为什么，我就是任性（开个玩笑）！我认为做技术的不能太偏执，还是要根据具体的场景和需求，技术和框架没有绝对好的，只有相对好的。技术是发展的，但技术不可能面面俱到。

　　那么为什么要用Repository呢？因为我要写单元测试，我需要通过Mock的方式抛开数据库访问的依赖，要Mock的话，要通过接口或虚方法（virtual）。现在的EF 6确实包含了Repository的思想，但是直接用dbContext的话，还是无法Mock（如果可Mock，请告之），因此需要用Repository来包装一下。就好像当你需要测试一个internal的类的时候，你需要定义一个public的类包一下这个内部类。

　　因此，如果你需要写单元测试的话，那么你应该需要Repository，否则，觉得怎么爽就怎么用吧！

通用接口IRepository

[复制代码](javascript:void(0);)

public interface IRepositoryWithTypedId<T, in TId> where T : EntityWithTypedId<TId>

{

IEnumerable<bool> InsertBatch(IEnumerable<T> entities);

bool Insert(T entity);

T Get(TId id);

bool Save(T entity);

bool Delete(TId id);

IQueryable<T> AsQueryable();

bool RemoveAll();

}

public interface IRepository<T> : IRepositoryWithTypedId<T, ObjectId> where T : Entity

{

}

[复制代码](javascript:void(0);)

通用实现MongoRepository

[复制代码](javascript:void(0);)

public class MongoRepositoryWithTypedId<T, TId> : IRepositoryWithTypedId<T, TId> where T : EntityWithTypedId<TId>

{

private readonly MongoCollection<T> \_collection;

public MongoRepositoryWithTypedId()

{

var client = new MongoClient("mongodb://localhost:27017");

var server = client.GetServer();

var db = server.GetDatabase("Temp");

var collectionName = InferCollectionNameFrom();

\_collection = db.GetCollection<T>(collectionName);

}

private string InferCollectionNameFrom()

{

var type = typeof(T);

return type.Name;

}

protected internal MongoCollection<T> Collection

{

get { return \_collection; }

}

public IEnumerable<bool> InsertBatch(IEnumerable<T> entities)

{

var result = Collection.InsertBatch(entities);

return result.Select(n => n.Ok);

}

public bool Insert(T entity)

{

var result = Collection.Insert(entity);

return result.Ok;

}

public T Get(TId id)

{

return Collection.FindOneById(BsonValue.Create(id));

}

public bool Save(T entity)

{

var result = Collection.Save(entity);

return result.Ok;

}

public bool Delete(TId id)

{

var result = Collection.Remove(Query<T>.EQ(t => t.Id, id));

return result.Ok;

}

public IQueryable<T> AsQueryable()

{

return Collection.AsQueryable();

}

public bool RemoveAll()

{

var result = Collection.RemoveAll();

return result.Ok;

}

}

public class MongoRepository<T> : MongoRepositoryWithTypedId<T, ObjectId>, IRepository<T> where T : Entity

{

}

[复制代码](javascript:void(0);)

股票接口IStockRepository

public interface IStockRepository : IRepository<Stock>

{

bool UpdateBatch(double minPrice, string name, double price);

bool DeleteBatch(double minPrice);

}

 注：如果通用方法足够用的话，可不需要自定义接口，直接使用IRepository<T>。 如IRepository<Stock> repository = new MongoRepository<Stock>();

股票接口实现StockRepository

[复制代码](javascript:void(0);)

public class StockRepository : MongoRepository<Stock>, IStockRepository

{

public bool UpdateBatch(double minPrice, string name, double price)

{

var query = Query<Stock>.Where(n => n.Price >= minPrice);

var update = Update<Stock>.Set(n => n.Name, name)

.Set(n => n.Price, price);

var result = Collection.Update(query, update, UpdateFlags.Multi);

return result.Ok;

}

public bool DeleteBatch(double minPrice)

{

var result = Collection.Remove(Query<Stock>.Where(n => n.Price >= minPrice));

return result.Ok;

}

}

[复制代码](javascript:void(0);)

代码调用

[复制代码](javascript:void(0);)

static void Main()

{

SetConvention();

var repository = new StockRepository();

var stocks = new List<Stock>

{

new Stock

{

Symbol = "000001",

Name = "股票1",

Price = 100,

Followers = new List<Follower>

{

new Follower{ Name = "张三", Age = 20 },

new Follower{ Name = "李四", Age = 22 },

new Follower{ Name = "王五", Age = 23 }

}

},

new Stock

{

Symbol = "000002",

Name = "股票2",

Price = 200,

Followers = new List<Follower>

{

new Follower{ Name = "张三", Age = 20 },

new Follower{ Name = "李四", Age = 22 }

}

},

new Stock

{

Symbol = "000003",

Name = "股票3",

Price = 300,

Followers = new List<Follower>

{

new Follower{ Name = "张三", Age = 20 }

}

},

new Stock

{

Id = ObjectId.GenerateNewId(), //这里可以自己设定Id，也可以不设，不设的话操作后会自动分配Id

Symbol = "000004",

Name = "股票4",

Price = 400

}

};

Console.WriteLine("批量插入");

var results = repository.InsertBatch(stocks);

Console.WriteLine(results.Count());

Console.WriteLine();

var stock = new Stock

{

Id = ObjectId.GenerateNewId(), //这里可以自己设定Id，也可以不设，不设的话操作后会自动分配Id

Symbol = "000005",

Name = "股票5",

Price = 500

};

Console.WriteLine("单条插入");

var result = repository.Insert(stock);

Console.WriteLine("插入是否成功：{0}", result);

Console.WriteLine();

Console.WriteLine("通过Id检索");

var findedStock = repository.Get(stock.Id);

Console.WriteLine("Symbol:{0}, Name:{1}, Price:{2}", findedStock.Symbol, findedStock.Name, findedStock.Price);

Console.WriteLine();

Console.WriteLine("保存操作，库里有数据");

stock.Symbol = "000006";

result = repository.Save(stock);

Console.WriteLine("保存是否成功：{0}", result);

Console.WriteLine();

Console.WriteLine("删除");

result = repository.Delete(stock.Id);

Console.WriteLine("删除是否成功：{0}", result);

Console.WriteLine();

Console.WriteLine("保存操作，库里没数据");

result = repository.Save(stock);

Console.WriteLine("保存是否成功：{0}", result);

Console.WriteLine();

Console.WriteLine("简单查询");

var list = repository.AsQueryable().Where(n => n.Price >= 300).ToList();

Console.WriteLine("查询结果条数：{0}", list.Count);

Console.WriteLine();

Console.WriteLine("复杂类型查询");

list = repository.AsQueryable().Where(n => n.Followers.Any(f => f.Name == "王五")).ToList();

Console.WriteLine("查询结果条数：{0}", list.Count);

Console.WriteLine();

Console.WriteLine("批量更新");

result = repository.UpdateBatch(300, "股票300", 299);

Console.WriteLine("批量更新是否成功：{0}", result);

Console.WriteLine();

Console.WriteLine("批量删除");

result = repository.DeleteBatch(299);

Console.WriteLine("批量删除更新是否成功：{0}", result);

Console.WriteLine();

Console.WriteLine("删除所有记录");

result = repository.RemoveAll();

Console.WriteLine("删除所有记录是否成功：{0}", result);

Console.WriteLine();

Console.ReadKey();

}

[复制代码](javascript:void(0);)

 注：我这里没有提供Unit of Work的实现，因为我认为MongoDB对连接的处理比关系型好，当然用Unit of Work的话应该会更好。

**源码下载**

　　下载地址：<https://github.com/ErikXu/MongoDBUsage>