C++ 对象关系映射 (ORM) - 吃面包 - N 的第 1 部分

BrainlessLabs.com，sibani.p

2019 年 4 月 21 日 BSD

在 SQL 数据库之上为 C++ 创建一个简单的 ORM

下载 bun-1.5.0-alpha.zip - 3.4 MB

开放中心

GitHub

介绍

对象关系映射是在面向对象的语言（如 C++）到关系类型系统（如 SQL）之间映射数据类型的过程。那么挑战是什么？C ++有不同类型的原语类型，如的`int`，`char`，`float`，`double`和的变型。因此，将所有这些映射到实际的 SQL 类型是一个真正的挑战。可能有也可能没有与 C++ 类型相似的确切类型。比如说 for `float`，C++ 和 SQL 也可能支持不同类型的标准。所以有不同的工具来完成这项工作。市场上也有很多成熟的库。ODB是非常好的一个。

为了帮助我完成日常工作，我创建了一个名为Bun的简单 C++ 库。

什么是新的?

- *** Bun 1.5.0**将带有**向量**的对象转换为 JSON 和 Msgpack，并从包含向量的 JSON 创建一个对象。**注意：** 它还不包括向量持久性。 它仍在开发中。
- **Bun 1.4.0**支持将对象转换**为 JSON**并**从 JSON**创建对象。它具有将**对象**转换**为消息包**和**从消息包**构造对象的能力。
- **Bun 1.3**支持**对象延迟迭代**和**基于范围的循环**支持。键值存储也支持相同的功能。
- **Bun 1.2**支持嵌入式键值存储。但默认情况下，键值存储基于Unqlite。

特征

- *Easy* 使用
- 使用普通的旧 C++ 对象 (POCO)
- 对象持久化 - 可以`C++ objects`直接持久化
- *Not intrusive* - 您不必修改类以使其持久化
- 普通 C++ 中的约束规范
- 坚持 *Nested Objects*
- *EDSL Object Query Language* （不需要 SQL 查询）

- *Compile time EDSL* 类型安全的语法检查 - 在执行开始之前捕获错误
- 多数据库支持 - SQLite、Postgres、MySQL
- 易于使用的*embedded key-value*商店
- 转换 C++ 对象*to JSON*并创建 C++ 对象*from JSON*。
- 转换 C++ 对象*to Message Pack*并创建 C++ 对象*from Message Pack*。
- STL友好的。这些是常规的 C++ 对象。所以可以在C++ STL算法中使用。

谁在用包子?

本节描述了谁都在使用 Bun 以及在什么情况下使用。如果您发现 Bun 有用并使用它，请告诉我我会在这里添加它。

1. 与 PI 的小冒险。

背景

在我的许多工具应用程序中，我使用 SQLite 作为主数据库。每次使用 SQL 查询时，我都觉得在与我的实际用例没有真正相关的任务上浪费了大量精力。所以我想到了为这些类型的自动映射创建一个框架。图书馆的标准如下：

1. 免费用于任何类型的项目（BSD 许可证）
2. 易于使用（无需 SQL 查询知识）
3. 为字段提供诸如唯一键约束之类的约束
4. 不需要 SQL 查询。EDSL 查询。
5. 非侵入式
6. 富有表现力
7. 应该是 C++ 的 DSL，以便 C++ 编译器可以检查查询语法
8. 无需自定义编译器（C++11 及以上）
9. 高性能
10. 支持多个数据库后端，如 SQLite、Postgres、MySQL
11. 简单的嵌入式键值存储

所有这些直到现在都没有遇到。最终，我将解决所有这些问题。目前，仅开发了该库的基本版本。

使用代码

Bun 对象存储接口

在我们深入了解内部细节之前，在第一篇文章中，让我们看看如何使用该库。

Bun拥有BSD 3-Clause 许可证。它取决于以下开源和免费库：

1. boost（我已经在 1.61 版本上进行了测试，Boost 许可证）
2. fmt（小型、安全、快速的格式化库，BSD 许可证）
3. spdlog（快速 C++ 日志记录，MIT 许可证）
4. SQLite（自包含、无服务器、零配置、事务性 SQL 数据库引擎、公共域）
5. SOCI（C++ 数据库层，BSL 许可）
6. 现代 C++ 的JSON（C++ JSON 和消息包实用程序，MIT许可证）
7. 快速 JSON（快速 C++ JSON 库，请参阅许可证）

GitHub 页面包含所需的所有依赖项。为了便于使用，它还包含一个 Visual Studio 2015 解决方案文件。升压和 SOCI 不包括在内。要下载项目，请将 boost 头文件放在“ *include* ”目录下或更改解决方案文件中的解决方案文件路径。构建 SOCI（使用 cmake 很容易构建）并将库与 Bun 链接。

C++ 缩小▲ 复制代码

```
#include "blib/bun/bun.hpp"

namespace test {
  // Class that needs to be persisted
  struct Person {
```

```
    std::string name;
    std::string uname;
    int age;
    float height;
  };
}

/// @class Child
struct Child {
    int cf1;
    Child(const int cf = -1) : cf1(cf) {}
    Child& operator=(const int i) {
        cf1 = i;
        return *this;
    }
};

/// @class Paret
struct Parent {
    int f1;
    std::string f2;
    // Nested object
    Child f3;
    Parent() :f1(-1), f2("-1"), f3(-1) {}
};

// Both should be persistable
SPECIALIZE_BUN_HELPER((Child, cf1));
SPECIALIZE_BUN_HELPER((Parent, f1, f2, f3));

////////////////////////////////////////////////
/// Generate the database bindings at compile time.
////////////////////////////////////////////////
SPECIALIZE_BUN_HELPER( (test::Person, name, uname, age, height) );

int main() {
  namespace bun = blib::bun;
  namespace query = blib::bun::query;

  // Connect the db. If the db is not there it will be created.
  // It should include the whole path
  // For SQLite
  //bun::connect( "objects.db" );
  // For PostGres
  bun::connect("postgresql://localhost/postgres?user=postgres&password=postgres");
  // Get the fields of the Person. This will be useful in specifying constraints and also
  // querying the object.
  using PersonFields = query::F<test::Person>;

  // Generate the configuration. By default it does nothing.
  blib::bun::Configuration<test::Person> person_config;
  // This is a unique key constraints that is applied.
  // Constraint are applied globally. They need to be set before the
  // execution of the create schema statement
  // The syntax is Field name = Constraint
  // We can club multiple Constraints as below in the same statement.
  // There is no need for multiple set's to be called. This is how
  // We can chain different constraints in the same statement
  person_config.set(PersonFields::name = blib::bun::unique_constraint)
                   (PersonFields::uname = blib::bun::unique_constraint);

  // Create the schema. We can create the schema multiple times. If its already created
  // it will be safely ignored. The constraints are applied to the table.
  // Adding constraints don't have effect if the table is already created
  bun::createSchema<test::Person>();

  // Start transaction
  bun::Transaction t;
  // Create some entries in the database
```

```
  for (int i = 1; i < 1000; ++i) {
    // PRef is a reference to the persistent object.
    // PRef keeps the ownership of the memory. Release the memory when it is destroyed.
    // Internally it holds the object in a unique_ptr
    // PRef also has a oid associated with the object
    bun::PRef<test::Person> p = new test::Person;

    // Assign the members values
    p->age = i + 10;
    p->height = 5.6;
    p->name = fmt::format( "Brainless_{}", i );
    // Persist the object and get a oid for the persisted object.
    const bun::SimpleOID oid = p.persist();

    //Getting the object from db using oid.
    bun::PRef<test::Person> p1( oid );
  }
  // Commit the transaction
  t.commit();

  // To get all the object oids of a particular object.
  // person_oids is a vector of type std::vector<blib::bun<>SimpleOID<test::Person>>
  const auto person_oids = bun::getAllOids<test::Person>();

  // To get the objects of a particular type
  // std::vector<blib::bun::Pref<test::Person>>
  const auto person_objs = bun::getAllObjects<test::Person>();

  // EDSL QUERY LANGUAGE ----------------------
  // Powerful EDSL object query syntax that is checked for syntax at compile time.
  // The compilation fails at the compile time with a message "Syntax error in Bun Query"
  using FromPerson = query::From<test::Person>;
  FromPerson fromPerson;
  // Grammar are checked for validity of syntax at compile time itself.
  // Currently only &&, ||, <, <=, >, >=, ==, != are supported. They have their respective
meaning
  // Below is a valid query grammar
  auto valid_query = PersonFields::age > 10 && PersonFields::name != "Brainless_0";
  std::cout << "Valid Grammar?: " << query::IsValidQuery<decltype(valid_query)>::value <<
std::endl;

  // Oops + is not a valid grammar
  auto invalid_query = PersonFields::age + 10 &&
  PersonFields::name != "Brainless_0";
  std::cout << "Valid Grammar?: " <<
  query::IsValidQuery<decltype(invalid_query)>::value << std::endl;

  // Now let us execute the query.
  // The where function also checks for the validity of the query, and fails at compile time
  const auto objs = fromPerson.where( valid_query ).where( valid_query ).objects();
  // Can even use following way of query
  // As you see we can join queries
  const auto q = PersonFields::age > 21 && PersonFields::name == "test";
  const auto objs_again = FromPerson().where( q ).objects();
  const auto objs_again_q = FromPerson().where( PersonFields::age > 21
  && PersonFields::name == "test" ).objects()
  // Not going to compile if you enable the below line.
  // Will get the "Syntax error in Bun Query" compile time message.
  // const auto objs1 = FromPerson.where( invalid_query ).objects();

  // Check the query generated. It does not give the sql query.
  std::cout << fromPerson.query() << std::endl;

  // Support for Nested object persistence and retrieval
  blib::bun::createSchema<Child>();
  blib::bun::createSchema<Parent>();
  std::cout << "How many objects to insert? " << std::endl;
  int count = 0;
  std::cin >> count;
```

```
  for (int i = 0; i < count; ++i) {
      blib::bun::l().info("===============Start==================");
      blib::bun::PRef<Parent> p = new Parent;
      p->f1 = i;
      p->f2 = i % 2 ? "Delete Me" : "Do not Delete Me";
      p->f3 = 10 * i;
      // Persists the Parent and the Nested Child
      p.persist();
      std::cout << "Added to db: \n" << p.toJson() << std::endl;
      blib::bun::l().info("===============End==================\n");
    }

    std::cout << "Get all objects and show" << std::endl;
    auto parents = blib::bun::getAllObjects<Parent>();
    // Iterate and delete the Parent and the nested Child
    // Here p is a PRef type. We can modify the object and persist
    // the changes if needed.
    for (auto p : parents) {
        std::cout << p.toJson() << std::endl;
        p.del();
    }

  return 0;
}
```

所以这就是我们持久化对象的方式。运行此命令后，将在 SQLite 数据库中创建以下列表：

现在让我们更深入地了解这里的几个元素。该模式的 DDL 如下：

SQL 复制代码

```
CREATE TABLE "test::Person" (object_id INTEGER NOT NULL, name TEXT, age INTEGER, height REAL);
```

此架构由库在内部创建。我只是在这里展示以供参考。

数据如下：

持久存储

类	姓名	年龄	高度
90023498019372	Brainless_1	11	5.6
90023527619226	Brainless_2	12	5.6
90023537497149	Brainless_3	13	5.6
90023553459526	Brainless_4	14	5.6
90023562946990	Brainless_5	15	5.6

基于范围的迭代

Bun 还支持使用 C++ 中基于范围的 for 循环来迭代对象。下面给出了一个简单的例子，说明这是如何工作的。

复制代码

```
    // Iterate the parent with range based for loop
    using FromParents = query::From<Parent>;
```

```
    using ParentFields = query::F<Parent>;
    FromParents from_parents;
    // Select the query which you want to execute
    auto parents_where = from_parents.where(ParentFields::f2 == "Delete Me");
    // Fetch all the objects satisfying the query. This is a lazy fetch. It will be fetched
    // only when it is called. And not all the objects are fetched.
    // Here v is a PRef so it can be used to modify and persist the object.
    for(auto v : parents_where) {
        std::cout << v.toJson() << std::endl;
    }
```

JSON 和消息包转换（To Object 和 From Object）

现在我们可以将 C++ 对象转换为 JSON 并从 JSON 创建 C++ 对象。我们甚至可以将 C++ 对象转换为 Message Pack 并从消息包创建 C++ 对象

它非常容易，只需专门化面包帮手，然后就可以玩儿游戏了。

C++ 缩小▲ 复制代码

```
namespace dbg {
    struct C1 {
        int c1;
        C1() :c1(2) {}
    };

    struct C {
        int c;
        C1 c1;
        C(const int i = 1) :c(i) {}
    };

    struct P {
        std::string p;
        C c;
        P() :p("s1"), c(1) {}
    };
}
SPECIALIZE_BUN_HELPER((dbg::C1, c1));
SPECIALIZE_BUN_HELPER((dbg::C, c, c1));
SPECIALIZE_BUN_HELPER((dbg::P, p, c));

int jsonTest() {
    namespace bun = blib::bun;

    blib::bun::PRef<dbg::P> p = new dbg::P;
    p->p = "s11";
    p->c.c = 10;

    p->c.c1.c1 = 12;


    blib::bun::PRef<dbg::C> c = new dbg::C;
    c->c = 666;
    // Convert the object to JSON
    const std::string json_string = p.toJson();
    // Construct the new object out of JSON
    blib::bun::PRef<dbg::P> p1;
    p1.fromJson(json_string);
    const auto msgpack = p1.toMesssagepack();
    // Construct another object out of messagepack
    blib::bun::PRef<dbg::P> p2;
    p2.fromMessagepack(p1.toMesssagepack());
    // messagepack to string
    std::string msgpack_string;
    for (auto c : msgpack) {
        msgpack_string.push_back(c);
    }
```

```
    std::cout << "1. Original object Object:" << json_string << std::endl;
    std::cout << "2. Object from JSON      :" << p1.toJson() << std::endl;
    std::cout << "3. Object to Messagepack :" << msgpack_string << std::endl;
    std::cout << "4. Object from Messagepck:" << p2.toJson() << std::endl;
    std::cout << "=== Vector JSON Conversion ===" << std::endl;
    blib::bun::PRef<bakery::B> b = new bakery::B;
    b->j = "test";
    b->i.push_back(12);
    b->i.push_back(23);
    std::cout << "5. Object with Vector: " << b.toJson() << std::endl;
    blib::bun::PRef<bakery::B> b1 = new bakery::B;
    b1.fromJson(b.toJson());
    std::cout << "6. Object copy with Vector: " << b1.toJson();
    return 1;
}
```

键值存储

Bun 有一个嵌入式键值存储。默认实现基于 Unqlite。

C++ 缩小▲ 复制代码

```
/// @class KVDb
/// @brief The main class for the key value store
template<typename T = DBKVStoreUnqlite>
class KVDb {
public:
    /// @fn KVDb
    /// @param param
    /// @brief The constructor for the KV class
    KVDb(std::string const& param);

    /// @fn KVDb
    /// @param other. The other KVDb from which we can copy values.
    /// @brief The copy constructor for the KV class
    KVDb(KVDb const& other);

    /// @fn ~KVDb
    /// @brief destructor for the KV class
    ~KVDb();

    /// @fn ok
    /// @brief Returns Ok
    bool ok() const;

    std::string last_status() const;

    /// @fn put
    /// @param key The key
    /// @param value the value that needs to be stored
    /// @details Put stores the key and value and returns true of the store is done,
    /// else it returns false
    ///          All primary C++ data types including std::string is supported as key and value
    template<typename Key, typename Value>
    bool put(Key const& key, Value const& value);

    /// @fn get
    /// @param key The key
    /// @param value the value is of type ByteVctorType. This carries the out value
    /// @details Gets the value corresponding the key.
    /// If the retrieval it returns true else it returns false.
    ///          All primary C++ data types including std::string is supported as key.
    ///          The value is a byte (std::uint8_t) value
    template<typename Key>
    bool get(Key const& key, ByteVctorType& value);

    /// @fn get
```

```
    /// @param key The key
    /// @param value the value is of type ByteVctorType. This carries the out value
    /// @details Gets the value corresponding the key. If the retrieval it returns true
    /// else it returns false.
    ///          All primary C++ data types including std::string is supported as key.
    ///          The value C++ primary datatype.
    ///          This function is a wrapper on top of the previous function
    ///          which returns the byte vector.
    template<typename Key, typename Value>
    bool get(Key const& key, Value& value);

    /// @fn del
    /// @param key The key
    /// @details Delete the value corresponding to key.
    /// If delete is success then returns true else returns false.
    ///          All primary C++ data types including std::string is supported as key.
    template<typename Key>
    bool del(Key const& key);
};
```

以下是我们可以使用它的方式：

C++ 缩小▲ 复制代码

```
/// @fn kvTest
/// @brief A test program for
int kvTest() {
    /// @var db
    /// @brief Create the database. If the database already exists
    /// it opens the database but creates if it doesnt exist
    blib::bun::KVDb<> db("kv.db");
    /// @brief put a value in database.
    db.put("test", "test");
    std::string val;
    /// @brief get the value. We need to pass a variable by reference to get the value.
    db.get("test", val);
    std::cout << val << std::endl;

    const int size = 10000;
    for (int i = 0; i < size; ++i) {
        const std::string s = fmt::format("Value: {}", i);
        db.put(i, s);
    }

    for (int i = 0; i < size; ++i) {
        std::string val;
        db.get(i, val);
        std::cout << val << std::endl;
    }

    return 1;
}
```

键值的基于范围的迭代

Bun 支持 kv 存储中元素的键值的基于范围的迭代。这个迭代就像地图的迭代。键和值都成对返回。如果您看到下面kv是一对，则kv.first携带键值和kv.second携带值。的kv.first 和kv.second具有值作为一个**字节的矢量**。

缩小▲ 复制代码

```
    // ========= KV Store
    blib::bun::KVDb<> db("kv.db");

    const int size = 3;
    for (int i = 0; i < size; ++i) {
        const std::string s = fmt::format("storing number: {}", i);
        db.put(i, s);
```

```
    }

    std::cout << "Start iteration Via size "<< std::endl;
    for (int i = 0; i < size; ++i) {
        std::string val;
        db.get(i, val);
        std::cout << val << std::endl;
    }

    std::cout << "Start iteration via foreach "<< std::endl;
    count = 0;
    // Iterate the key value store using foreach.
    // We have both the key and value here. So we can change the value at the key
    for (auto kv : db) {
        int key = 0;
        blib::bun::from_byte_vec(kv.first, key);

        std::string value;
        blib::bun::from_byte_vec(kv.second, value);
        std::cout << count++ << ")> key: "<< key << "\n Value: " << value << std::endl;
    }
```

内件

ORM 的一些内部结构如下。

反射

Bun 在内部使用简单的反射来生成照顾**编译时的类型**信息。有一个计划将它扩展一点，以便它更有用。

SPECIALIZE_BUN_HELPER

该宏将在编译时生成对象的所有绑定。所有模板特化都是使用这个宏创建的。在多个头文件或 CPP 文件中使用宏应该是安全的。

应将以下内容传递给宏：

（<类名，也应包括命名空间详细信息>，要持久化的成员...）

成员列表也可以是部分类成员。假设我们有一个我们使用的对象的句柄，没有必要将它存储在数据库中。在这种情况下，我们可以省略句柄并保留所有其他功能。这样，只会填充给定的字段。

约束

在**Bun 中**应用约束很容易。下面的例子解释了它。

C++ 复制代码

```
// Get the fields of the Person. This will be useful in specifying constraints and also
// querying the object.
using PersonFields = query::F<test::Person>;

// Generate the configuration. By default it does nothing.
blib::bun::Configuration<test::Person> person_config;
// This is a unique key constrains thats applied.
// Constraint are applied globally. They need to be set before the
// execution of the create schema statement
// The syntax is Field name = Constraint
// Here is how we can chain the different constraints in a single set statement
person_config.set(PersonFields::name = blib::bun::unique_constraint)
                 (PersonFields::uname = blib::bun::unique_constraint);
```

如您所见，创建唯一约束非常容易。如上所述，我们可以使用重载**()**运算符将多个约束组合在一起，而不是多次调用 set。

要记住的事情：

- 目前，只能在创建表之前应用约束。创建表后，这些语句不起作用。
- 支持唯一的唯一键。

在后续版本中，我将删除这些限制。

参考

`PRef`是图书馆的核心元素之一。它持有需要持久化的对象。它还包含`oid`对象的，它独立于实际对象。使对象持久化的几条规则：

- 需要持久化的成员必须是`public`.
- `PRef` 维护对象的所有权，并在超出范围时删除对象。
- 如果我们将 a 分配`PRef`给另一个，那么`PRef`前者将失去对象的所有权。就像一个`unique_ptr`. 实际上，`PRef`将对象存储在`unique_ptr`下面。
- 在持久化对象之前，我们必须创建模式（使用`blib::bun::createSchema<>()`）并生成绑定（`using SPECIALIZE_BUN_HELPER( (test::Person, name, age, height) );`）
- 它还包含特定实例中对象的 md5 总和。所以如果对象没有变化，那么它就不会持久化它。我自己使用它，我保留了更新的时间戳。我不想每次都更新对象。对于这次公开发布，我省略了时间戳。

插入或更新

库如何知道我们是要插入还是更新数据库？这发生在对象的 md5 上。如果 md5 有一些价值，那么它就是一个`update`else 它是一个`insert`. 为 自动生成以下查询`insert`：

SQL 复制代码

```
INSERT INTO 'test::Person' (object_id,name,age,height)
VALUES(91340162041484,'Brainless_4',14,5.6)
```

搜索

在 Bun 中搜索非常容易。有不同的搜索机制。

- **Oid Search**：我们可以`Oids`使用以下方法获取所有信息：

  C++ 复制代码

  ```
  // The return type is std::vector<blib::bun<SimpleOID<test::Person>>
  const auto person_oids = blib::bun::getAllOids<test::Person>();
  ```

- **搜索某个类型的所有对象**：我们可以将数据库中的所有对象作为对象向量获取：

  C++ 复制代码

  ```
  // std::vector<blib::bun::Pref<test::Person>>
  const auto person_objs = blib::bun::getAllObjects<test::Person>();
  ```

- **Object EDSL**：我们可以通过 Bun 提供的 EDSL 查询进行搜索。EDSL 是使用 boost proto 库实现的。C++ 编译器在编译时检查查询。当被调用时，它会创建一些特殊的变量。 `SPECIALIZE_BUN_HELPER`

  例如：对于`Person` 类，`SPECIALIZE_BUN_HELPER` 生成以下内容：

  C++ 复制代码

  ```
  bun::query::F<test::Person>::name
  bun::query::F<test::Person>::age
  bun::query::F<test::Person>::heigh
  ```

`bun::query::F`Bun的类将专门用于类的所有领域`Person`。

要应用任何类型的过滤器，您只需要使用“`where`”函数，例如：

C++ 复制代码

```
// The where(Query) is a lazy function, it does not query the db.
// The actual execution is done in the object() function
const auto objs_again = bun::query::From<test::Person>().where( valid_query ).objects();
// We can also join queries or filters using && or the || operator
const auto objs_again = bun::query::From<test::Person>().where( valid_query && valid_query
).objects();
```

论坛

- Gitter.im：如果我们有空，您可以在这里提问以获得更快的答案或与我们聊天
- Github 问题：在此处创建问题

历史

- α1（16 日 2016日）
  - 库的初始版本
- 阿尔法2（2 次 2016七月）
  - 实施 Bun EDSL
- Alpha 3（2018 年 3 月14 日）：
  - 集成 SOCI 作为数据库交互层。这使得库可以使用任何 SQL 数据库作为 SQLite、Postgres、MySQL。它主要支持 SOCI 支持的其他数据库，但尚未经过测试。
  - 使用Boost Fusion。代码更简洁，预处理器宏更少 代码更易于调试。
  - 支持使用`Transaction`类进行事务处理
  - 更好的错误处理和错误记录
  - 添加了大量评论以帮助用户
- Alpha 4（2018 年 3 月5 日）
  - 支持嵌套对象
  - `SimpleOID` 现在使用 boost UUID 生成唯一标识符
  - 补充评论
  - 小的性能增强
- Alpha 5（2018 年 5 月19 日）
  - 支持创建表前的约束
- Alpha 6（2018 年 7 月 18 日）：
  - 向 bun 添加键值功能
- Alpha 7（2018 年 8 月 11 日）：
  - 为对象迭代添加了基于范围的 for 循环支持。
  - 为键值存储迭代添加了基于范围的 for 循环支持。
  - 两次迭代都是惰性迭代。
- Alpha 8（2018 年 10 月 19 日）
  - 添加了从 JSON 字符串创建 C++ 对象的支持
  - 添加了从 C++ 对象创建消息包的支持
  - 添加了从消息包创建 C++ 对象的支持
- Alpha 9（2018 年 1 月 13 日）
  - 添加了将包含向量的 C++ 对象转换为 JSON 和 Msgpack 的支持
  - 添加了将包含向量的 JSON 或 Msgpack 转换为 C++ 对象的支持。

下一个功能

- **添加 C++ 向量持久性**
- ~~基于迭代器的惰性数据拉取~~
- 自定义`Oid`类支持
- 支持弹性搜索
- 改进的错误处理
- EDSL 查询语言增强
- 创建表后的约束修改
- 支持其他约束
- 索引支持
- 支持处理对象的前后钩子
- 永久`std::vector`会员
- 单元测试实现
- 支持 Leveldb
- ~~键值迭代器~~
- ~~支持复合类型~~。（完毕）

需要帮助

大家好，
考虑到使这个库进一步丰富所需的工作，我将需要任何帮助。在以下方面需要帮助。

1. 增强
2. 修复错误。
3. 重构和清理代码。
4. 增强文档。
5. 建设性的批评和特色建议。
6. 编写测试。
7. 使用包子

任何小事或大事都值得赞赏。

本文最初发表于https://github.com/BrainlessLabs/bun

执照

分享

关于作者

BrainlessLabs.com
建筑师
印度
手表
该会员

我喜欢探索技术的不同方面。尝试新事物，并获得快乐。我的兴趣是编程语言和成像。但在其他事情上工作也不难。算法让我在喝咖啡休息时感到高兴。

我基本上用 C++ 编写代码，但 JAVA 对我来说并不陌生。我也知道很少的脚本语言。基本上我觉得知道一门编程语言只是一个无...

展示更多

西巴尼
软件开发人员
印度

手表该会员

专业的软件工程师对了解不同的技术、编程语言、科学的最新趋势非常好奇。我也花一些时间在网站设计和开发。人工智能、遗传编程、计算机图形是我喜欢探索的一些领域。

评论和讨论

添加评论或问题 | 电子邮件提醒 | Search Comments

First Prev Next

Help Needed!!!!
BrainlessLabs.com 9-Dec-18 1:20

My vote of 5
koothkeeper 15-Aug-18 5:06

Re: My vote of 5
BrainlessLabs.com 16-Aug-18 14:49

Bun now supports Range based for loop and lazy iteration.
BrainlessLabs.com 13-Aug-18 18:27

Why is this categorised under NoSQL?
Zebedee Mason 2-Aug-18 20:52

Re: Why is this categorised under NoSQL?
BrainlessLabs.com 13-Aug-18 18:28

Bun is getting a new db engine
BrainlessLabs.com 6-Aug-16 18:24

Re: Bun is getting a new db engine
BrainlessLabs.com 14-Mar-18 18:13

looks good
Taulie 26-Jul-16 16:43

Re: looks good
BrainlessLabs.com 30-Jul-16 1:44

Please Vote
BrainlessLabs.com 10-Jul-16 21:12

Future Releases
BrainlessLabs.com 10-Jul-16 4:07

What is the next feature that you want to see?
BrainlessLabs.com 17-May-16 20:47

Re: What is the next feature that you want to see?
Nathan Going 18-May-16 4:09

Re: What is the next feature that you want to see?
BrainlessLabs.com 18-May-16 14:56

Re: What is the next feature that you want to see?
BrainlessLabs.com 13-Aug-18 18:25

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。