CPP-Summit

2023全球C++及系统软件技术大会

基于模板元编程的C++序列化库：struct_pack

李泽政

阿里云-基础软件部

目录

01

struct_pack简介

struct_pack简介

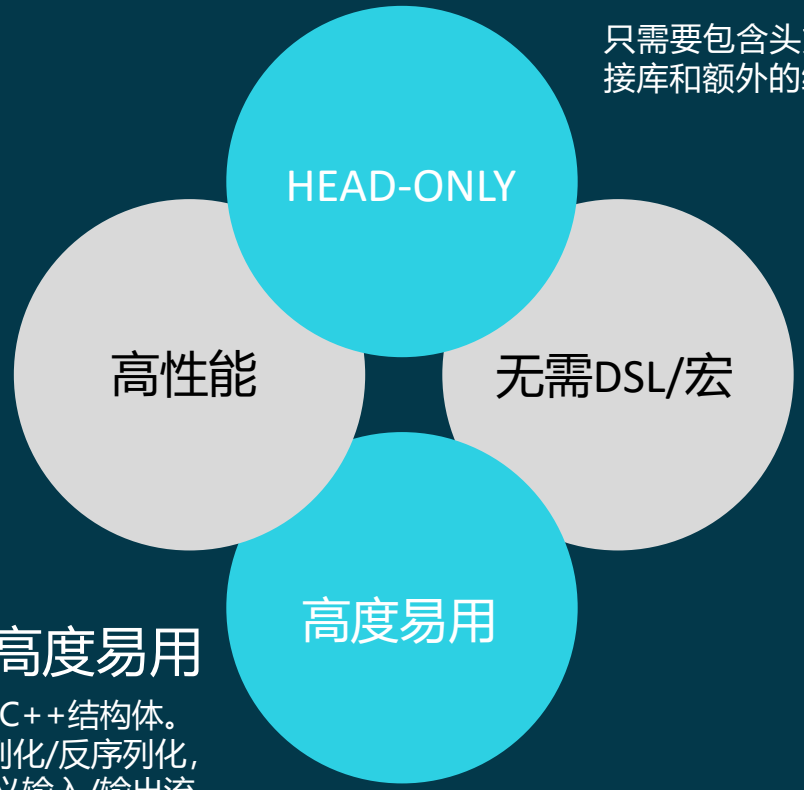
世界上已经有了很多开源的二进制序列化库:
Protobuf/MsgPack/FlatBuffer……

为什么我们要”再”发明一个轮子?

struct_pack的优势

HEAD-ONLY

只需要包含头文件即可使用，无需动态链接库和额外的编译步骤，不依赖代码生成。



高性能

基于编译期计算的高效类型检查，允许平凡拷贝甚至“零拷贝”优化。

无需DSL/宏

STRUCT_PACK不依赖自定义DSL，可以直接解析原生的C++结构体。最大程度降低用户心智负担。

高度易用

非侵入式设计，直接序列化C++结构体。一行代码完成C++结构体的序列化/反序列化，支持STL/第三方库·，支持自定义输入/输出流

struct_pack简介

```cpp
#include <ylt/struct_pack.hpp>
// head-only
struct person {
  int age;
  std::string name;
  bool operator==(const person&) const = default;
};
//可直接序列化C++原生结构体
void test(const person& p) {
  // 一行代码完成序列化/反序列化
  auto buffer = struct_pack::serialize(p);
  auto result = struct_pack::deserialize<person>(buffer);
  assert(result.has_value());
  // 错误检查（类型不匹配，版本不兼容，buffer不完整……）
  assert(result == p);
}
```

struct_pack简介

```cpp
// 将序列化结果保存到用户提供的容器
{
    auto buffer = struct_pack::serialize<boost::container::string>();
}
// 将序列化结果附加到已有的容器中
{
  std::string buffer = "The next line is struct_pack data.\n";
  struct_pack::serialize_to(buffer, p);
}
// 将序列化结果写入到输出流中
{
  std::ofstream writer("struct_pack_demo.data",
                       std::ofstream::out | std::ofstream::binary);
  struct_pack::serialize_to(writer, p);
}
```

struct_pack简介

```cpp
// 反序列化到已有的对象
{
  person p;
  std::string buffer;
  auto error_code = struct_pack::deserialize_to(p, buffer);
  assert(error_code == struct_pack::errc{});
}
// 从流中反序列化对象
{
  std::ifstream reader("struct_pack_demo.data",
                       std::ifstream::in | std::ifstream::binary);
  auto result = struct_pack::deserialize(reader);
  assert(result.has_value());
}
```

02

静态反射

反射的基本概念

什么是反射?

动态反射 vs 静态反射

为什么序列化库需要“反射”？

姗姗来迟的C++静态反射：C++17 -> C++20 -> C++23 -> C++26 ...

Protobuf：依赖自定义DSL/工具链生成代码

```
message Vec3 {
    float x = 1;
    float y = 2;
    float z = 3;
}
message Weapon {
    string name = 1;
    int32 damage = 2;
}
message Monster {
  Vec3 pos = 1;
  int32 mana = 2;
  int32 hp = 3;
  string name = 4;
  bytes inventory = 5;
  enum Color {
        Red = 0;
        Green = 1;
        Blue = 2;
  }
  Color color = 6;
  repeated Weapon weapons = 7;
  Weapon equipped = 8;
  repeated Vec3 path = 9;
}
```

Monster.proto

protoc

```
Weapon::Weapon(::PROTOBUF_NAMESPACE_ID::Arena*
arena,
                         bool is_message_owned)
  : ::PROTOBUF_NAMESPACE_ID::Message(arena,
is_message_owned) {
  SharedCtor(arena, is_message_owned);
  //
@@protoc_insertion_point(arena_constructor:mygame
.Weapon)
}
Weapon::Weapon(const Weapon& from)
  : ::PROTOBUF_NAMESPACE_ID::Message() {
  Weapon* const _this = this; (void)_this;
  new (&_impl_) Impl_{
      decltype(_impl_.name_){}
    , decltype(_impl_.damage_){}
    , /*decltype(_impl_._cached_size_)*/{}};

  _internal_metadata_.MergeFrom<::PROTOBUF_NAMESP
ACE_ID::UnknownFieldSet>(from._internal_metadata_
);
  _impl_.name_.InitDefault();
  #ifdef PROTOBUF_FORCE_COPY_DEFAULT_STRING
    _impl_.name_.Set("", GetArenaForAllocation());
  #endif // PROTOBUF_FORCE_COPY_DEFAULT_STRING
  if (!from._internal_name().empty()) {
    _this->_impl_.name_.Set(from._internal_name(),
      _this->GetArenaForAllocation());
  }
  _this->_impl_.damage_ = from._impl_.damage_;
  //
@@protoc_insertion_point(copy_constructor:mygame.
Weapon)
}
```

Monster.hpp & Monster.cpp

```
auto m = Monsters.add_monsters();
auto vec = new mygame::Vec3;
vec->set_x(1);
vec->set_y(2);
vec->set_z(3);
m->set_allocated_pos(vec);
m->set_mana(16);
m->set_hp(24);
m->set_name("it is a test");
m->set_inventory("\1\2\3\4");
m->set_color(::mygame::Monster_Color::Red);
auto w1 = m->add_weapons();
w1->set_name("gun");
w1->set_damage(42);
auto w2 = m->add_weapons();
w1->set_name("mission");
w1->set_damage(56);
auto w3 = new mygame::Weapon;
w3->set_name("air craft");
w3->set_damage(67);
m->set_allocated_equipped(w3);
auto p1 = m->add_path();
p1->set_x(7);
p1->set_y(8);
p1->set_z(9);

auto buffer = m.SerializeAsString();
```

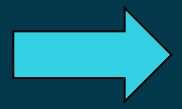
Init & Serialize

Msgpack：基于宏/预处理器的静态反射

```cpp
struct Monster {
  Vec3 pos;
  int16_t mana;
  int16_t hp;
  std::string name;
  std::vector<uint8_t> inventory;
  Color color;
  std::vector<Weapon> weapons;
  Weapon equipped;
  std::vector<Vec3> path;

  MSGPACK_DEFINE(pos, mana, hp,
name, inventory, (int &)color,
weapons, equipped, path);
};
```

预处理器



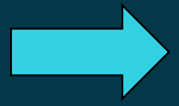
```cpp
#define MSGPACK_DEFINE_ARRAY(...) \
    template <typename Packer> \
    void msgpack_pack(Packer& msgpack_pk) const \
    { \
        msgpack::type::make_define_array(__VA_ARGS__)
.msgpack_pack(msgpack_pk); \
    } \
    void msgpack_unpack(msgpack::object const&
msgpack_o) \
    { \
        msgpack::type::make_define_array(__VA_ARGS__)
.msgpack_unpack(msgpack_o); \
    }\
    template <typename MSGPACK_OBJECT> \
    void msgpack_object(MSGPACK_OBJECT* msgpack_o,
msgpack::zone& msgpack_z) const \
    { \
        msgpack::type::make_define_array(__VA_ARGS__)
.msgpack_object(msgpack_o, msgpack_z); \
    }
```

struct_pack：基于模板元编程的静态反射

当对象是聚合类型时：

```
struct Monster {
  Vec3 pos;
  int16_t mana;
  int16_t hp;
  std::string name;
  std::vector<uint8_t> inventory;
  Color color;
  std::vector<Weapon> weapons;
  Weapon equipped;
  std::vector<Vec3> path;
}
```

模板元编程



自动生成字段的元信息：

Monster有多少个字段?

Monster各字段的类型是什么?

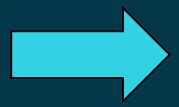
如何读写Monster的字段?

如何知道结构体有几个字段?

(C++11) Aggregate initialization:

```
struct person {
  int32_t id;
  std::string name;
};

person p0{};           //OK
person p1{1};          //OK
person p2{1,"name"};   //OK
person p3{1,"name",0};//Compile error
```



```
struct Any {
  template <typename T>
  operator T();
};

person p0{};                        // OK
person p1{Any{}};                   // OK
person p2{Any{}, Any{}};            // OK
person p3{Any{}, Any{}, Any{}};// Compile error;
```

在聚合初始化时，不断增加Any{}的个数，直到模板实例化失败为止。
能填入的最多的参数数目，就是结构体字段的数量

如何获取结构体各成员的类型并读写?

(C++17) structured binding:

```
{
  person p{.age = 24, .name = "student"};
  auto &&[e0, e1] = p;
  // e0==p.age, e1==p.name;
}
{
  std::pair<int, double> p;
  auto &&[e0, e1] = p;
  // e0==p.first, e1=p.second;
}
{
  std::array<int, 3> ar;
  auto &&[e0, e1, e2] = ar;
  // e0==ar[0], e1==ar[1], e2==ar[2];
}
```

只要知道结构体的字段数量，就能通过结构化绑定获取成员的引用。

struct_pack：用户自定义反射（基于宏）

为什么要支持用户自定义反射?

1.用户只想序列化部分字段

```
struct account {
  uint64_t ID;
  std::string name;
  std::string password;
};
STRUCT_PACK_REFL(account, ID, name);
```

struct_pack：用户自定义反射（基于宏）

为什么要支持用户自定义反射?

2.用户希望读写private字段

```
class person {
  int age;
  std::string name;
  STRUCT_PACK_FRIEND_DECL(person);
};
STRUCT_PACK_REFL(person, age, name);
```

struct_pack：用户自定义反射（基于宏）

为什么要支持用户自定义反射?

3.用户提供的类型不是聚合类

```
struct my_person : public person {
  double salary;
};
STRUCT_PACK_REFL(my_person, age, name, salary);
```

03

类型系统与类型校验

类型系统与类型校验

C++具有极其复杂的类型系统：

我们需要支持基本类型/STL的数据结构/嵌套类型

我们还需要支持第三方库的数据结构和用户自定义的数据结构

反序列化时如何进行类型校验？

类型系统

我们需要将C++的类型映射到struct_pack：

分类	描述
基本类型	整数，浮点数，bool，字符类型
约束类型	string, array, container, map, set, optional……
聚合类型	struct/class, pair, tuple

类型系统

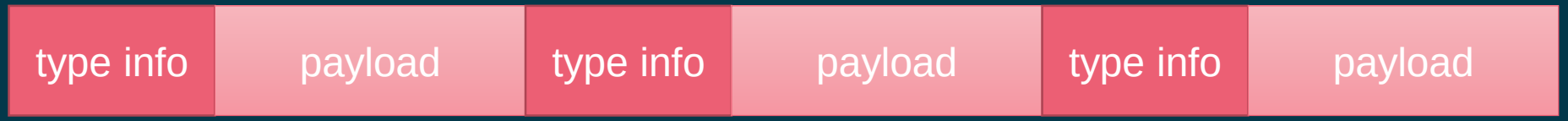
约束类型：等价于一个特定的concept约束。
即使是第三方库/用户自定义类型，只要满足条件即属于对应的约束类型

```
template <typename Type>
concept string = requires(Type container) {
  container.size();
  container.begin();
  container.end();
  container.length();
};
```

满足上述约束即为字符串类型，如：
std::string/std::wstring/boost::container::string/folly::string……

类型校验

传统的类型校验方式：

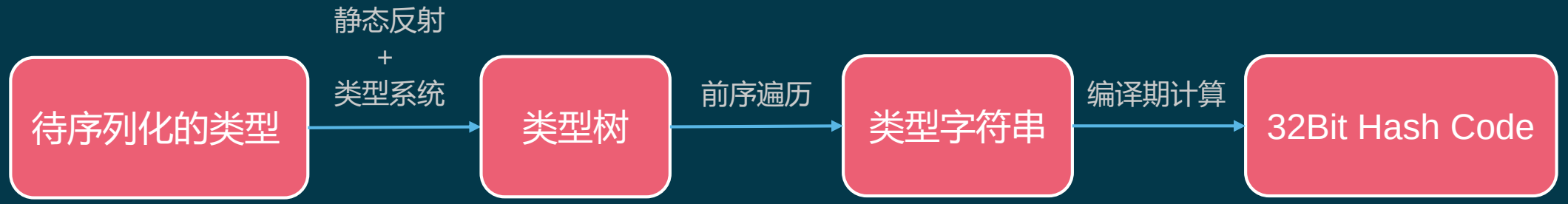


struct_pack：基于编译期计算的高性能类型检查：体积小&高效

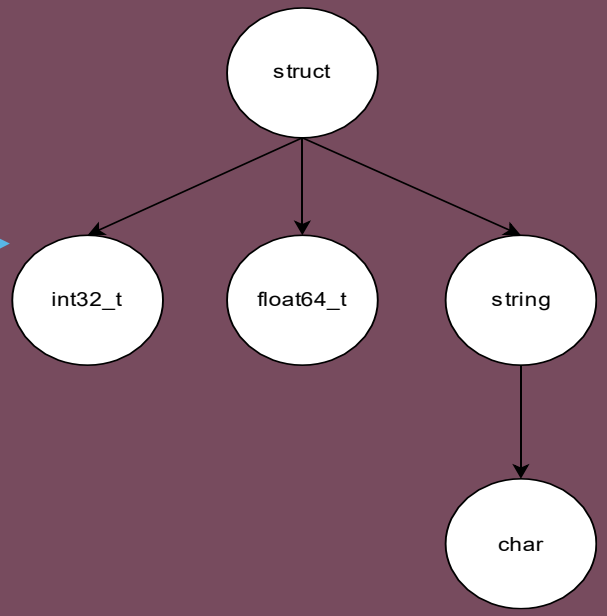
运行时代价非常低：只需要读取头4个字节并比较即可。

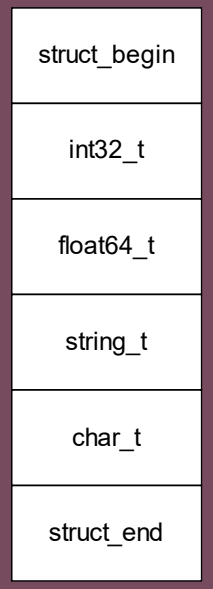
类型校验

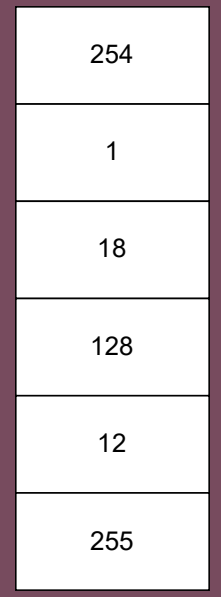
通过C++强大的编译期计算能力，struct_pack可在编译期完成哈希计算：




```
struct dummy {
  int32_t id;
  double number;
  basic_string<char>
    str;
};
```







0xb1f36258

类型校验

哈希校验自然可能存在哈希冲突。

用户不小心写错类型且通过哈希校验的概率为2^(-32)，几乎可以忽略不计。

尽管如此：在Debug模式下struct_pack会在元信息中包含完整的类型字符串，以缓解哈希冲突。

用户自定义序列化

有些第三方库类型难以映射到已有的struct_pack类型。如何和struct_pack适配?

```
struct array2D {
  unsigned int x;
  unsigned int y;
  float* p;
};
```

struct_pack可通过ADL查找到用户自定义的序列化函数。

```
std::size_t sp_get_needed_size(const array2D& ar);
template <struct_pack::writer_t Writer>
void sp_serialize_to(Writer& writer, const array2D& ar);
template <struct_pack::reader_t Reader>
struct_pack::errc sp_deserialize_to(Reader& reader, array2D& ar);
```

类型校验字符串：默认情况下会将该类型的名字拼接到字符串中。

04

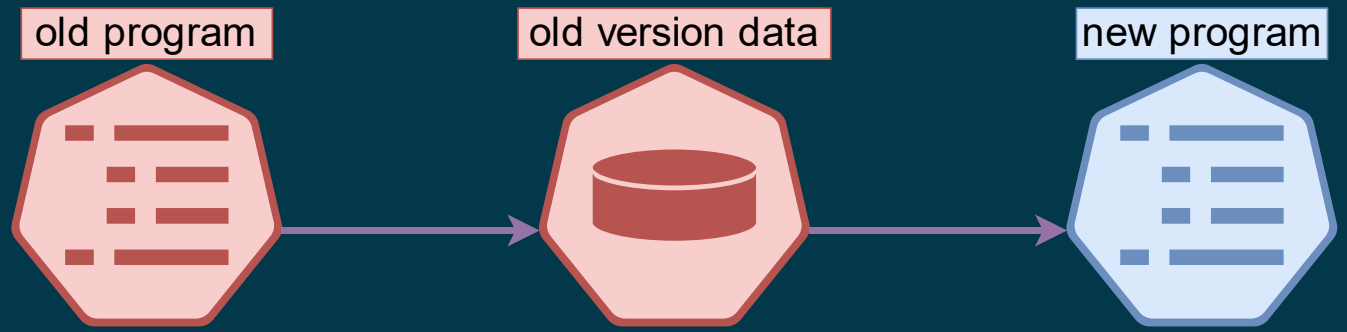
协议兼容性

向前/向后兼容

在实际业务中，用户可能不断迭代更新协议，这就要求序列化库必须具有前向/后向兼容的能力。

向前兼容：
旧版本的程序可以正常解析新版本的数据

向后兼容：
新版本的程序可以正常解析旧版本的数据



compatible字段

struct_pack通过compatible字段来实现向前/向后兼容。

限制：1. 升级协议时，可新增struct_pack::compatible<T, version> 类型。
2. 应保证新增字段的版本号大于旧版协议的版本号。
3. 只允许新增字段，不允许删除/修改原有的字段

```
enum Color { RED, BLUE, GREEN };

struct rect_v0 {
  float x, y, height, width;
};

struct GameObject_V0 {
  uint64_t ID;
  std::vector<rect_v0> rects;
};
```

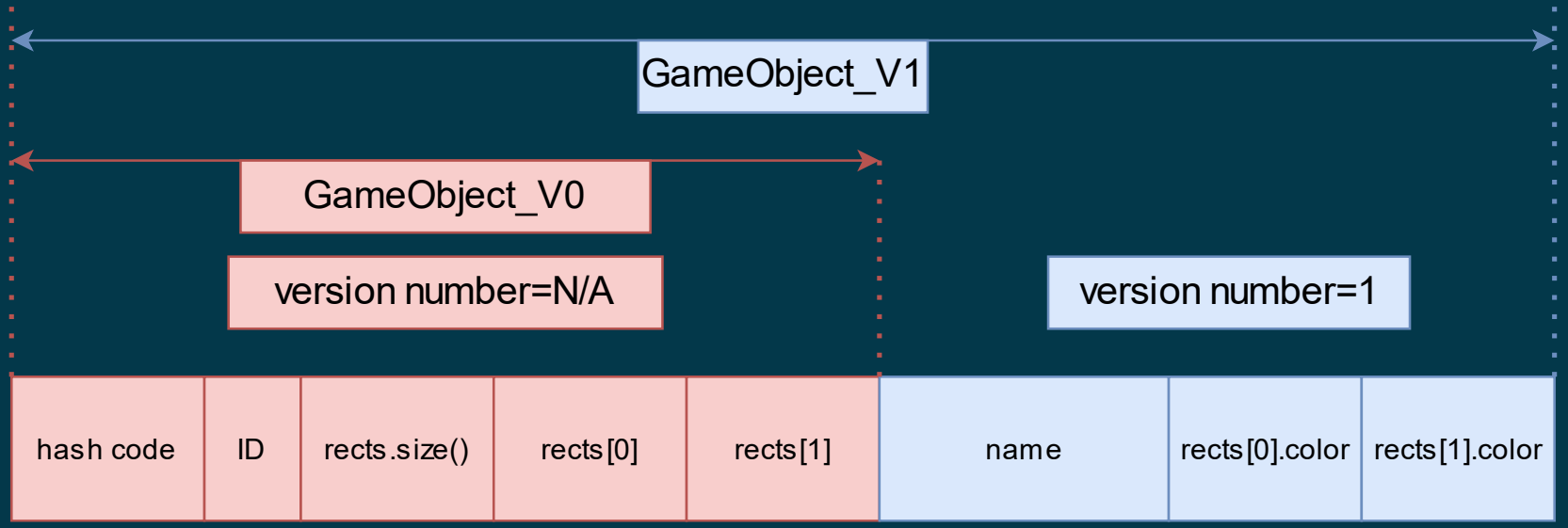


```
struct rect_v1 {
  struct_pack::compatible<Color, 1> color;
  float x, y, height, width;
};

struct GameObject_V1 {
  uint64_t ID;
  struct_pack::compatible<string, 1> name;
  std::vector<rect_v1> rects;
};
```

如何实现向前/向后兼容

1.生成类型字符串时跳过compatible字段（从而保证hash校验码相同）
2.各字段的内存布局按版本号排布（在编译期对版本号进行排序实现），
从而保证旧版数据一定是新版数据的一个合法前缀。



向前兼容：程序正常反序列化所有字段，忽略末尾未知的新字段。
向后兼容：程序正常反序列化所有旧字段，将不存在的新字段设为空值。

05

平凡拷贝与零拷贝优化

平凡布局

如果一段数据，其序列化布局和其在内存中的布局完全相同，我们就称该布局是平凡的。

在struct_pack中，哪些对象的布局是平凡的?

1. 基本类型
2. 数组类型(T[sz], std::array<T,sz>)，且数组的元素T是平凡的
3. 结构体类型(class/struct/std::pair), 且其所有的字段都是平凡的。

平凡拷贝

对于平凡布局的数据，我们可以直接通过memcpy/write函数进行序列化，从而大大提升序列化性能。

此外，当我们序列化一个内存布局连续的容器(std::string/std::vector<T>)时，如果元素是平凡的，那么我们也可以直接通过memcpy/write一次性序列化整个容器的所有元素。

```
struct rect {
  float x, y, height, width;
};

struct GameObject {
  uint64_t ID;
  std::vector<rect> rects;
  //only once memcpy when serialize the content of rects
};
```

零拷贝

能平凡拷贝就能不拷贝：通过string_view/span等工具:

```
struct rect {
  float x, y, height, width;
};

struct GameObject {
  std::string_view name;
  std::span<rect> rects;
};

void serialize(const std::string& name, const std::vector<rect>& rects) {
  GameObject o{name, rects};                        // zero-copy when init
  auto buffer = struct_pack::serialize(o);          // once copy when serialize
  auto result = struct_pack::deserialize(buffer);   // zero-copy when deserialize
}
```

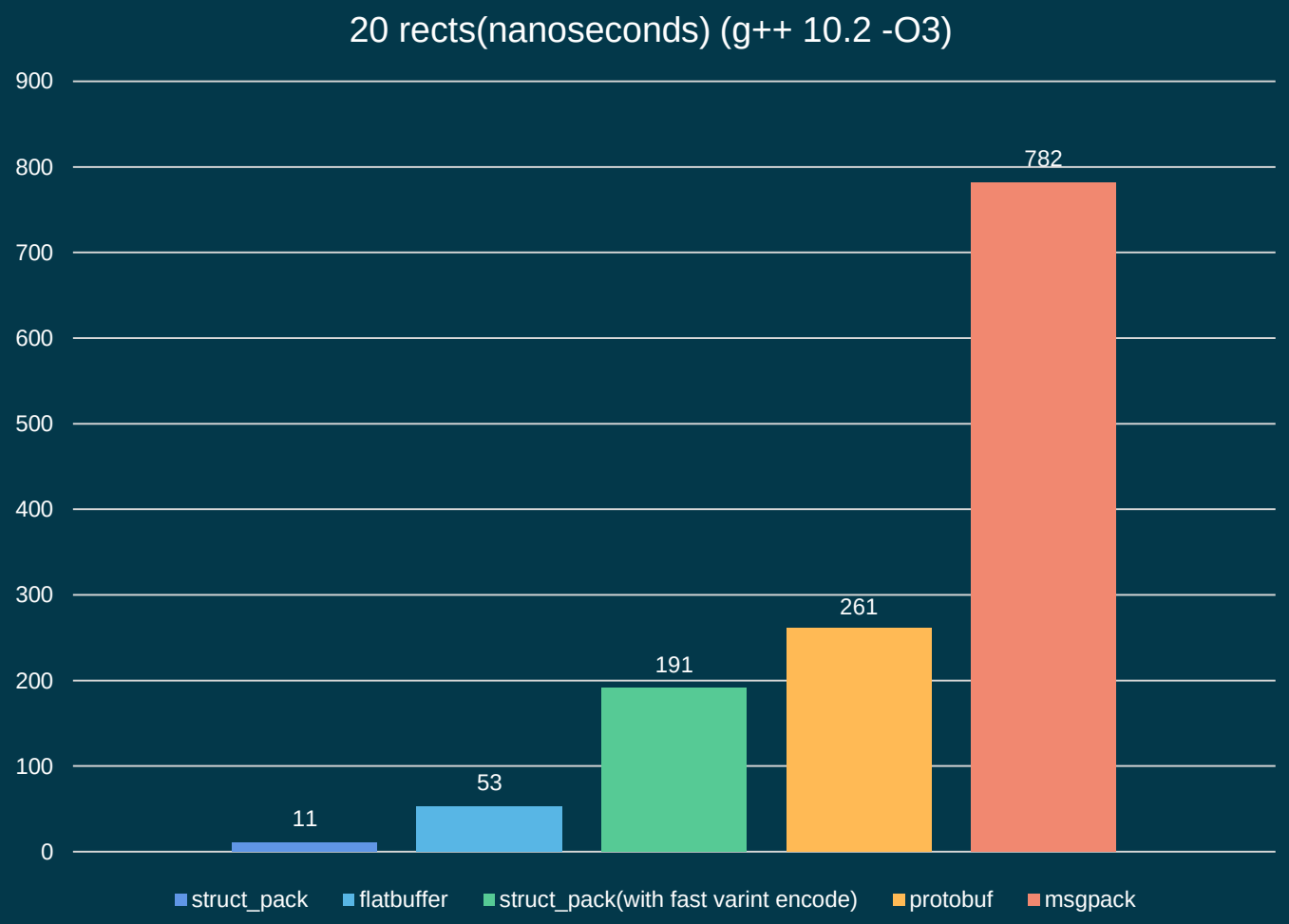
06

Benchmark

序列化性能

struct_pack的整数序列化性能:
平凡编码下快于flatbuffer
变长编码下快于protobuf和msgpack

```
struct rect {
  int32_t x = 1;
  int32_t y = 0;
  int32_t width = 11;
  int32_t height = 1;
};
```

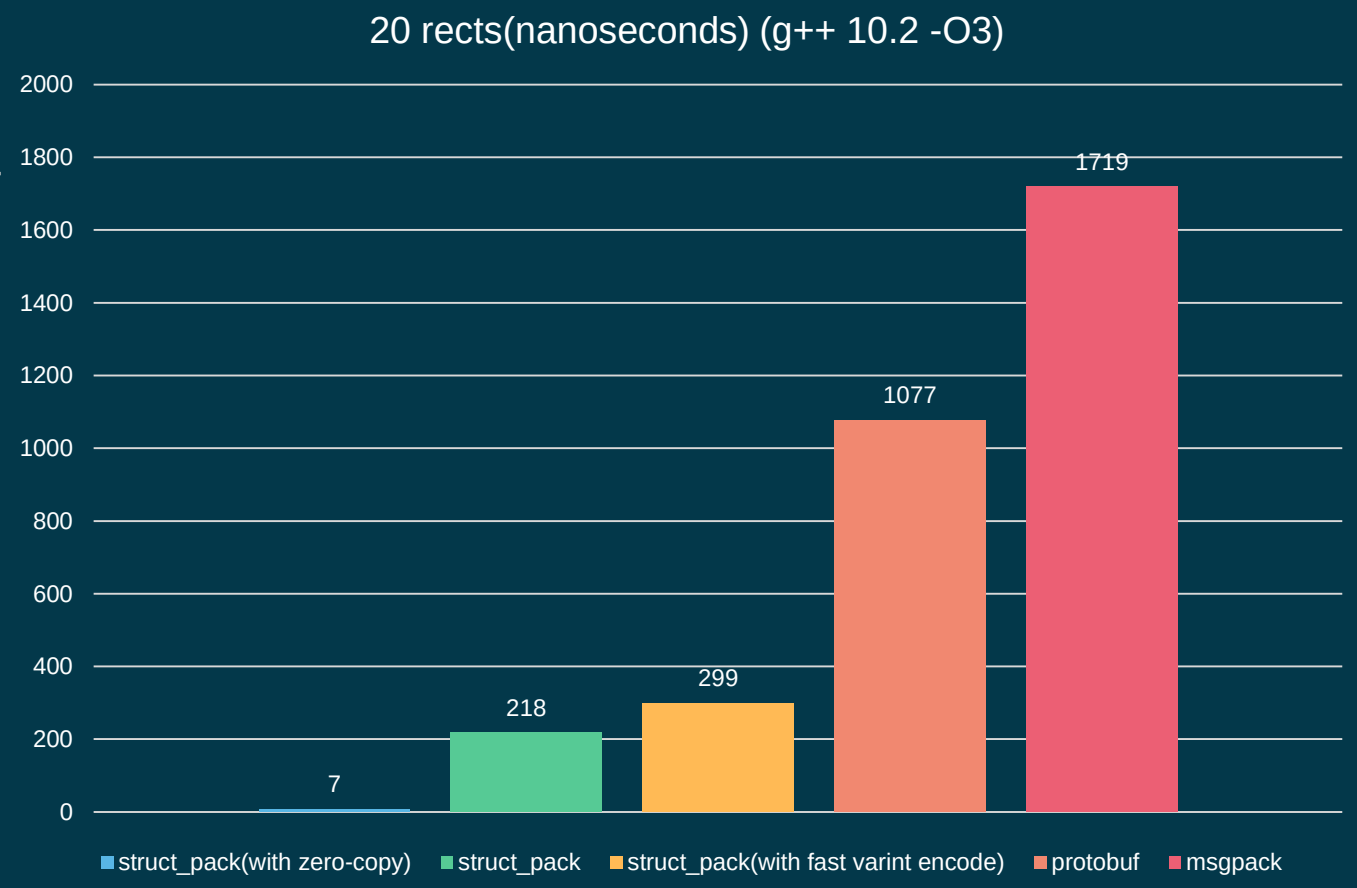


反序列化性能

此时，struct_pack(零拷贝模式)反序列化几乎完全不耗时。

正常模式/快速变长编码模式下，性能远高于protobuf和msgpack。

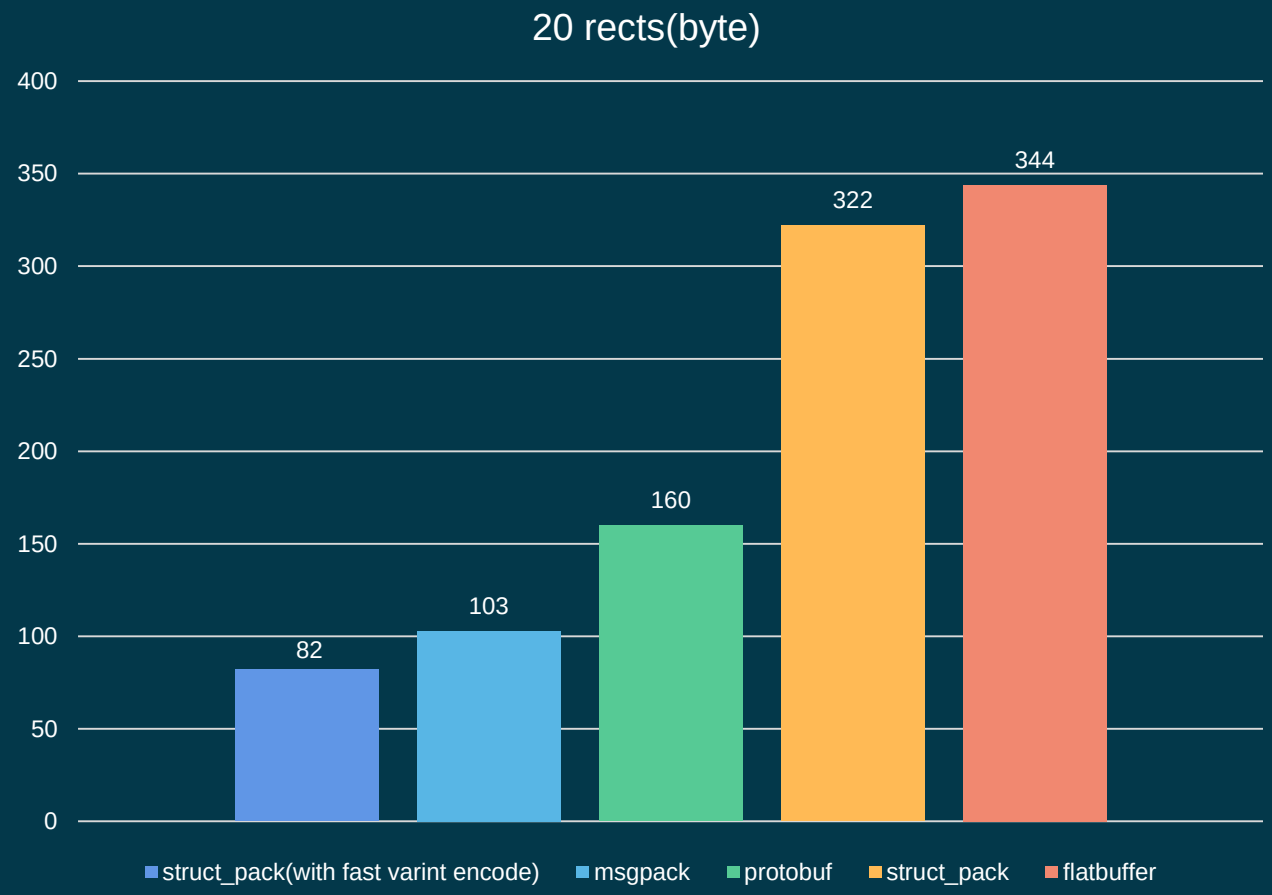
```
struct rect {
  int32_t x = 1;
  int32_t y = 0;
  int32_t width = 11;
  int32_t height = 1;
};
```



二进制大小

struct_pack在启用整数快速变长编码时，体积相当小。
平凡编码下体积略小于flatbuffer。

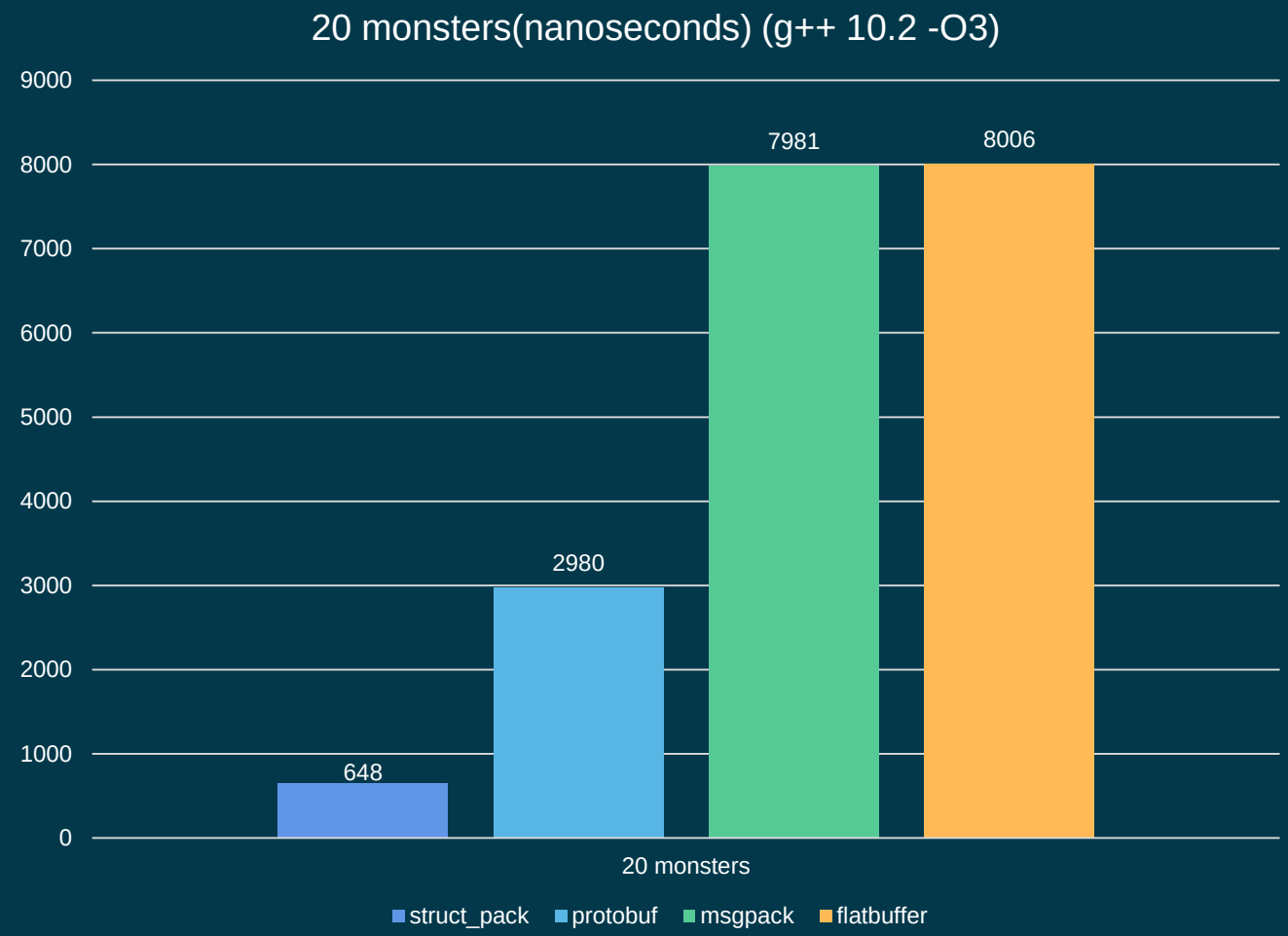
```
struct rect {
  int32_t x = 1;
  int32_t y = 0;
  int32_t width = 11;
  int32_t height = 1;
};
```



序列化性能

对monster这样的复杂结构体，struct_pack的序列化速度极快。

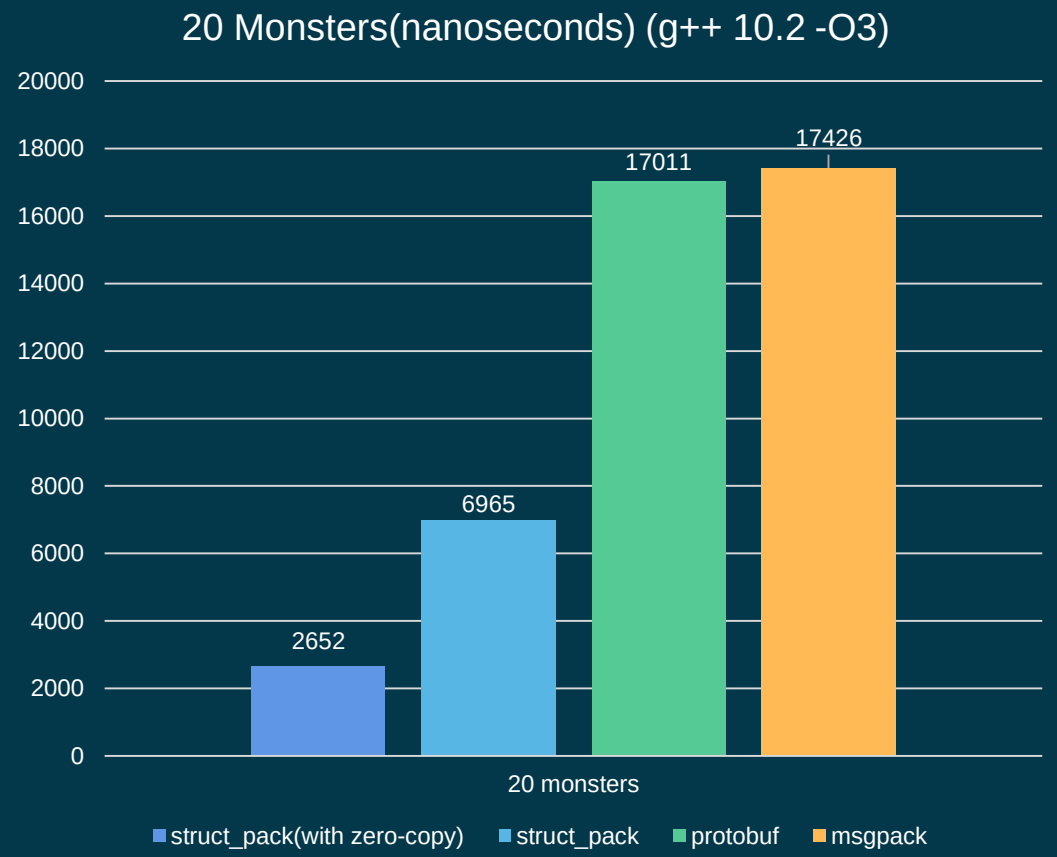
```
enum Color : uint8_t { Red, Green, Blue };
struct Vec3 {
  float x;
  float y;
  float z;
};
struct Weapon {
  std::string name;
  int16_t damage;
};
struct Monster {
  Vec3 pos;
  int16_t mana;
  int16_t hp;
  std::string name;
  std::string inventory;
  Color color;
  std::vector<Weapon> weapons;
  Weapon equipped;
  std::vector<Vec3> path;
 };
```



反序列化性能

零拷贝反序列化的模式下，struct_pack的性能相当优秀（不幸的是，只有部分字段支持零拷贝）
普通模式下依然远快于protobuf和msgpack

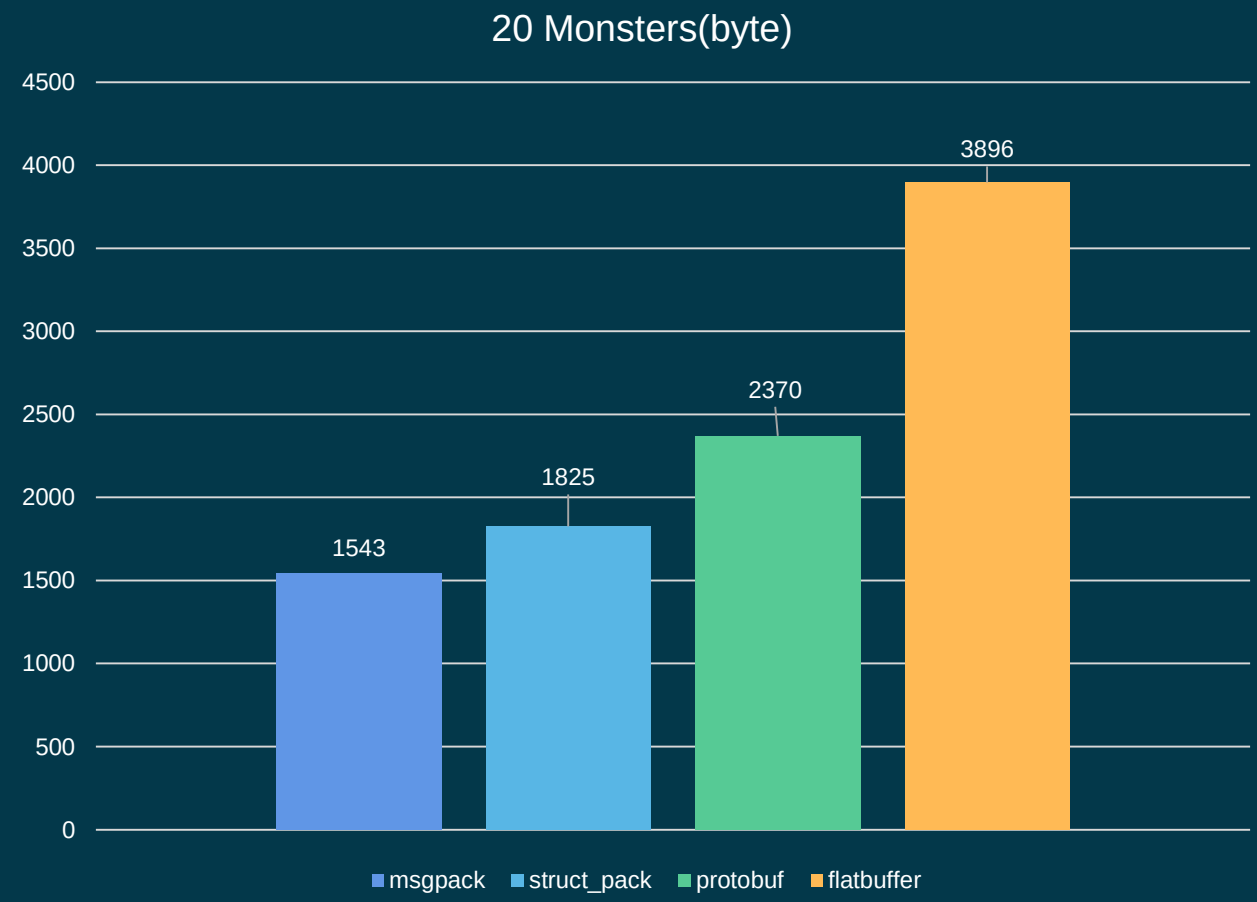
```cpp
enum Color : uint8_t { Red, Green, Blue };
struct Vec3 {
  float x;
  float y;
  float z;
};
struct Weapon {
  std::string_view name;
  int16_t damage;
};
struct Monster {
  Vec3 pos;
  int16_t mana;
  int16_t hp;
  std::string_view name;
  std::string_view inventory;
  Color color;
  std::vector<Weapon> weapons;
  Weapon equipped;
  std::span<Vec3> path;
 };
```



二进制大小

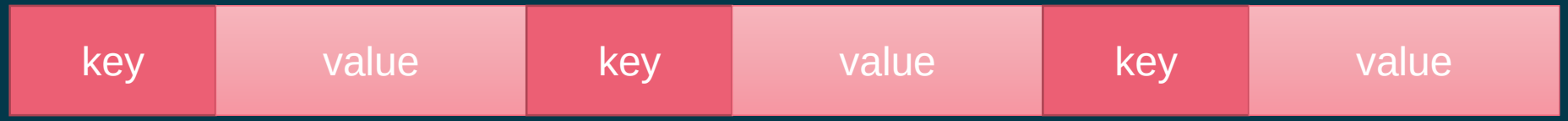
对复杂结构体，struct_pack的二进制大小也相对优秀

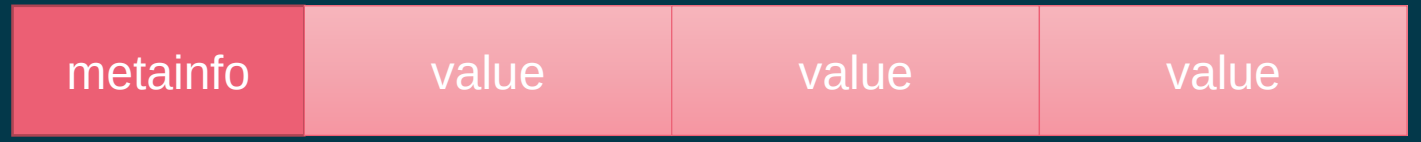
```
enum Color : uint8_t { Red, Green, Blue };
struct Vec3 {
  float x;
  float y;
  float z;
};
struct Weapon {
  std::string name;
  int16_t damage;
};
struct Monster {
  Vec3 pos;
  int16_t mana;
  int16_t hp;
  std::string name;
  std::string inventory;
  Color color;
  std::vector<Weapon> weapons;
  Weapon equipped;
  std::vector<Vec3> path;
 };
```



struct_pack的优势

1. 精简的类型信息，高效的类型校验。MD5计算在编译期完成，运行时只需要比较32bit的hash值是否相同即可。
2. struct_pack是一个head-only的模板库，允许编译器进行更多的内联优化。
3. 通过剥离类型信息+版本号排序，struct_pack尽可能减少了元数据的体积，并将元数据和数据分离开来。数据紧凑且紧密排列。





4. 平凡拷贝优化与零拷贝优化。（有赖于3）
5. 设计哲学：零成本抽象。
6. 通过预计算长度预分配内存，减少序列化过程中的内存分配与移动。
7. 通过hack技巧，跳过了std::string等类型resize时的初始化，从而减少一次内存写入。
8. etc…

Q&A

仓库地址：github.com/alibaba/yalantinglibs

文档：alibaba.github.io/yalantinglibs

开发团队：阿里云-基础软件部

谢谢