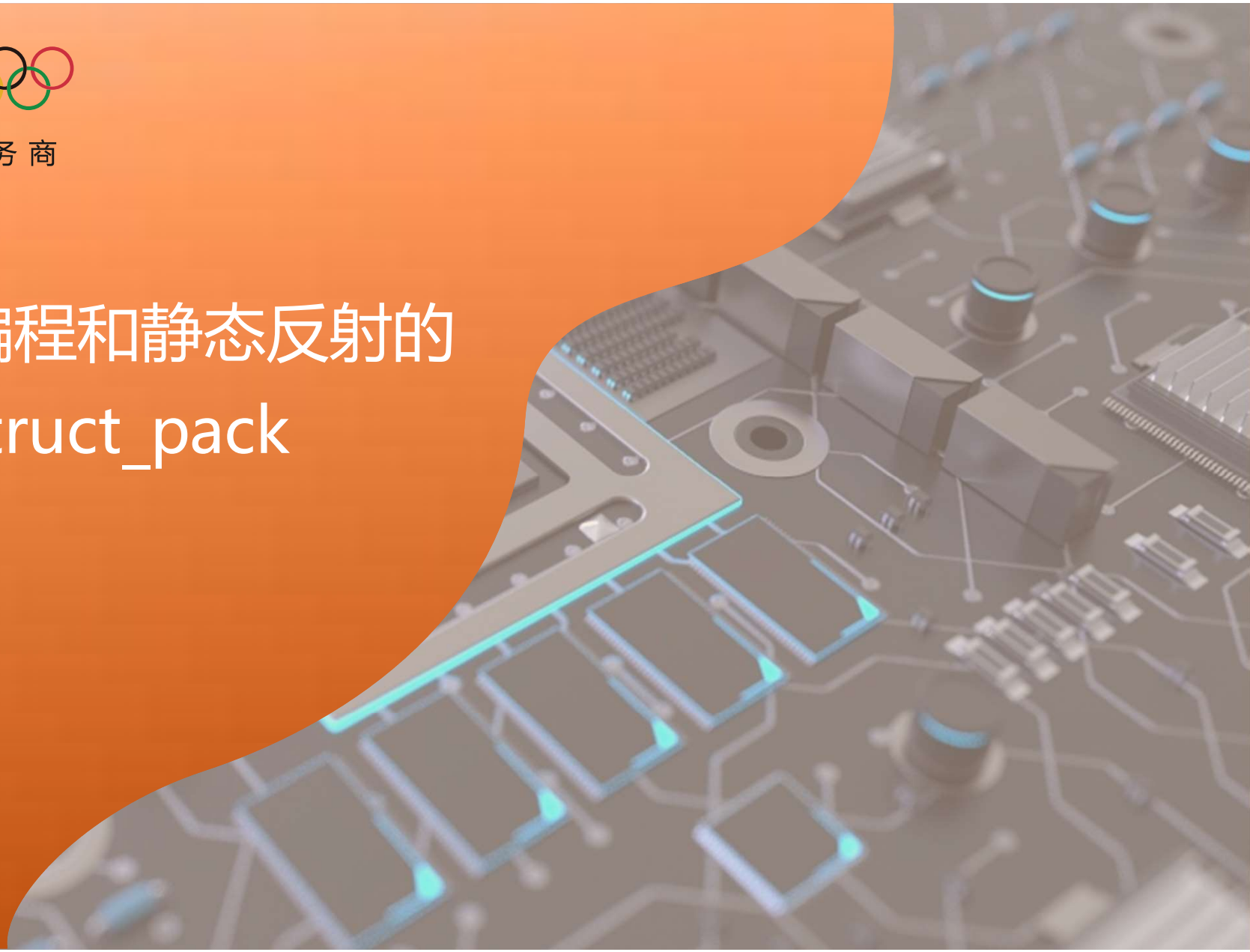
阿里云 | 奥运会全球指定云服务商

基于模板元编程和静态反射的序列化库：struct_pack



演讲人：李泽政

阿里云基础软件部

2022.10.23

阿里云 | 奥运会全球指定云服务商

目录
CONTENTS

阿里云 | 奥运会全球指定云服务商

PART 01

简介

struct_pack 简介


```cpp
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::vector<uint8_t> inventory;
  Color color;
  std::vector<Weapon> weapons;
  Weapon equipped;
  std::vector<Vec3> path;
};
```

```cpp
//初始化Monster
Monster m{…};
```

如何最优雅的序列化/反序列化这个Monster?

struct_pack: 一行代码完成序列化/反序列化。
不需要任何多余代码。

```cpp
//一行代码序列化
auto buffer = struct_pack::serialize(m);
//一行代码反序列化
auto monster2 = struct_pack::deserialize(buffer);
```

struct_pack 简介

struct_pack的接口十分易用：

```
struct person {
  int age;
  std::string name;
};
Person p{.age=24,.name="Betty"};

auto buffer = struct_pack::serialize(p);
//buffer is std::vector<char>

auto buffer = struct_pack::serialize<std::string>(p);
//use user-defined container

std::string buffer = "The next line is struct_pack data.\n";
struct_pack::serialize_to(buffer, p);
//serialize to container’s back;

auto buffer = struct_pack::serialize(p.age, p.name, std::vector{1,4,2,4,5});
//serialize type is std::tuple<int,std::string,std::vector<int>>
//serialize variadic parameters
```

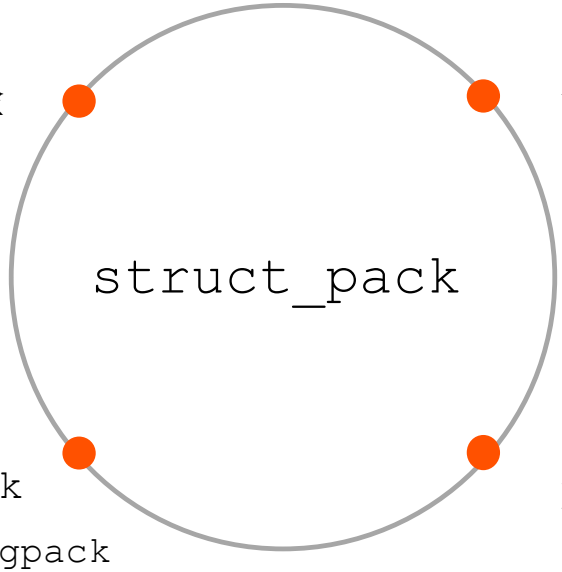
struct_pack 简介

struct_pack的接口十分易用：

```
auto buffer = struct_pack::serialize(p);
// 1. deserialize object to return value
{
    auto p2 = struct_pack::deserialize<person>(buffer);
    // p2 is expected<person,std::errc>
    assert(p2); //p2.has_value();
    assert(p == p2.value());
}
// 2. deserialize object to reference
{
    person p2;
    std::errc ec = struct_pack::deserialize_to(p2, buffer);
    assert(ec == std::errc{});
    assert(p == p2);
}
// 3. partial deserialize
{
    auto [err, name] = struct_pack::get_field<person, 1>(buffer);
    assert(p.name == name);
}
```

struct_pack 简介

阿里云 | 奥运会全球指定云服务商



类型校验

如何快速安全的校验数据的类型？

静态反射

在没有宏和任何其他辅助工具的情况下，如何访问一个类的成员？

类型系统

如何建立C++类型（包括用户自定义的类型）到struct_pack类型的映射？

benchmark

struct_pack vs protobuf vs msgpack

阿里云 | 奥运会全球指定云服务商

PART 02

静态反射

struct_pack – 静态反射

序列化/反序列化的前提：对于一个给定的对象，我们能够访问该对象的所有成员，并获取其类型。

Protobuf:

```
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;
}
```

.proto DSL

```
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.cpp &
Monster.h

```
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::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

struct_pack – 静态反射

序列化/反序列化的前提：对于一个给定的对象，我们能够访问该对象的所有成员，并获取其类型。

```
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);
};
```

```
#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); \
    }
```

Msgpack-C: 使用宏

struct_pack – 静态反射

序列化/反序列化的前提：对于一个给定的对象，我们能够访问该对象的所有成员，并获取其类型。

```
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;
}
```

struct_pack:不需要DSL或其他额外代码

通过编译期静态反射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？

struct_pack – 静态反射

如何在没有任何额外标记的情况下，在编译期获取结构体的各个字段？

C++标准中的Static Reflection 最快也需要C++26才能进入标准……我们需要寻找现有C++语法的“漏洞”

漏洞1： (C++11) Aggregate initialization：获取结构体的成员个数。

```
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;
```

struct_pack – 静态反射

如何在没有任何额外标记的情况下，在编译期获取结构体的各个字段？

C++标准中的Static Reflection 最快也需要C++26才能进入标准……我们需要寻找现有C++语法的“漏洞”

漏洞1： (C++11) Aggregate initialization：获取结构体的成员个数。

```
template <typename T, typename... Args>
consteval auto member_count() {
  //如果模板T{Args…,Any{}}不能被实例化，说明T的成员个数等于参数包Args的大小
  if constexpr (requires { T{{Args{}}..., {Any{}}}; } == false) {
    return sizeof...(Args);                    //返回参数包的大小
  }
  else {
    return member_count<T, Args..., Any>(); //否则，说明参数包小于T的成员数量，
那么我们给参数包增加一个Any，递归求解。
  }
}
// member_count<person>()==2
```

struct_pack – 静态反射

如何在没有任何额外标记的情况下，在编译期获取结构体的各个字段？

C++标准中的Static Reflection 最快也需要C++26才能进入标准……我们需要寻找现有C++语法的“漏洞”

漏洞2：(C++17) structured binding：读写结构体的各个成员

```
{
  std::tuple<int, double, float> t;
  auto &&[e0, e1, e2] = t;
  // e0==get<0>(t), e1==get<1>(t), e2==get<2>(t);
}
{
  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];
}
```

```
struct person {
  int age;
  std::string name;
};
person p{.age = 24, .name = "student"};
auto &&[e0, e1] = p;
// e0==p.age, e1==p.name;
```

struct_pack – 静态反射

阿里云 | 奥运会全球指定云服务商

结论： 我们找到了足够多的漏洞，C++17标准可实现aggregated class和tuple-like class的静态反射。

```
constexpr decltype(auto) visit_members(auto &&object, auto &&visitor) {
  constexpr auto Count = member_count<decltype(object)>();
  if constexpr (Count == 0) {
    return visitor();
  }
  else if constexpr (Count == 1) {
    auto &&[a1] = object;
    return visitor(a1);
  }
  else if constexpr (Count == 2) {
    auto &&[a1, a2] = object;
    return visitor(a1, a2);
  }
  else if constexpr (Count == 3) {…}
  else …
}
```

struct_pack – 静态反射

结论： 我们找到了足够多的**漏洞**，**C++17标准**可实现**aggregated class**和**tuple-like class**的静态反射。

```
struct Person {
  std::string name;
  int age;
};
struct Book {
  uint64_t ID;
  std::string name;
  double price;
};
auto visitor = [](auto &&...items) {
  ((std::cout << items << " "), ...);
};
Person p{.name = "student", .age = 24};
Book book{.ID=16,.name="A_tour_of_C++",price=4.2};
visit_members(p, visitor);
//输出: student 24
visit_members(book, visitor);
//输出: 16 A_tour_of_C++ 4.2
```

struct_pack – 静态反射

遗憾的是：如果一个类既不**aggregated**也不**tuple-like**，在真正的静态反射可用前，我们还是无法得到它的成员。
此外，用户可能想手动标记哪些字段需要被序列化，或者手动标记各字段序列化的顺序。
struct_pack也提供宏用于处理这些复杂情况。

```
class Customer {
 private:
  std::string first;
  std::string last;
  long val;
 public:
  std::string& firstname() { return first;}
  std::string& lastname()  { return last; }
  long& value()            { return val;  }
  STRUCT_PACK_REFL(Customer,val(),firstname(),lastname())
  //先序列化val，再序列化firstname和lastname。
};
```

PART 03

类型系统

struct_pack – 类型系统

C++具有极其复杂的类型系统：

如何支持各种复杂的模板类型？

如何支持用户自定义的数据结构？

如何支持嵌套类型？

如何进行类型校验？

struct_pack – 类型系统

我们需要将C++的类型映射到struct_pack的类型系统：

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

struct_pack – 类型系统

基本类型：

整数	编码
uint8_t , uint16_t , uint32_t , uint64_t	定长编码
int8_t , int16_t , int32_t , int64_t	定长编码（补码）
v_uint16_t,v_uint32_t,v_uint64_t,	变长编码
v_int16_t,v_int32_t,v_int64_t,	Zigzag+变长编码
float16_t,float32_t,float64_t	定长IEEE754浮点编码

struct_pack – 类型系统

约束类型：每个约束类型对应一个特定的C++20 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();
};
```

字符串类型的约束

struct_pack – 类型系统

约束类型：每个模板类型代表一个特定的C++20 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

类型名	concept	常见对应类型
string	字符串类型	std::string ,boost::string, folly::Fbstring std::u8string,std::wstring……
array	是C数组或std::array	T [] or std::array<T, size>
container	顺序容器	std::vector<T>,std::list<T>,std::deque< T>, boost::container::static_vector<T>
map	关联容器	std::map<T,U>,std::flat_map<T,U>, std::unordered_map<T,U>
set	顺序容器	std::set<T,U>,std::flat_set<T,U>, std::unordered_set<T,U>
optional_t	Optional-like 类型	std::optional<T>,boost::optional<T>
variant_t	Variant-like 类型	std::variant<T…>, boost::variant<T..>
expected_t	Expected-like 类型	std::expected<T,E>, tl::expected<T,E>

struct_pack – 类型系统

聚合类型：由若干不同类型组合成的结构体：

类型名	常见对应类型
Aggregated struct	struct/class，pair，tuple

阿里云 | 奥运会全球指定云服务商

PART 04

类型校验

struct_pack – 类型校验

有了完备的类型系统，我们可以轻松的支持对绝大部分类型的序列化和反序列化。

那么，如何进行快速的类型校验呢？

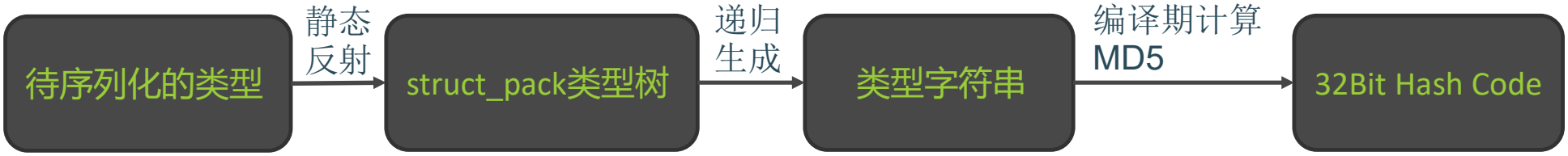
传统的类型校验方式：

struct_pack采用更为高效的类型校验方式：

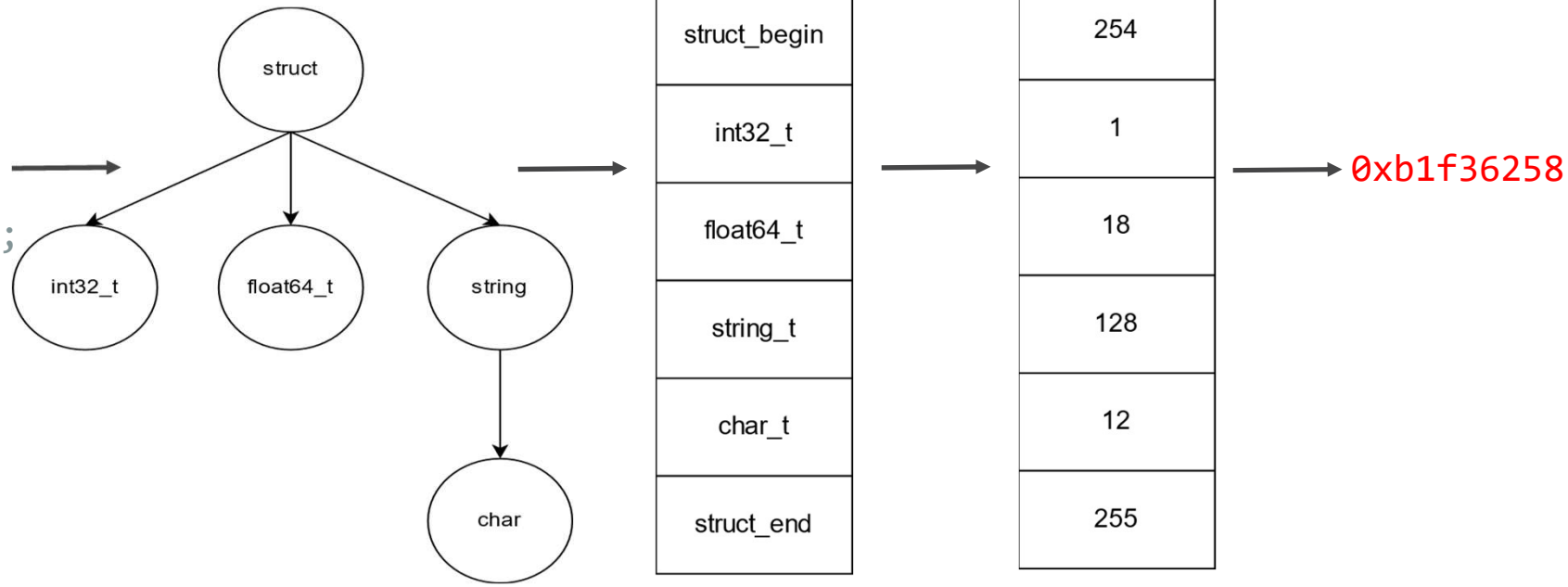
struct_pack – 类型校验

阿里云 | 奥运会全球指定云服务商

通过C++强大的类型系统，我们可以在编译期完成哈希计算：



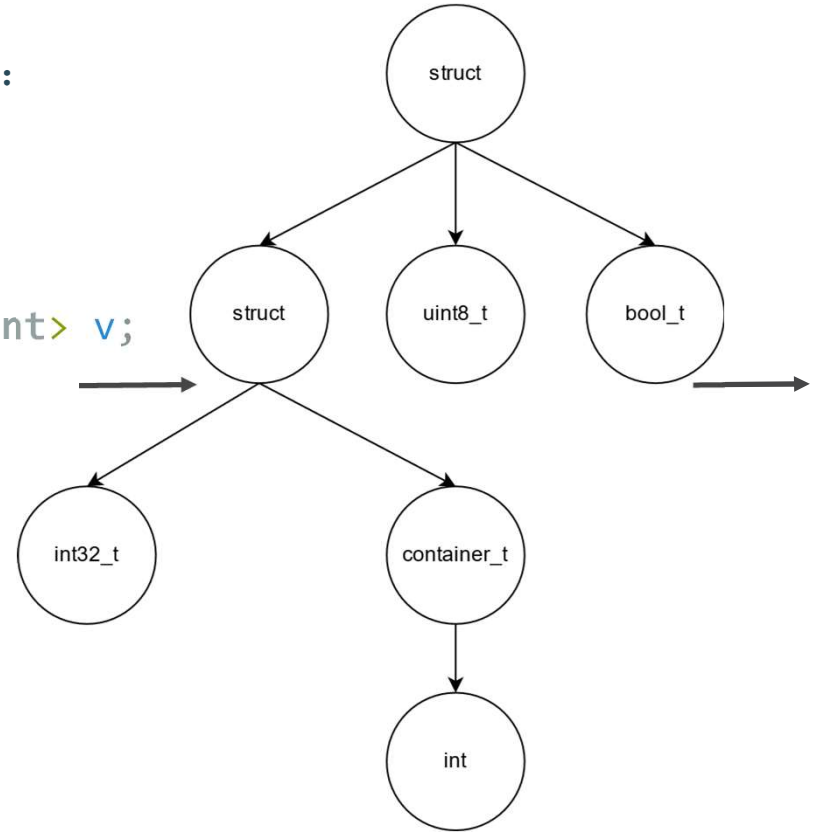

```
struct dummy {
  int32_t id;
  double number;
  std::string str;
};
```

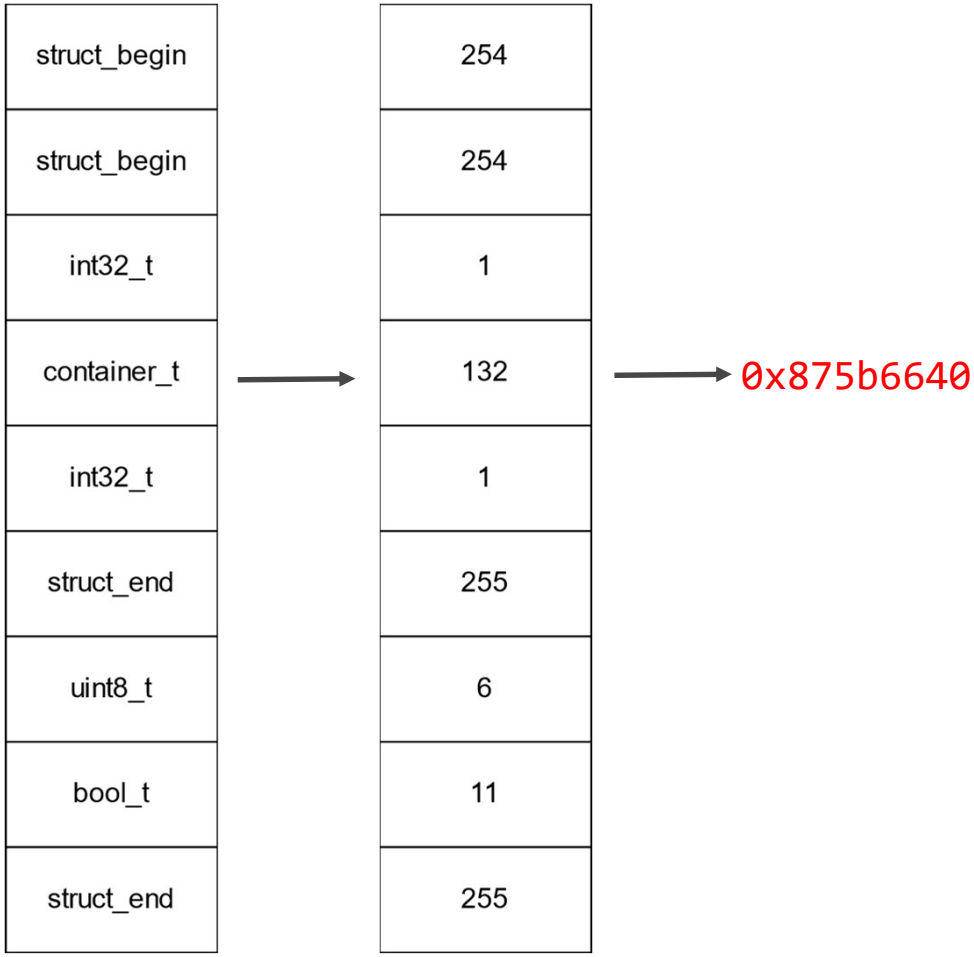


struct_pack – 类型校验

一个更复杂的类型：

```
struct dummy {
  int32_t i;
  std::vector<int> v;
};
struct nested {
  dummy d;
  uint8_t id;
  bool b;
};
```





struct_pack – 类型校验

如何缓解哈希冲突？

struct_pack在debug模式下记录完整的类型信息，从而在测试时检查出哈希冲突。

同时，用户可以打印肉眼可读的类型信息用于调试。

struct_pack – 类型校验

向前/向后兼容？

struct_pack提供struct_pack::compatible<T>模板，该类型可空，实现基本等于std::optional<T>, 但它不参与到哈希计算中，从而支持不同版本的结构体的向前/向后兼容性。

```
struct person_v1 {
  int age;
  std::string name;
};

struct person_v2 {
  int age;
  std::string name;
  struct_pack::compatible<bool> id;
};
```

```
person_v1 p1{.age=24,.name="Betty"};

auto buffer=struct_pack::serialize(p1);
auto p2=struct_pack::deserialize<person_v2>(buffer);

assert(p2);       //p2.has_value()==true;
assert(p2.value().age==p1.age);
assert(p2.value().name==p1.name);
assert(p2.value().id.has_value()==false);
```

struct_pack – 类型校验

类型转换

两个不同的C++类型可能具有相同的struct_pack类型(如：std::vector<int>和std::deque<int>)，struct_pack允许在这样不同的类型之间做类型转换(序列化/反序列化)。

```
auto buffer = struct_pack::serialize(std::vector{1, 4, 5, 7, 4});
auto res = struct_pack::deserialize<std::deque<int>>(buffer);
assert(res); //res.has_value()==true
assert(res.value() == std::deque{1, 4, 5, 7, 4});
```

阿里云 | 奥运会全球指定云服务商

PART 05

benchmark

struct_pack – benchmark

测试方法：

我们选取了三种代表性的结构体，对其进行一百万次序列化/反序列化，计算出单次反序列化的平均耗时。
结构体已预先初始化，已提前分配序列化结果的内存。

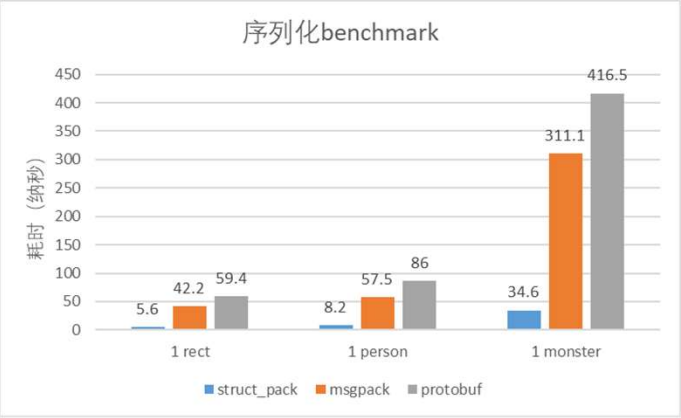
```cpp
struct rect {
  int32_t x;
  int32_t y;
  int32_t width;
  int32_t height;
};
```

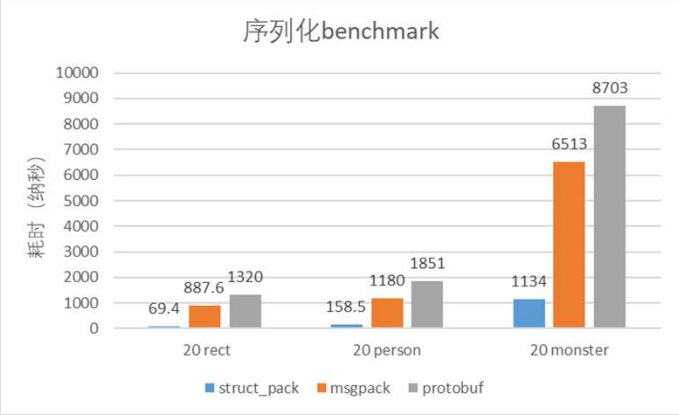
```cpp
struct person {
  int32_t id;
  std::string name;
  int age;
  double salary;
};
```

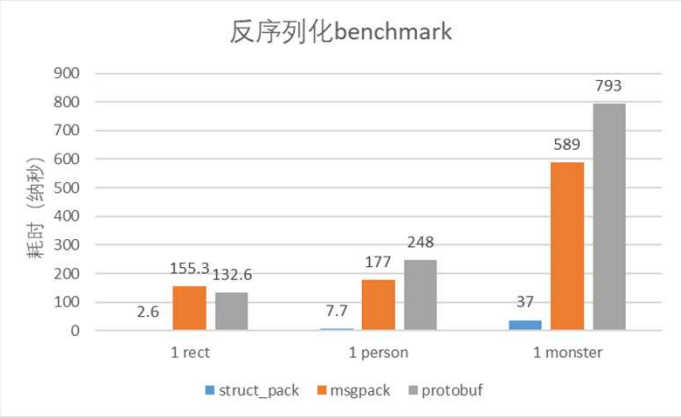
```cpp
enum Color : uint8_t { Red, Green, Blue };
struct Vec3 {
  float x;
  float y;
  float z;
};
struct Weapon {
  std::string name;
  int16_t damage;
};
struct Monster {
  Vec3 pos;
  int16_t mana;
  int16_t hp;
  std::string name;
  std::vector<uint8_t> inventory;
  Color color;
  std::vector<Weapon>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

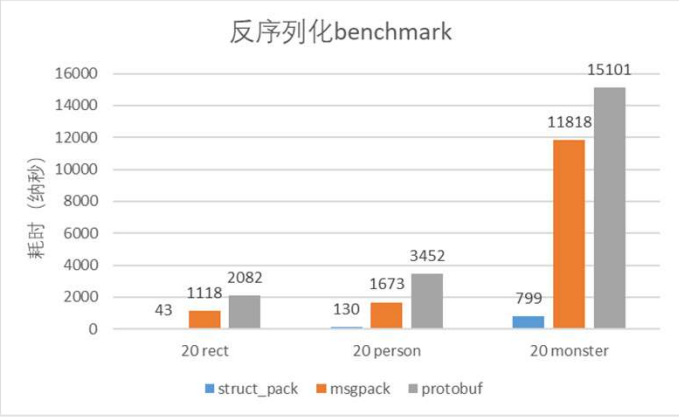
struct_pack – benchmark

结论：
struct_pack的性能比msgpack和protobuf要高出一个数量级
struct_pack能非常快的序列化/反序列化内存布局连续的元素。
struct_pack是零成本抽象的：性能和专家用户手写的序列化代码相当（来自于高德地图的使用反馈）









struct_pack – benchmark

为什么struct_pack如此之快？

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是一个模板库，鼓励编译器积极的内联函数。

3. 0成本抽象，不会为用不到的特性付出运行时代价。

4. struct_pack的内存布局更接近于C++结构体原始的内存布局，减少了序列化反序列化的工作量。

struct_pack – benchmark

为什么struct_pack如此之快？

5. 编译期类型计算允许struct_pack根据不同的类型生成不同的代码。因此我们可以根据不同的类型的特点做优化：

memcpy优化： 当结构体内含有continous_container（内存布局连续的容器），并且其元素是trival_copyable（可平凡拷贝）的，此时我们可以直接memcpy整个容器，从而利用SIMD指令加速。

```
// a large vector
std::vector<int> ar(1000000);
// serialize:   only once memcpy -> SSE/AVX
auto buffer = struct_pack::serialize(ar);
// deserialize: only once memcpy -> SSE/AVX
auto result = struct_pack::deserialize<std::vector<int>>(buffer);
```

struct_pack – benchmark

为什么struct_pack如此之快？

5. 编译期类型计算允许struct_pack根据不同的类型生成不同的代码。因此我们可以根据不同的类型的特点做优化：

Zero-Copy优化：当反序列化string_view,这样的类型时，我们不需要执行真正的拷贝，该string_view可以直接指向buffer，从而实现零拷贝。

```
auto buffer = struct_pack::serialize(std::string{"Hello world!"});
// deserialize: zero-copy
auto result = struct_pack::deserialize<std::string_view>(buffer);
```

struct_pack 仓库地址

struct_pack是yaLanTingLibs开源程序集的一部分。由阿里云基础软件部C++编译器团队维护。

yaLanTingLibs 目前包含：

struct_pack:	本次演讲的主人公。
coro_rpc:	基于协程的跨平台RPC框架，高效易用。
async_simple:	C++异步组件库，支持无栈协程/有栈协程。

代码地址：https://github.com/alibaba/yalantinglibs

Thank you