A Faster Serialization Library Based on Compile-time Reflection and C++ 20

YU QI
chufeng.qy@alibaba-inc.com

Outline

- Introduction of struct_pack
- How to Make Serialization Faster
- Compile-time Type Hash
- Optimize Performance by Reflection
- Backward Compatibility
- Benchmark

Introduction of struct_pack

struct_pack is a very fast serialization library based on compile-time reflection and c++20.

struct_apck will be open source soon.

Serialize an object with protobuf

- Serialize an object with protobuf

```
// person.proto
message person {
  int32 id = 1;
  string name = 2;
  int32 age = 3;
  double salary = 4;
}
```

```
void serialize_person() {
  person res;
  res.set_id(1);
  res.set_name("hello");
  res.set_age(20);
  res.set_salary(1024);

  std::string buf;
  res.SerializeToString(&buf);
}
```

Serialize an object with msgpack

```
struct person {
  int64_t id;
  std::string name;
  int age;
  double salary;
  MSGPACK_DEFINE(id, name, age, salary);
};

Intrusive


person p{};
msgpack::sbuffer ss;
msgpack::pack(ss, p);
```

Serialize an object with boost.serialization

```
namespace boost {
namespace serialization {

template<class Archive>
void serialize(Archive & ar, person & p, const unsigned int){
  ar & p.id;
  ar & p.name;
  ar & p.age;
  ar & p.salary;
}

} // namespace serialization
} // namespace boost

std::stringstream stream;
boost::archive::binary_oarchive archive(stream);

person p{};
archive << p;
std::string buf = stream.str();
```

It can be not safe, because you might miss some fields when you are writing code.

Serialize an object with struct_pack

```
person p{.id = 1, .name = "hello struct pack", .age =
20, .salary = 1024.42};


//serialize with one line code
auto buf = struct_pack::serialize(p);

person p1{};
//deserialize with one line code
auto [err, len] = struct_pack::deserialize_to(p1, buf.data(), buf.size());

assert(p == p1);
```

```cpp
struct complicated_object {
  Color color;
  int a;
  std::string b;
  std::vector<person> c;
  std::list<std::string> d;
  std::deque<int> e;
  std::map<int, person> f;
  std::multimap<int, person> g;
  std::set<std::string> h;
  std::multiset<int> i;
  std::unordered_map<int, person> j;
  std::unordered_multimap<int, int> k;
  std::array<person, 2> m;
  person n[2];
  std::pair<std::string, person> o;
};

struct nested_object {
  int id;
  std::string name;
  person p;
  complicated_object o;
};
```

```cpp
nested_object nested{.id = 2, .name = "tom", .p = {20, "tom"}, .o = v1};

auto buf = serialize(nested);

nested_object nested1{};
deserialize_to(nested1, buf.data(), buf.size());
```

No matter how complex the struct is,
only one line code is needed to serialize and deserialize.

Serialize an object with struct_pack

Features of struct_pack:

- Very easy to use(only one line code is needed)
- Very fast(benchmark)
- No macro
- No intrusive

Problems of classic serialization libs

Msgpack int16

```
+--------+--------+--------+
|  0xd1  |ZZZZZZZZ|ZZZZZZZZ|
+--------+--------+--------+
```

Msgpack float32

```
+--------+--------+--------+--------+--------+
|  0xca  |XXXXXXXX|XXXXXXXX|XXXXXXXX|XXXXXXXX|
+--------+--------+--------+--------+--------+
```

Protobuf int16

```
message Test {
  optional int16 a = 1;
}
```


```
+-------------------+-------------------+-------------------+
 | field_numer wire_type |zzzzzzzzzzzzzzzzzzzz |zzzzzzzzzzzzzzzzzzzz |
+-------------------+-------------------+-------------------+
```

- Steps of classic serialization

```
struct dummy{
 int id;
 float number;
 std::string str;
};
```

```
void serialize() {
  dummy t{};
  pack_type_info<int>();
  pack_value(t.id);
  pack_type_info<float>();
  pack_value(t.number);
  pack_type_info<std::string>();
  pack_value(t.str);
}
```

- Steps of classic deserialization

```
struct dummy{
  int id;
  float number;
  std::string str;
};
```

```
void deserialize(auto data, auto size) {
  dummy t{};
  auto type = unpack_type_info(data, size); #1
  if(is_int(type)) { #2
    t.id = unpack_value<int>(data, size); #3
  }else {
    throw std::invalid_argument("type error");
  }

  type = unpack_type_info(data, size);
  if(is_double(type)) {
    t.number = unpack_value<double>(data, size);
  }else {
    throw std::invalid_argument("type error");
  }

  if(is_str(type)) {
    t.str = unpack_value<std::string>(data, size);
  }else {
    throw std::invalid_argument("type error");
  }
}
```

How to speed up?

```
void serialize() {
  dummy t{};
  pack_type_info<int>();
  pack_value(t.id);
  pack_type_info<float>();
  pack_value(t.number);
  pack_type_info<std::string>();
  pack_value(t.str);
}
```

Don’t pack type info for each field every time, only save the value

```
void serialize() {
  dummy t{};
  pack_value(t.id);
  pack_value(t.number);
  pack_value(t.str);
}
```

How to speed up?

```
void deserialize(auto data, auto size) {
  dummy t{};
  auto type = unpack_type_info(data, size);
  if(is_int(type)) {
    t.id = unpack_value<int>(data, size);
  }else {
    throw std::invalid_argument("type error");
  }

  type = unpack_type_info(data, size);
  if(is_double(type)) {
    t.number = unpack_value<double>(data, size);
  }else {
    throw std::invalid_argument("type error");
  }

  if(is_str(type)) {
    t.str = unpack_value<std::string>(data, size);
  }else {
    throw std::invalid_argument("type error");
  }
}
```

Don’t unpack type and check type every time, only unpack the value

```
void deserialize(auto data, auto size) {
  dummy t{};
  unpack_value(t.id, data, size);
  unpack_value(t.number, data, size);
  unpack_value(t.str, data, size);
}
```

How to speed up?

- There is no need to serialize/deserialize type information for every field, because we can get fields meta data by compile-time reflection.

Serialization and Compile-time Reflection

```
struct dummy {
 int id;
 float number;
 std::string str;
};

void serialize(dummy t) {

 // reflect all fields of dummy into items…
 visit_members(t, [](auto &&...items){
  // serialize each item.
  serialize_many(items...);
 });
}
```

```
// reflect all fields of dummy.
visit_members(t, [](auto &&...items){
});

decltype(auto) visit_members(auto &&object, auto &&visitor) {
  using type = std::remove_cvref_t<decltype(object)>;

  constexpr auto Count = member_count<type>();

  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) {
    auto &&[a1, a2, a3] = object;
    return visitor(a1, a2, a3);
  }
  // ...
}
```

Get member count;

Structure binding;
Serialize/deserialize each field

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

template <typename T, typename... Args> consteval auto member_count() {
  // must be aggregate type.
  static_assert(std::is_aggregate_v<std::remove_cvref_t<T>>);

  // Utilize c++20 concepts to detect arguments count of an aggregate object.
  if constexpr (requires { T{{Args{}}..., {UniversalType{}}}; } == false) {
    return sizeof...(Args);
  } else {
    return member_count<T, Args..., UniversalType>();
  }
}
```

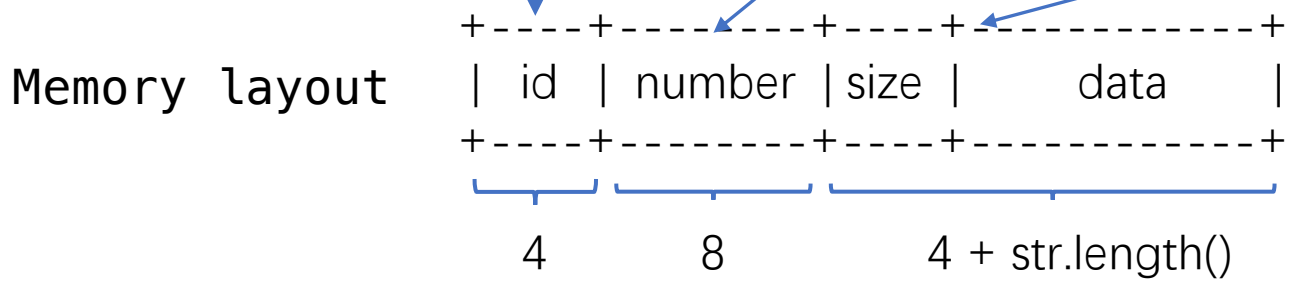
Limitations:

1. Must be aggregate type;
2. The max member count is limited;

```cpp
// reflection-ts no limitations.
// https://cplusplus.github.io/reflection-ts/draft.pdf
static_assert(get_size_v<T> == 1, "");
```

Data format of struct_pack

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



Memory layout is compact, no extra type information any more.
This data format make serialization/deserialization more efficient!

How to make type safe when deserialize an object?

Reflection based deserialization

```
void deserialize(auto data, auto size) {
  dummy t{};
  unpack_value(t.id, data, size);
  unpack_value(t.number, data, size);
  unpack_value(t.str, data, size);
}
```

No type info in binary, how to keep type safe when deserialize an object?

Type checking

- How to check types when deserialization?

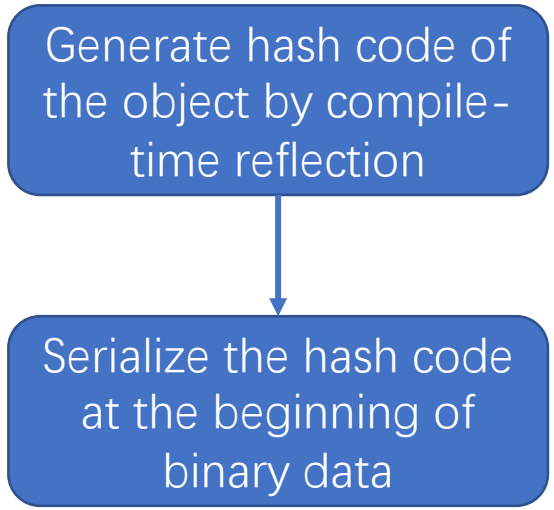
Of cause struct_pack need to do type checking, however struct_pack type checking is very efficient, because do type checking only one time.

Encode object type and all fields type into a 32 bits hash code at compile time!
The hash code will be used to do type checking

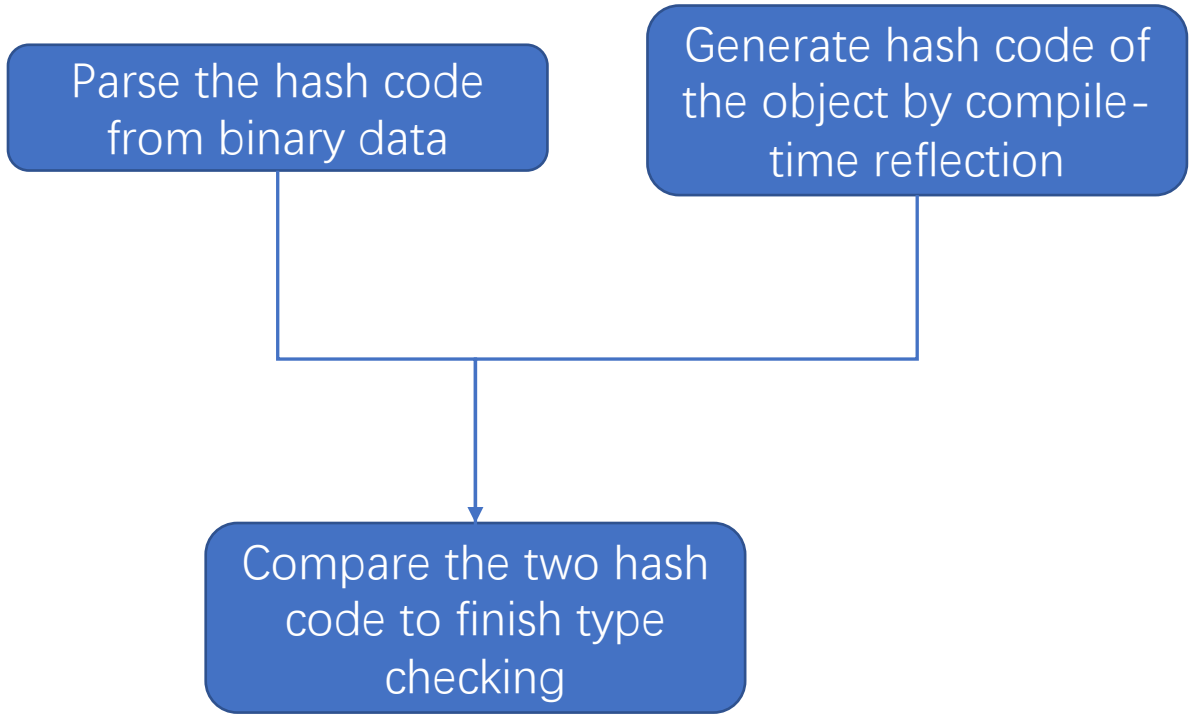
Type hash

- How to do type checking?

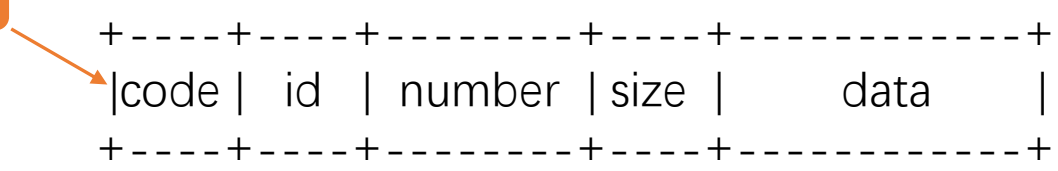
Serialization



Deserialization



Hash code



Type checking

`Performance compare`

	efficiency	frequency	Binary size
`Classic type checking`	Low. Parse type and check type for every field	Frequently, do type checking when parsing every field	Big, need to save extra type info for every field
`Reflection based type checking`	High. Parse an integer only one time at the beginning	Only one time, type checking at the beginning	Small, only one extra type information saved

Generate hash code of an object at compile-time

- How to generate the hash code of an object?
  - Mapping each member type to a unique type id
  - Generate a compile-time string by members type id
  - Generate a MD5 code from the string at compile-time

```
enum class type_id {
  int32_t,
  uint32_t,
  int64_t,
  uint64_t,
  int8_t,
  uint8_t,
  int16_t,
  uint16_t,
  char_t,
  uchar_t,
  bool_t,
  float_t,
  double_t,
  string_t,
  array_t,
  map_container_t,
  set_container_t,
  container_t,
  optional_t,
  aggregate_class_t = 254,
  aggregate_class_end_flag = 255,
};
```

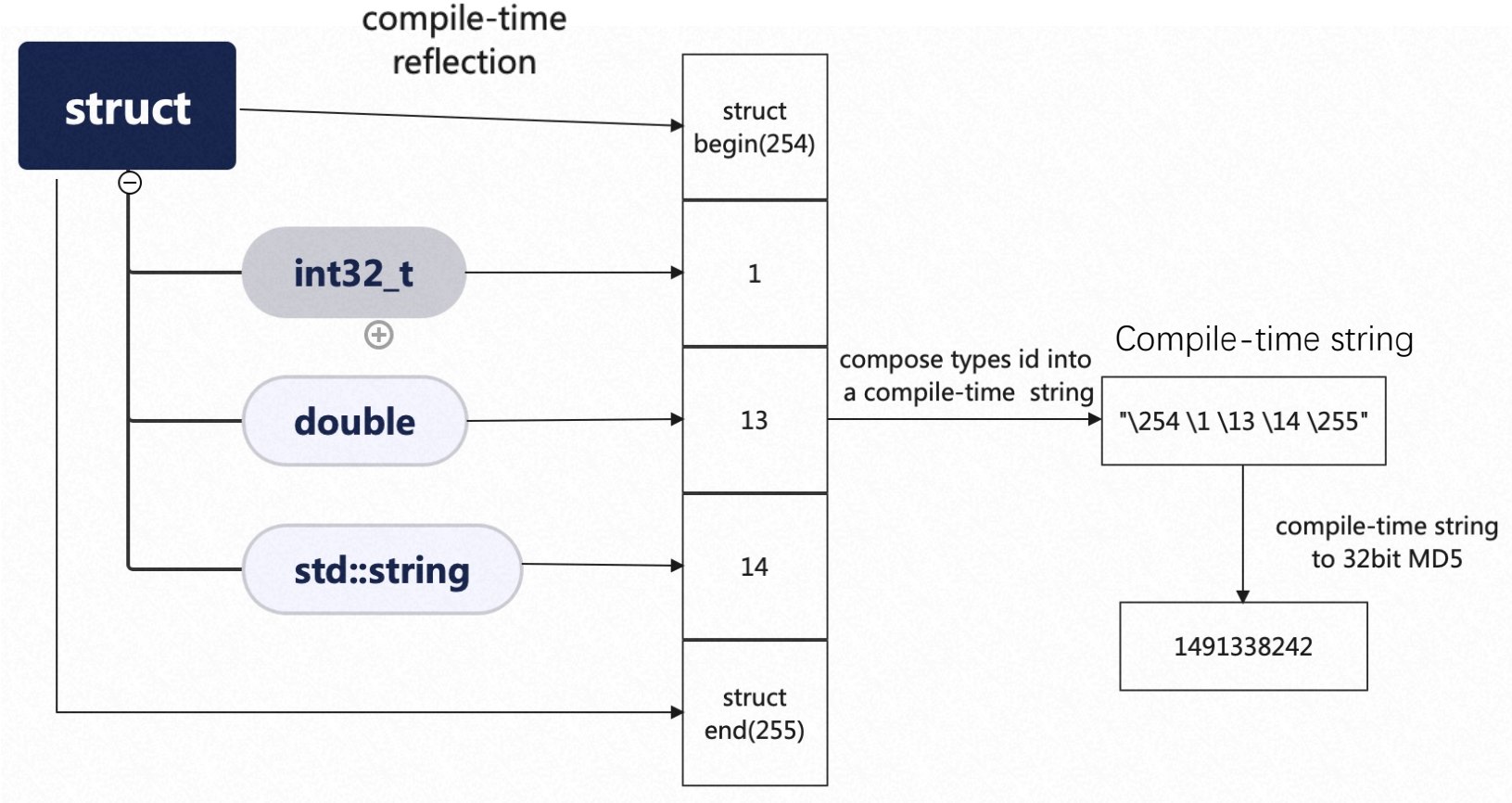
fundamental

Template class

Aggregate class

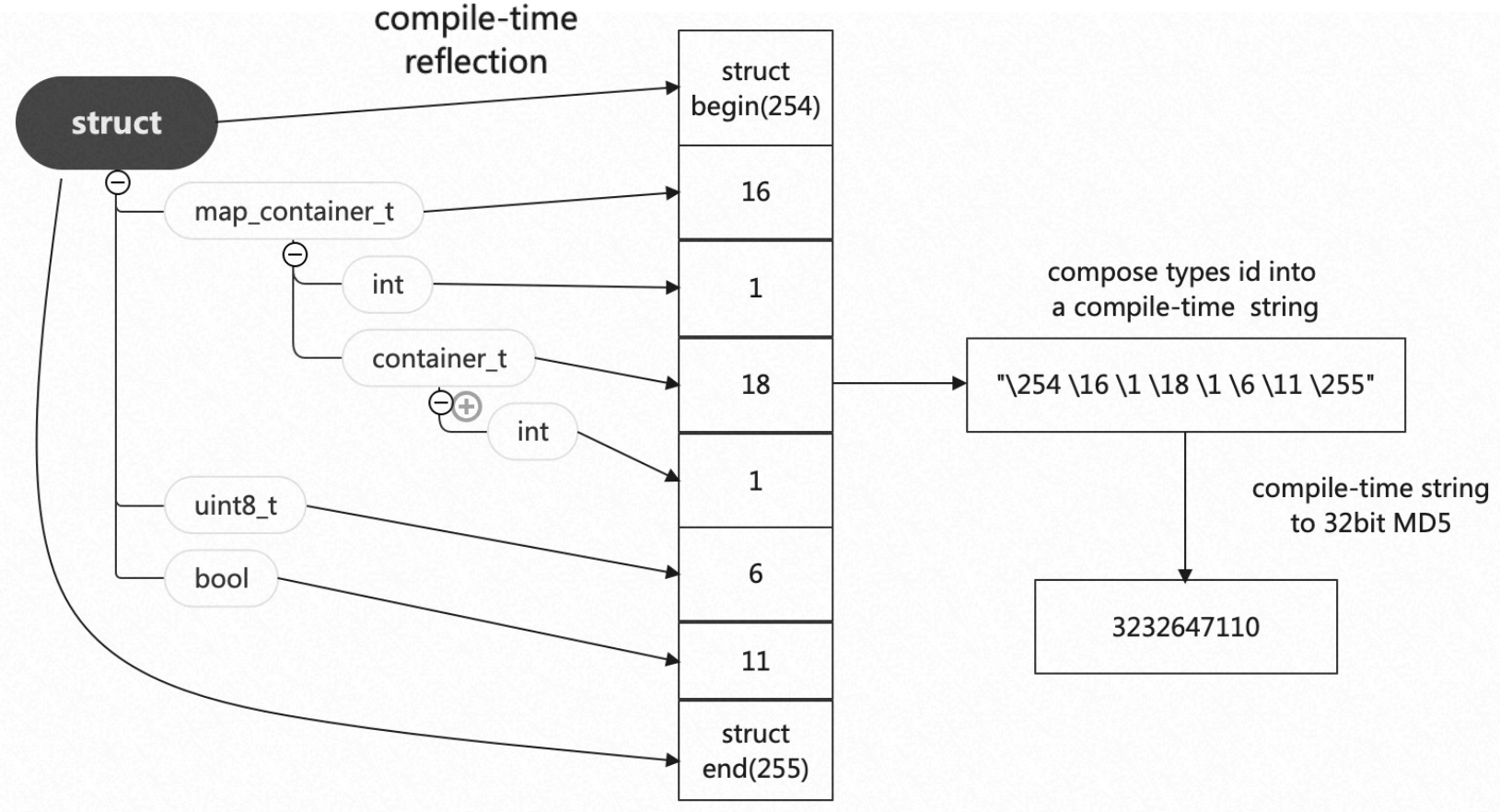
Generate hash code of a struct at compile time

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



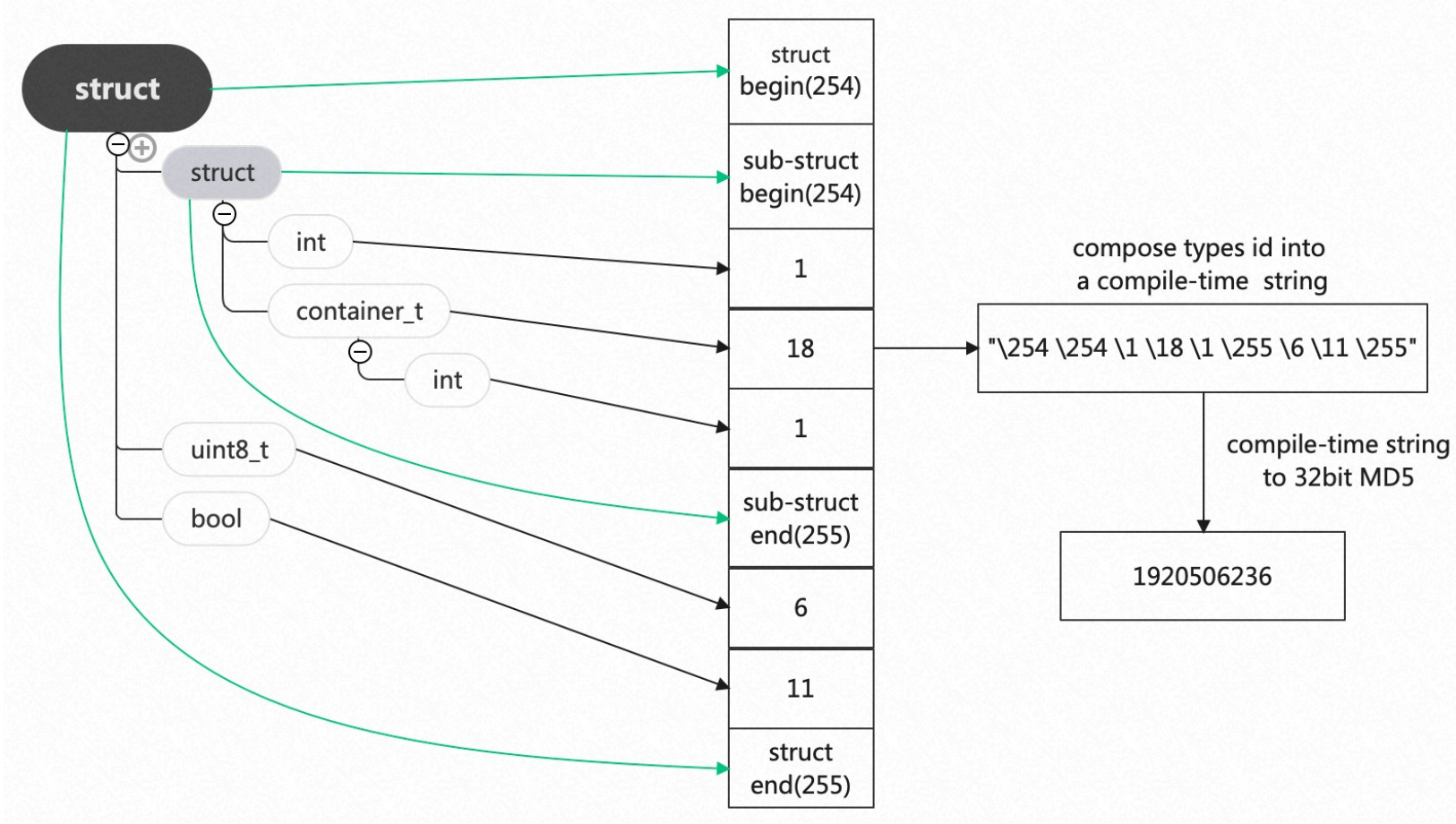
Generate hash code of a struct at compile time

```
struct dummy{std::map<int, std::vector<int>> map; uint8_t id; bool b; };
```



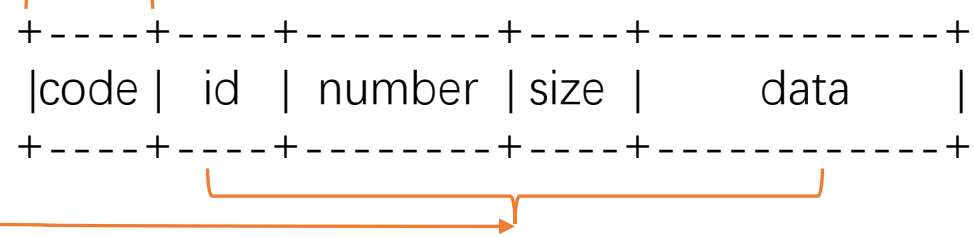
Generate hash code of a nested struct at compile time

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



Serialize hash code at the begging

```
template <typename T>
void serialize(T &&t) {
  // generate hash code of t.
  constexpr uint32_t types_code =
      get_types_code<decltype(get_types(std::forward<T>(t)))>();

  // serialize the hash code for deserialization type checking
  std::memcpy(data_ + pos_, &types_code, sizeof(uint32_t));
  pos_ += sizeof(uint32_t);

  // serialize t
  serialize_one(t);
}
```



Optimize performance

- Performance of serializing trivially copyable object can be greatly improved.
- Memory continuous container can be optimized

Optimize performance

```
struct dummy {
 int32_t a;
 int32_t b;
 int32_t c;
 int32_t d;
 double e;
 char f;
};

void serialize(dummy t) {
 pack_int(t.a);
 pack_int(t.b);
 pack_int(t.c);
 pack_int(t.d);
 pack_double(t.e);
 pack_char(t.f);
}
```

```
void serialize_one(auto &&item) {
  using type = std::remove_cvref_t<decltype(item)>;

  if constexpr (std::is_trivially_copyable_v<type>) {
    std::memcpy(data_ + pos_, &item, sizeof(type));
    pos_ += sizeof(type);
  }
}
```

Trivially copyable object only memcpy 1 time

memcpy 6 times

Optimize performance

```
struct rect { int x; int y int width; int height; };

void classic_serialize(const std::vector<rect> &rects) {
  for(auto &r : rects) {
    classic_serialize(r);
  }
}

void classic_serialize(const rect &r) {
  pack_int_type();
  pack_int_value(r.x);
  pack_int_type();
  pack_int_value(r.y);
  pack_int_type();
  pack_int_value(r.width);
  pack_int_type();
  pack_int_value(r.height);
}
```

Classic serialize times: rects.size() * 8

Optimize performance

```
struct rect { int x; int y int width; int height; };


template <typename T>
void reflection_based_serialize(const T &rects) {
  if constexpr (continuous_container<T>) {
    pack_size(rects.size());
    auto size = rects.size() * sizeof(T);
    std::memcpy(data_ + pos_, &rects[0], size);
  }
}
```

Only 2 times memcpy, Much more efficient!

2 Vs n * 8

Backward Compatibility

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

struct new_person {
  int age;
  std::string name;
  // add two fields later
  int32_t id;
  bool maybe;
};
```

Person and new_person are different types, how to keep backward compatibility?

Backward Compatibility

```cpp
struct person {
  int age;
  std::string name;
};
```

```cpp
struct new_person {
  int age;
  std::string name;
  compatible<int32_t> id;
  compatible<bool> maybe;
};
```

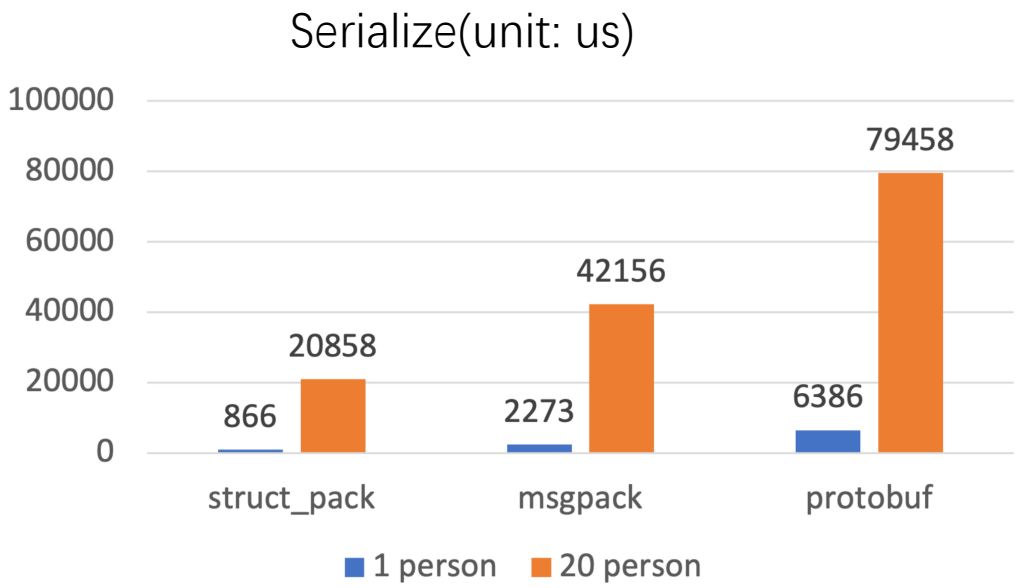
```cpp
static_assert(get_types_code<person>() == get_types_code<new_person>());
```

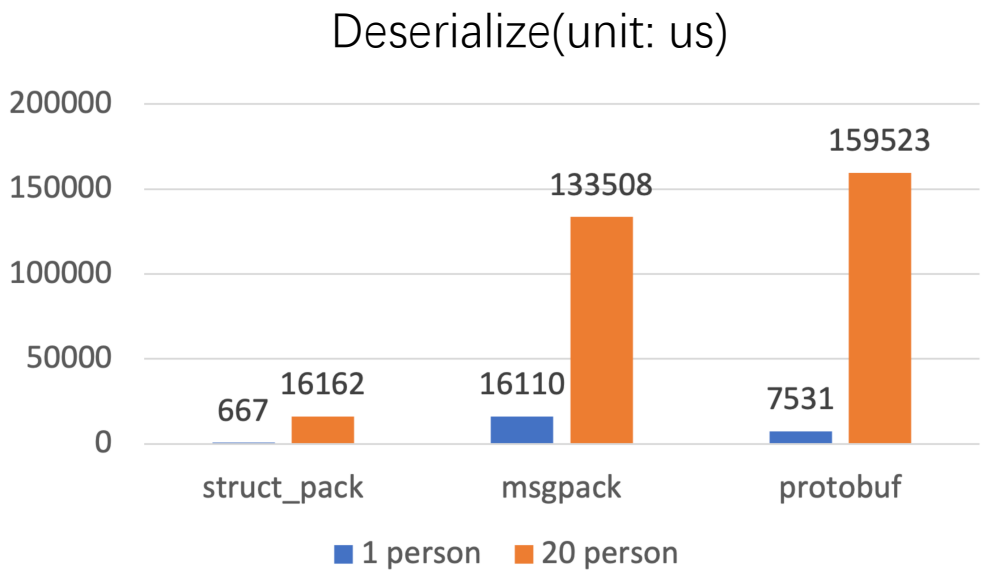
Idea:
Compatible only used for backward compatibility, only on the tail position;
Omit the tail compatible fields hash code;
Compile-time check compatible position;

Benchmark

- Simple object

```
struct person { int64_t id; std::string name; int age; double salary; };
```





Deserialize: 10x ~ 20x faster

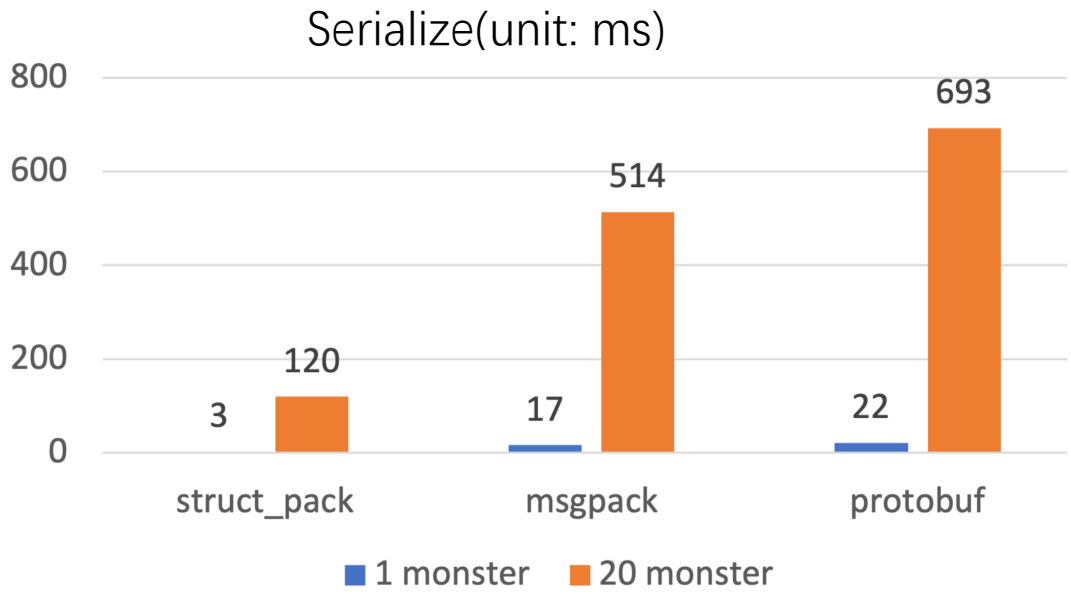
Benchmark

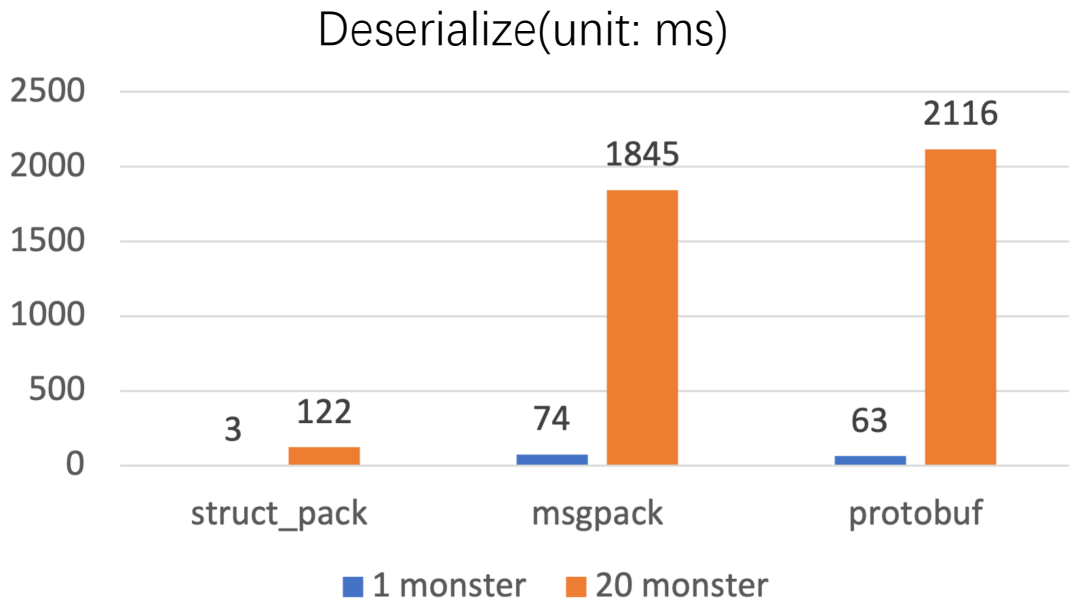
- Complex Object

```
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;
};
```

Benchmark

- Complex Object



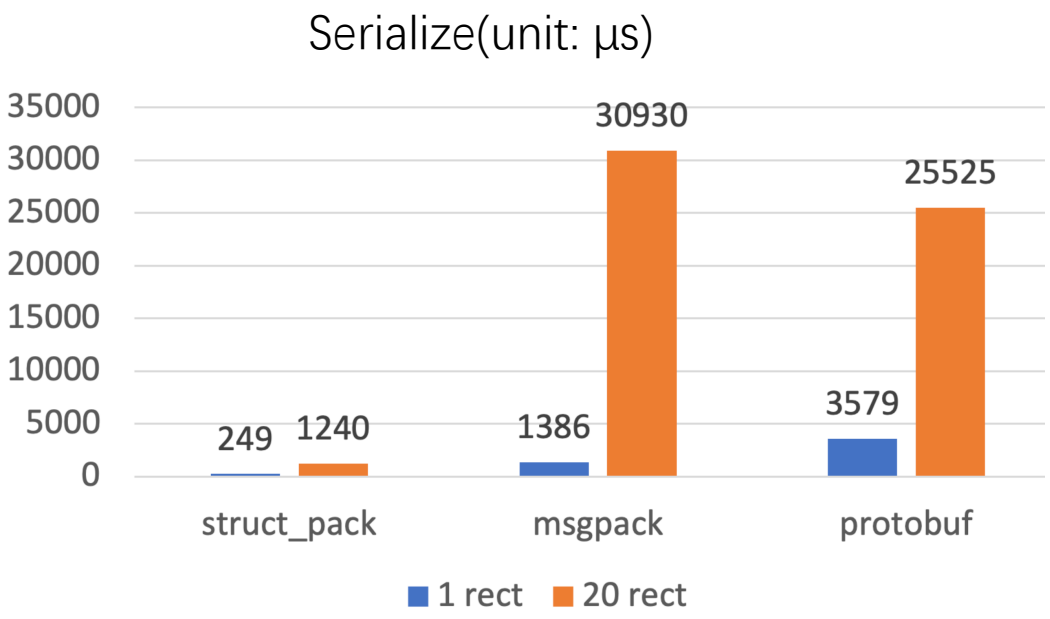


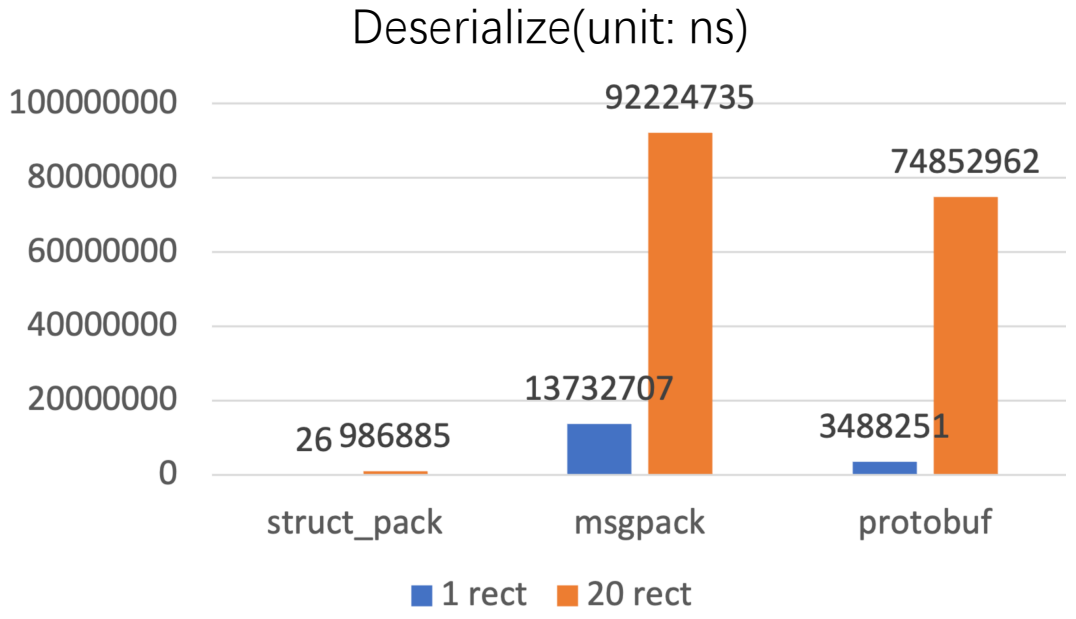
Deserialize: More than 20x faster

Benchmark

- Trivial copyable object

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





Deserialize: More than 100x faster

Thank you!

chufeng.qy@alibaba-inc.com