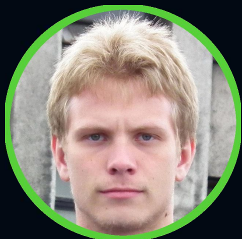
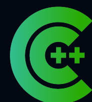
YsonStruct: дешевая сериализация иерархических JSON-структур



Иван Смирнов

Яндекс

ifsmirnov@ytsaurus.tech



C++ Russia 2023

Яндекс

План доклада

- YTsaurus и Yson
- Зачем нужны конфиги
- YsonSerializable: первый подход к снаряду
- YsonStruct

Кто такой YTsaurus

- Шардированная файловая система
- Распределённые блокировки
- MapReduce
- Key-value хранилище
- Единое пространство имён для MapReduce и key-value
- Интеграция с ClickHouse® и Apache Spark™

Yson

Аналог JSON

- Сильная типизация чисел: int, uint, double
- Атрибуты
- Бинарный формат
- Байтовые строки
- Опциональная запятая в конце

```
[
    "abc";
    12345;
    <
        "description" = "pi";
    > "3.1415";
    {
        "foo" = "bar";
        "baz" = "qux";
    };
]
```

Или без кавычек:

```
{foo=bar; baz=qux}
```

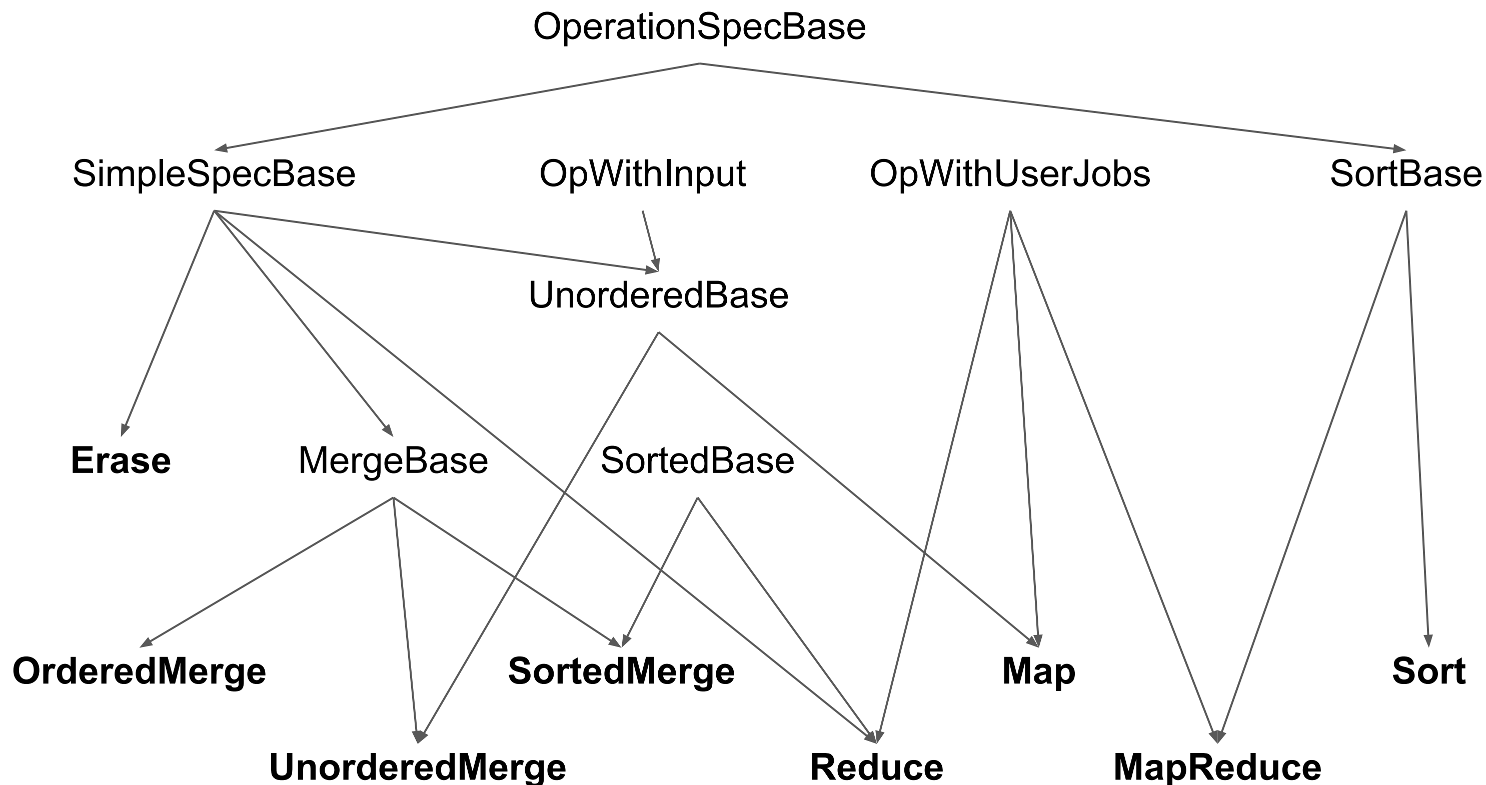
Зачем нужны конфиги

- Статическая конфигурация: файл «под ногами» у бинарника
- Динамическая конфигурация: файл в ZooKeeper/etcd/YTsaurus, инстансы регулярно его опрашивают
- Спецификации MapReduce операций
    - Входные и выходные таблицы
    - Квоты, параллельность, размеры тасков
    - Ввод-вывод, SSD/HDD, размеры буфера, ретраи, кеширование
- Настройки KV-хранилища
    - TTL
    - параметры LSM-деревьев

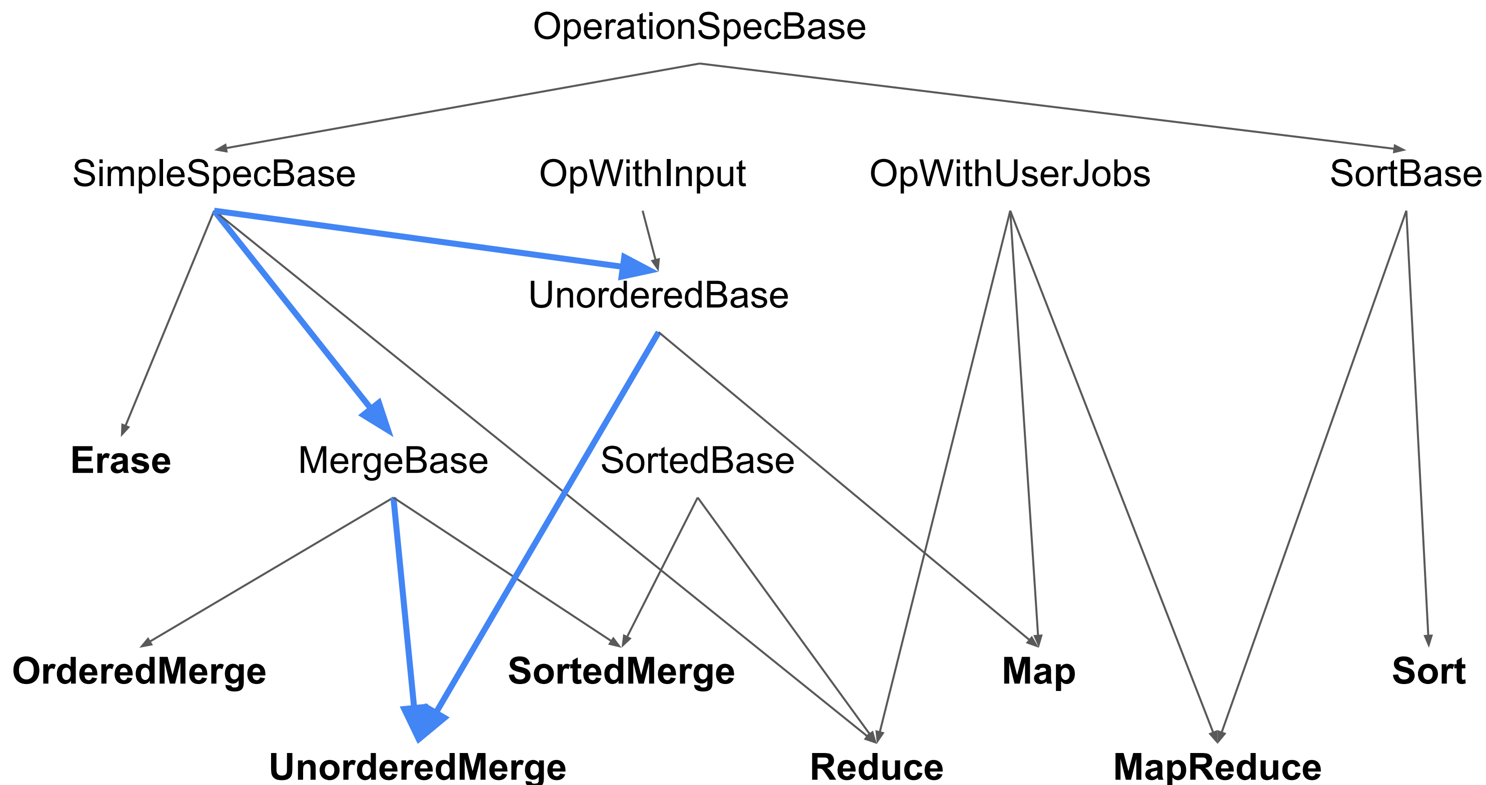
Конфиги в C++: техзадание

- Конвертация между Yson и структурами C++
- Сложные иерархии
  - Иерархия спецификаций MapReduce — более 20 классов

Иерархия спецификаций (выдержка)



Иерархия спецификаций (выдержка)



Конфиги в C++: техзадание

- Конвертация между Yson и структурами C++
- Сложные иерархии
  - Иерархия спецификаций MapReduce — более 20 классов
- Эффективность
  - Более 1000 опций для простейшей MapReduce операции
  - Десятки операций и сотни тасков в секунду
- Удобство
  - Легко завести новый конфиг
  - Легко скопипастить старый конфиг и получить новый

Взаимодействие с Yson

- Пользователь реализует перегрузки функций для своих типов
- `TYsonWriter` предоставляет интерфейс для записи в поток
- `TYsonValue` представляет материализованный объект Yson

```
void Serialize(const T& value, TYsonWriter& writer);
void Deserialize(T& value, const TYsonValue& yson);
```

- Реализации для стандартных типов есть в библиотеке
- `Deserialize` можно делать без материализации Yson

Взаимодействие с Yson: примеры

```
void Serialize(
    int value,
    TYsonWriter& writer)
{
    writer.OnInt(value);
}

template <class T>
void Serialize(
    const std::map<std::string, T>& map,
    TYsonWriter& writer)
{
    writer.OnBeginMap();
    for (const auto& [key, value] : map) {
        writer.OnMapKey(key);
        Serialize(value, writer);
        writer.OnMapDelimiter();
    }
    writer.OnEndMap();
}
```

```
void Deserialize(
    int& value,
    const TYsonValue& yson)
{
    value = yson.Get<int>();
}

template <class T>
void Deserialize(
    std::map<std::string, T>& map,
    const TYsonValue& yson)
{
    const auto& ysonMap = yson.AsMap();
    for (const auto& [key, value] : ysonMap) {
        Deserialize(map[key], value);
    }
}
```

YsonSerializable

- Базовый класс `TYsonSerializable`
  - словарь `std::string` → `std::unique_ptr<IParameter>`
- `IParameter` — интерфейс для (де)сериализации полей произвольного типа
- Типизированные наследники `IParameter` ссылаются на конкретное поле экземпляра класса
- Пользователь наследует `TYsonSerializable` и регистрирует поля в конструкторе

YsonSerializable: пример использования

```
struct TServerConfig : public TYsonSerializable {
    std::string HostName;
    int Port;

    TServerConfig()
    {
        RegisterParameter("host_name", HostName);
        RegisterParameter("port", Port);
    }
};

auto yson = ReadYsonFromFile("config.txt");
auto config = std::make_unique<TServerConfig>();
Deserialize(config, yson);
```

YsonSerializable: код

```
struct TYsonSerializable {
    std::map<std::string, std::unique_ptr<IParameter>> Parameters;

    template <class T>
    IParameter& RegisterParameter(const std::string& name, T& value) {
        return *Parameters.emplace(name, std::make_unique<TParameter<T>>(parameter))
            .first->second;
    }
};

void Serialize(const TYsonSerializable& value, TYsonWriter& writer) {
    writer.OnBeginMap();
    for (const auto& [key, parameter] : value.Parameters) {
        writer.OnMapKey(key);
        parameter->Serialize(writer);
        writer.OnMapDelimiter();
    }
    writer.OnEndMap();
}

// Deserialize симметричный
```

YsonSerializable: реализация TParameter

`TParameter` — тривиальная типизированная обёртка над полем.

```
template <class T>
struct TParameter : public IParameter {
    T& Field;

    void Serialize(TYsonWriter& writer) const override {
        Serialize(Field, writer);
    }

    void Deserialize(const TYsonValue& yson) override {
        Deserialize(Field, yson);
    }

    TParameter(T& field) : Field(field) {}
};
```

YsonSerializable

- Перевели на него все конфиги в системе
- Добавили фичи
    - валидация параметров: при ошибке вылетает исключение

```
RegisterParameter("port", Port)
    .InRange(1024, 65535);
```

YsonSerializable

- Перевели на него все конфиги в системе
- Добавили фичи
  - валидация параметров: при ошибке вылетает исключение
  - алиасы для имён для сохранения совместимости

```
RegisterParameter("host_name", HostName)
    .Alias("host"); // старые конфиги с прежним именем продолжают работать
```

YsonSerializable

- Перевели на него все конфиги в системе
- Добавили фичи
    - валидация параметров: при ошибке вылетает исключение
    - алиасы для имён для сохранения совместимости
    - значения по умолчанию

```
RegisterParameter("host_name", HostName)
    .Default("ytsaurus.tech");
```

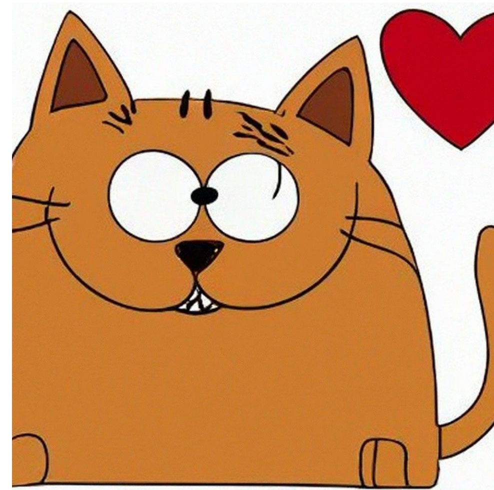
YsonSerializable

- Перевели на него все конфиги в системе
- Добавили фичи
    - валидация параметров: при ошибке вылетает исключение
    - алиасы для имён для сохранения совместимости
    - значения по умолчанию
    - постпроцессинг

```
RegisterPostprocessor([] (TServerConfig* config) {
    if (!config->HostName.starts_with("https://")) {
        config->HostName = "https://" + config->HostName;
    }
});
```

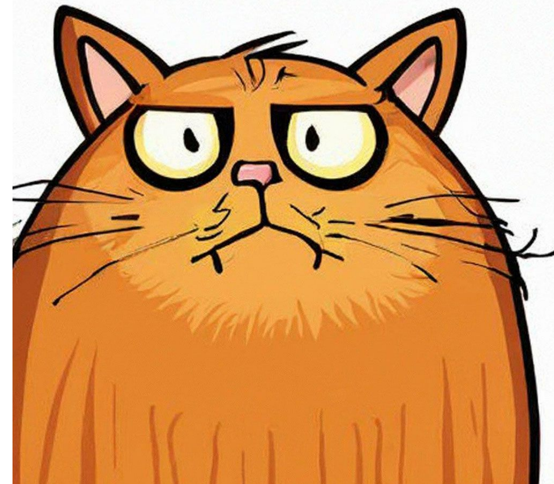
YsonSerializable

- Перевели на него все конфиги в системе
- Добавили фичи
  - валидация параметров: при ошибке вылетает исключение
  - алиасы для имён для сохранения совместимости
  - значения по умолчанию
  - постпроцессинг
- Полюбили



YsonSerializable

- Перевели на него все конфиги в системе
- Добавили фичи
  - валидация параметров: при ошибке вылетает исключение
  - алиасы для имён для сохранения совместимости
  - значения по умолчанию
  - постпроцессинг
- Полюбили
- Конфиги растут, RPS растёт
- Сериализация начинает тормозить



YsonSerializable

- Перевели на него все конфиги в системе
- Добавили фичи
  - валидация параметров: при ошибке вылетает исключение
  - алиасы для имён для сохранения совместимости
  - значения по умолчанию
  - постпроцессинг
- Полюбили
- Конфиги растут, RPS растёт
- Сериализация начинает тормозить
  - профилировщик: конструкторы и деструкторы
  - словарь дескрипторов полей создаётся заново в конструкторе каждого экземпляра

YsonStruct: идея

- Храним дескрипторы полей per-тип, а не per-экземпляр
- Один мета-класс на тип
- `TParameter` хранит pointer-to-member вместо ссылки на поле экземпляра
- Мета заполняется при создании первого экземпляра класса
- Регистрируем параметры в статическом методе вместо конструктора

YsonStruct

```
struct TYsonStruct {
    TMeta* Meta;
};

struct TServerConfig : public TYsonStruct {
    std::string HostName;
    int Port;

    REGISTER_YSON_STRUCT(TServerConfig);

    static void Register(TRegistrar registrar);

    /* Что скрывается за REGISTER_YSON_STRUCT:
    TServerConfig() {
        TRegistry::Initialize(this);
    }
    using TRegistrar = TYsonStructRegistrar<TServerConfig>;
    using TThis = TServerConfig;
    */
};
```

YsonStruct: нет наследования — инициализация

- `TMeta`: словарь `std::string → std::unique_ptr<IParameter>`
- `TParameter<TStruct, TValue>`: храним pointer-to-member
- `TRegistry::Initialize` зовётся в конструкторе пользовательских конфигов
    - При первом обращении создаём мету, при последующих — переиспользуем

```
template <class TStruct>
void TRegistry::Initialize(TStruct* instance) {
    static TMeta* meta = nullptr; // Не thread-safe, об этом позже
    if (!meta) {
        meta = new Meta(); // Течёт память, но это не страшно
        TStruct::Register(TRegistrar<TStruct>(meta));
    }
    instance->Meta = meta;
}
```

YsonStruct: регистрация

- `RegisterParameter` стал методом `TMeta`
- `TRegistrar` — типизированная обёртка над `TMeta`

```
void TServerConfig::Register(TRegistrar* registrar) {
    registrar->Parameter("host_name", &TThis::HostName)
        .Default("ytsaurus.tech");
    registrar->Parameter("port", &TThis::Port)
        .InRange(1024, 65535);
}
```

YsonStruct: регистрация

Было:

```
TServerConfig::TServerConfig() {
    RegisterParameter("host_name", HostName)
        .Default("ytsaurus.tech");
    RegisterParameter("port", Port)
        .InRange(1024, 65535);
}
```

Стало:

```
void TServerConfig::Register(TRegistrar* registrar) {
    registrar->Parameter("host_name", &TThis::HostName)
        .Default("ytsaurus.tech");
    registrar->Parameter("port", &TThis::Port)
        .InRange(1024, 65535);
}
```

YsonStruct: сериализация

```
void Serialize(const TYsonStruct& self, TYsonWriter& writer) {
    writer.OnBeginMap();
    for (const auto& [key, parameter] : value.Meta->Parameters) {
        writer.OnMapKey(key);
        parameter->Serialize(self, writer);
        writer.OnMapDelimiter();
    }
    writer.OnEndMap();
}
```

YsonStruct: реализация TParameter

`TParameter` — чуть менее тривиальная типизированная обёртка.

```
template <class TStruct, class TValue>
struct TParameter : public IParameter {
    TValue TStruct::*Field;

    void Serialize(const TYsonStruct& self, TYsonWriter& writer) override {
        ::Serialize(dynamic_cast<const TStruct&>(self).*Field, writer);
    }
}
```

YsonStruct: наследование — инициализация

- Вспомним, что звалось в конструкторе

```
template <class TStruct>
void TRegistry::Initialize(TStruct* instance) {
    static TMeta* meta = nullptr;
    if (!meta) {
        meta = new Meta();
        TStruct::Register(TRegistrar<TStruct>(meta));
    }
    instance->Meta = meta;
}
```

- В `Initialize` базовых классов будет своя `meta`
- Нужно обойти всю иерархию
- Единственный способ — позвать конструктор

YsonStruct: наследование — инициализация

```
TMeta* CurrentlyInitializingMeta = nullptr;

template <class TStruct>
void TRegistry::Initialize(TStruct* instance) {
    if (CurrentlyInitializingMeta) {
        TStruct::Register(TRegistrar<TStruct>(CurrentlyInitializingMeta));
        return;
    }

    static TMeta* meta = nullptr;
    if (!meta) {
        meta = new Meta();
        CurrentlyInitializingMeta = meta;
        TStruct{}; // проваливаемся в первый if
        CurrentlyInitializingMeta = nullptr;
    }
    instance->Meta = meta;
}
```

YsonStruct: наследование — инициализация

```
TMeta* CurrentlyInitializingMeta = nullptr;

template <class TStruct>
void TRegistry::Initialize(TStruct* instance) {
    if (CurrentlyInitializingMeta) {
        TStruct::Register(TRegistrar<TStruct>(CurrentlyInitializingMeta));
        return;
    }

    static TMeta* meta = nullptr;
    if (!meta) {
        meta = new Meta();
        CurrentlyInitializingMeta = meta;
        TStruct{}; // проваливаемся в первый if
        CurrentlyInitializingMeta = nullptr;
    }
    instance->Meta = meta;
}
```

YsonStruct

- Оптимизировали конструкторы и деструкторы
- Работает медленнее, чем раньше!
- Профилировщик: `dynamic_cast`
- `dynamic_cast` вынужден обходить всю иерархию
    - cast не к предку/потомку
    - несколько подходящих предков: реализация `dynamic_cast` должна найти всех и вернуть `nullptr`

YsonStruct: кеширование dynamic cast

- Оптимизируем `dynamic_cast<TStruct*>(base)`
- Запомним смещение для каждого целевого `TStruct` относительно базового `TYsonStruct`

```
template <class TTargetStruct>
TTargetStruct* CachedDynamicCast(TYsonStruct* source) {
    static std::optional<ptrdiff_t> cachedOffset;

    if (!cachedOffset) {
        auto* target = dynamic_cast<TTargetStruct*>(source);
        cachedOffset = reinterpret_cast<intptr_t>(target) - reinterpret_cast<intptr_t>(source);
    }

    return reinterpret_cast<TTargetStruct*>(reinterpret_cast<intptr_t>(source) + *cachedOffset);
}
```

- Это не работает с виртуальным наследованием!

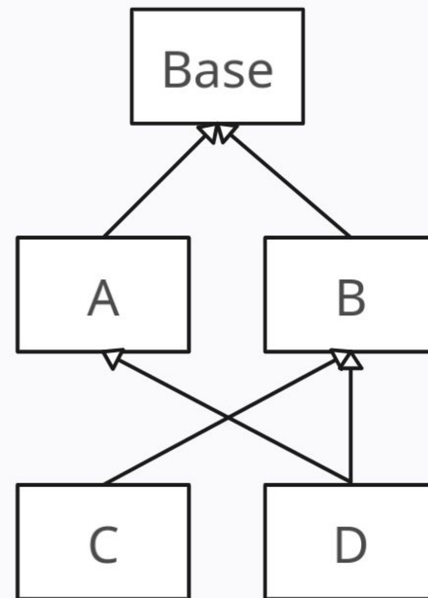
YsonStruct: кеширование dynamic cast

Смещение зависит от реального типа `source`!

```
struct A : public virtual Base {};
struct B : public virtual Base {};
struct C : public B {};
struct D : public A, public B {};

Base* c = new C;
Base* d = new D;
cout << reinterpret_cast<intptr_t>(dynamic_cast<B*>(c)) -
    reinterpret_cast<intptr_t>(c) << "\n";
cout << reinterpret_cast<intptr_t>(dynamic_cast<B*>(d)) -
    reinterpret_cast<intptr_t>(d) << "\n";

// 0
// 8
```



YsonStruct: кеширование dynamic cast

- Добавим в ключ кеширования `type_index` экземпляра
- Теперь всё корректно, если все касты идут от `TYsonStruct`

```
template <class TTargetStruct>
TTargetStruct* CachedDynamicCast(TYsonStruct* source) {
    static std::map<std::type_index, ptrdiff_t> cache;

    std::type_index type(typeid(*source));
    if (!cache.contains(type)) {
        auto* target = dynamic_cast<TTargetStruct*>(source);
        cache[type] = reinterpret_cast<intptr_t>(target) - reinterpret_cast<intptr_t>(source);
    }
    return reinterpret_cast<TTargetStruct*>(reinterpret_cast<intptr_t>(source) + cache[type]);
}
```

- Это не работает *в конструкторах!*

YsonStruct: кеширование dynamic cast

- `typeid` в конструкторе и деструкторе всегда возвращает тип текущего объекта, даже если это не наиболее вложенный тип
- То же верно про виртуальные функции в конструкторе и деструкторе
- Запрещаем использовать `IParameter` в конструкторах

YsonStruct: значения по умолчанию

- Установка значений по умолчанию

```
void TServerConfig::Register(TRegistrar registrar) {
    registrar.Parameter("host_name", &TThis::HostName)
        .Default("ytsaurus.tech");
}

// Конструктор скрыт за REGISTER_YSON_STRUCT
TServerConfig::TServerConfig() {
    for (const auto& [key, parameter] : Meta->Parameters) {
        parameter->SetDefault(this);
    }
}
```

- Не можем делать `parameter->SetDefault` в конструкторе
    - `dynamic_cast` — дорого
    - `CachedDynamicCast` не работает

YsonStruct: значения по умолчанию

- Не можем делать `parameter->SetDefault` в конструкторе
  - `dynamic_cast` — дорого
  - `CachedDynamicCast` не работает
- Решение 1: контролируем места создания и вызываем `instance->SetDefaults()` после создания экземпляра
  - своя подсистема аллокации объектов
  - фабричный метод
- Решение 2
  - знаем размер иерархии
  - `++counter` в каждом конструкторе
  - смогли детектировать конструктор финального класса

YsonStruct: нераспознанные параметры

- Пример применения рефлексии
- Словарь параметров, которые не смогли распознать
    - опечатки
    - устаревшие опции

YsonStruct: заметки на полях

- Потокобезопасность
  - инициализация меты
  - кеши
- Оптимизация времени компиляции
  - type erasure для `TParameter<TStruct, TValue>`

YTsaurus на github

https://github.com/ytsaurus/ytsaurus

```
yt/yt/core/ytree/yson_struct.h
yt/yt/core/ytree/yson_struct.cpp
```


автор библиотеки

Ренат Хайретдинов

докладчик

Иван Смирнов