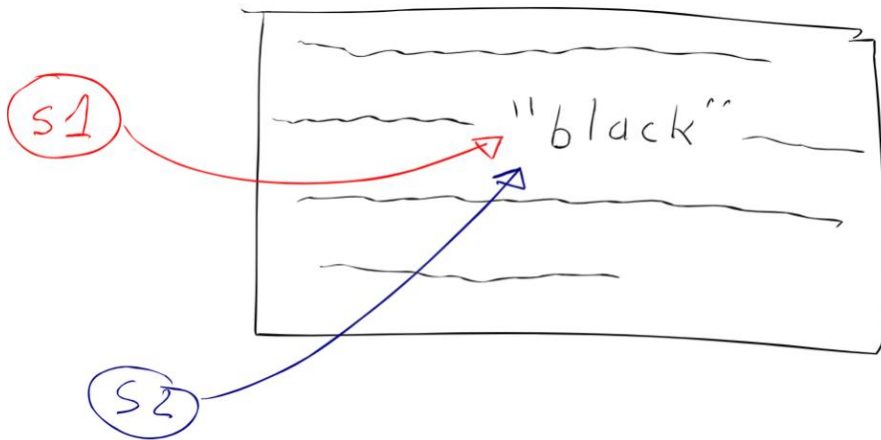


String interning

и все, все, все

Что такое string interning

«String interning», иногда это называют «пулом строк» — это оптимизация, при которой хранится только одна копия строки, независимо от того, сколько раз программа ссылается на нее



Идея

Идея пула строк возникла ещё в 1950–1960-х годах вместе с первыми языками высокого уровня — *Lisp* и *Smalltalk*

Считается, что концепцию ввёл Джон Маккарти (John McCarthy), создатель Lisp (~1958) в одной из реализаций

Первые систематические применения можно найти в реализации интерпретаторов Lisp, где **СИМВОЛЫ** (идентификаторы и имена функций) представлялись не как строки, а как уникальные объекты, хранящиеся в специальной таблице — области символов (symbol table)

Первые реализации

Последовательная неизменяемая таблица соответствия и было ранней формой string interning, строки не привязывались к каким то хешам, контрольной сумме, а просто нумеровались по мере того как интерпретатор встречал их в коде.

Цель - экономия памяти и ускорение сравнения строк

Оптимизация - если набор символов уникален, то его сравнение можно делать по индексу, а не по содержимому

💡 Позднее приём переключал в другие языки

Lua

Java

C#

Python и другие

```
PLANET → 0  
OWNER  → 1  
PLANET (again) → 0
```

Symbol table:

```
0: PLANET  
1: OWNER
```

В и С не как все

Автор языка С благополучно забыл о старой идее «интернированных символов» и подарил миру строки, которые все упорно сравнивают посимвольно — по данным, а не по идентичности

Изначально язык С вообще не имел полноценной модели строк как отдельного типа

Это наследие языка **B**, а тот — языка **BCPL**, где строки тоже представлялись как последовательности символов в памяти и никакого *string interning* не было: две одинаковые строки в коде просто занимали две области памяти, и их сравнение требовало прохода по всем байтам

С++ и тяжелое детство

В **С** строки — это просто `char*`, а не объект. Они не знают своей длины, не владеют памятью и не умеют себя копировать. Любое неосторожное обращение — и происходит все что угодно.

С++ унаследовал этот багаж. Первые версии языка (до стандарта 1998 года) не имели полноценного `std::string` — все пользовались `char*`, массивами и сторонними обёртками.

Появление `std::string` в STL стало мини-революцией - строки стали владеть своей памятью, копироваться, сравниваться, иметь длину и работать как объекты.

Но “детство” не прошло бесследно:

- `std::string` до сих пор хранит байты, просто байты, мы можем делать с ними что угодно
- строки путают с `const char*` (неявные преобразования и API на границе с **С**)
- годами наработанные в **С** правила долго мешали воспринимать `std::string` как безопасный инструмент

Чехарда в коде

Строки это много возможностей выстрелить по ногам

```
// Количество копий: 2 в rodata, 2 в RAM
```

```
const char* s1 = "black"; <<< это не black в s2 (не всегда, но лучше не рисковать)
```

```
const char* s2 = "black"; <<< это не black в s1
```

Чехарда в коде

Строки это много возможностей выстрелить по ногам

```
// Количество копий: 1 в rodata, ??? в RAM
```

```
const char* s1 = "black"; <<< это не black в s2 (не всегда, но лучше не рисковать)
```

```
const char* s2 = "black"; <<< это не black в s1
```

```
// Количество копий: 1 в rodata, 3 в RAM
```

```
const char s1[] = "black"; <<< где-то на стеке или в объекте
```

```
const char s2[] = "black"; <<< и это также не black в s1
```


Чехарда в коде

Строки это много возможностей выстрелить по ногам

```
// Количество копий: 1 в rodata, ??? в RAM
const char* s1 = "black"; <<< это не black в s2 (не всегда, но лучше не рисковать)
const char* s2 = "black"; <<< это не black в s1

// Количество копий: 1 в rodata, 3 в RAM
const char s1[] = "black"; <<< где-то на стеке или в объекте
const char s2[] = "black"; <<< и это также не black в s1

// Количество копий: 1 в rodata, сколько в RAM будет ???
// если такое положить в хедере + аллокации
const string s1 = "black";
const string s2 = "black";
```

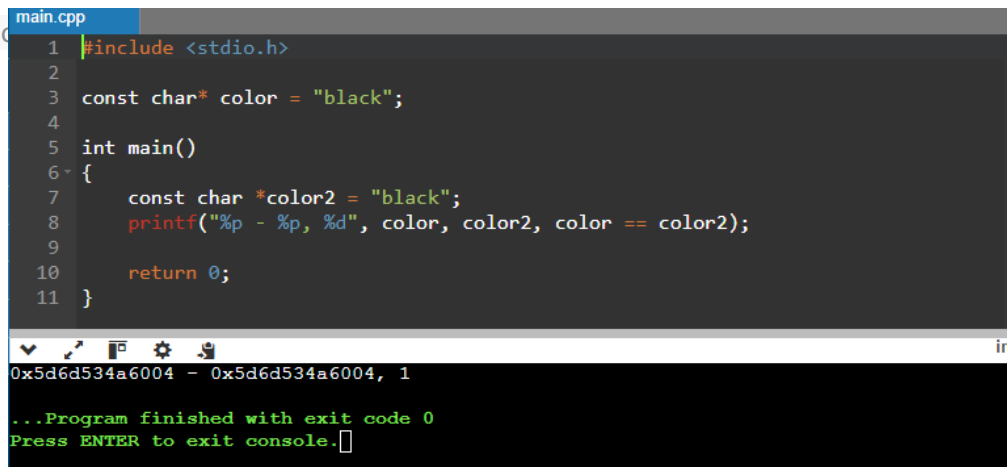
Как сравнивать обычные строки

```
const char* color = "black";  
if (color == "black") { // ???  
    // ...  
}
```

Как сравнивать обычные строки

Если компилятор умный и умеет в пул строк

```
const char* color = "black";  
  
if (color == "black") { // true, потому что  
    // ...  
}
```



```
main.cpp  
1 #include <stdio.h>  
2  
3 const char* color = "black";  
4  
5 int main()  
6 {  
7     const char *color2 = "black";  
8     printf("%p - %p, %d", color, color2, color == color2);  
9  
10    return 0;  
11 }
```


0x5d6d534a6004 - 0x5d6d534a6004, 1

...Program finished with exit code 0
Press ENTER to exit console.

Как сравнивать обычные строки

Если компилятор умный и умеет в пул строк

```
const char* color = "black";  
if (color == "black") { // true, потому что работает интернирование  
    // ...  
}
```



```
main.cpp  
1 #include <stdio.h>  
2  
3 const char* color = "black";  
4  
5 int main()  
6 {  
7     const char "color2" = "black";  
8     printf("%p - %p, %d", color, color2, color == color2);  
9  
10    return 0;  
11 }
```

0x5d6d534a6004 - 0x5d6d534a6004, 1
...Program finished with exit code 0
Press ENTER to exit console.

00007FF6E0C36320 - 00007FF6E0C36328, 0

Как правильно сравнивать строки

```
const char color[] = "black";  
if (color == "black") { // false, потому что color хранит копию  
    // ...  
}
```

Как правильно сравнивать строки

```
const char color[] = "black";  
if (color == "black") { // false, потому что color хранит копию  
    // ...  
}
```

Теперь все будет работать всегда и везде, но приходится сравнивать символы

```
const char color[] = "black";  
if (strcmp(color, "black") == 0) { // true, потому что  
сравнивается каждый символ  
    // ...  
}
```

Как правильно сравнивать строки

```
const char color[] = "black";  
if (color == "black") { // false, потому что color хранит копию  
    // ...  
}
```

Теперь все будет работать всегда и везде, но приходится сравнивать символы

```
const char color[] = "black";  
if (strcmp(color, "black") == 0) { // true, потому что  
    // ...  
}
```

- Читаемость уменьшилась
- Ошибки и надо помнить правила работы со строками
- Очень низкий перф, особенно на синтаксически похожих данных

Как это все влияет на время кадра

Это соотношение строк разных размеров в AAA проекте

Component	Unique Allocations
short strings (< 80 chars) Ключ, Идентификаторы	1218 KB (19,187)
middle strings (< 256 chars) Комментарии, Реплики	2185 KB (10,349)
long strings (< 1k) Описания предметов	454 KB (635)
very long string (< 4k) Исторические справки	828 KB (393)

Как это все влияет на время кадра

Нас интересует категория ключи и идентификаторы

Подкатегория	Длина (символов)	Кол-во сравнений на кадре	Сравнение (3Ghz)	Время всех сравнений
До 16	16	194,000	$16 * 0.66 \approx 10.6 \text{ нс}$	$194,000 * 10.6 \text{ нс} \approx 2.06 \text{ мс}$
До 32	32	74,000	$32 * 0.66 \approx 21.1 \text{ нс}$	$74,000 * 21.1 \text{ нс} \approx 1.56 \text{ мс}$
До 64	64	30,500	$64 * 0.66 \approx 42.2 \text{ нс}$	$30,500 * 42.2 \text{ нс} \approx 1.29 \text{ мс}$
До 80	80	16,000	$80 * 0.66 \approx 52.8 \text{ нс}$	$16,000 * 52.8 \text{ нс} \approx 0.84 \text{ мс}$

Откуда столько сравнений строк

Обработка событий

В кадре проверяется множество событий: пользовательские действия, AI-состояния, триггеры, скриптовые события. Каждый раз движок ищет ключи или проверяет имя события → строковые сравнения. Если есть 1000 объектов, каждый с 10–20 проверками — уже десятки тысяч сравнений

- Под конец игры у нас порядка 6000 объектов и ~50 сравнений в каждом

Обновление объектов

Каждому объекту нужно проверить параметры: имя, состояние, тип действия.

- часто сравниваются короткие идентификаторы или ключи словаря.

Для большого числа объектов (юниты, предметы, бафы)

- это накапливается в прогрессии

Скрипты и диалоговые системы

Любая система реплик, диалогов, реакций использует строки для ключей и условий

- мы не можем заставить модеров использовать числа
- тогда они не будут писать моды, поэтому мы даем им возможность сравнивать строки, так во всех играх

Например: `if(event == "pickup_item") {...}` → сравнение строк.

Модов могут быть сотни, и такие проверки за кадр легко дают десятки тысяч сравнений.

Возвращаемся к сравнению строк

А теперь представьте что любую строки мы можем сравнивать за константное время

Подкатегория	Длина (символов)	Кол-во сравнений на кадр	Сравнение (3 GHz)	Время всех сравнений	Время (было)
До 16	16	194 000	0.66 нс	$194\,000 \times 0.66 \text{ нс} \approx 0.128 \text{ мс}$	2.06 мс
До 32	32	74 000	0.66 нс	$74\,000 \times 0.66 \text{ нс} \approx 0.049 \text{ мс}$	1.56 мс
До 64	64	30 500	0.66 нс	$30\,500 \times 0.66 \text{ нс} \approx 0.020 \text{ мс}$	1.29 мс
До 80	80	16 000	0.66 нс	$16\,000 \times 0.66 \text{ нс} \approx 0.011 \text{ мс}$	0.84 мс

Как добиться такого поведения (xstring)

Как идея это выглядит довольно несложно

это простая таблица поиска, которая сопоставляет идентификаторы строкам

```
namespace utils {  
    // таблица для хранения таких строк, мы можем даже не удалять их  
    struct xstrings { hash_map< uint32_t, string > _interns; };  
  
    namespace strings {  
        // функция для встраивания новых строк  
        uint32_t Intern( xstrings &strs, const char *str );  
  
        // функция для получения данных строки  
        const char* GetString( xstrings &strs, uint32_t id );  
  
    }  
}
```

Логика работы со xstring

1. Intern(strs, "enemy_spotted")

- Вычисляется `crc = CRC32("enemy_spotted")`.
- Ищем `crc` в `strs._interns`.
 - Если найден, `reference++`, возвращаем `crc`.
 - Если не найден, создаём `xstring_value`:
 - `crc = хэш строки`
 - `reference = 1`
 - `length = strlen(str)`
 - `value = str`
 - добавляем в таблицу `_interns[crc] = value;`
- Возвращаем `crc` как ID строки.

Реализация xstring

```
struct xstring_value {  
    uint32_t crc;          // контрольная сумма строки  
    uint16_t reference;    // сколько осталось ссылок  
    uint16_t length;       // длина строки  
    std::string value;     // сырые данные, сейчас тут std::string  
                           // но могут быть любые структуры, заточенные под проект  
};  
  
std::map<uint32_t, xstring_value*> data;
```

Синтаксический сахар

На этом можно было остановиться, но душа просит красотостей, прячем всю работу с таблицей

```
struct xstring {
    xstring_value* _p;                // Указатель на разделяемое значение строки
                                      // используем для быстрого доступа к данным

    void _dec() {                     // Уменьшает счётчик ссылок на текущее значение строки
                                      // и обнуляет указатель, если счётчик стал равен нулю.

        if (0 == _p)
            return;

        _p->reference--;               // Уменьшаем количество ссылок на строку
        if (0 == _p->reference)        // Если больше никто не использует строку
            _p = 0;                   // освобождаем ссылку (но сама память не освобождается)
    }

    xstring_value *_dock(pcstr value); // Возвращает (или создаёт) объект xstring_value
    void _set(pcstr rhs) {              // Устанавливает новое значение строки
        xstring_value* v = _dock(rhs); // Получаем (или создаём) строковое значение из пула
        if (0 != v) {
            v->reference++;             // Увеличиваем счётчик ссылок на новую строку
        }
        _dec();                       // Уменьшаем ссылку на старое значение (если было)
        _p = v;                       // Переключаемся на новое значение
    }
}
```

Как пользоваться

Допустим мы делаем некоторую реакцию на события в игре

```
struct time_tag {  
    float12 time;  
    xstring tag;  
};  
  
void ai_unit::on_event(const time_tag &tt) {  
    // Проверяем, соответствует ли тег событию "aim_walk"  
    if (tt.tag == "aim_walk") {  
        ai_start_walk(tt);           // Если да – запускаем поведение "начать идти"  
    } else if (tt.tag == "aim_turn") {  
        ai_start_turn(tt);           // Или – запускаем поведение "начать поворот"  
    }  
    ai_unit_base::on_event(tt);      // Передаём событие дальше по иерархии (базовому классу)  
}
```


Как пользоваться. Заметили проблему?

Допустим мы делаем некоторую реакцию на события в игре

```
struct time_tag {  
    float12 time;  
    xstring tag;  
};  
  
void ai_unit::on_event(const time_tag &tt) {  
    // Проверяем, соответствует ли тег событию "aim_walk"  
    if (tt.tag == "aim_walk") {        ??????????  
        ai_start_walk(tt);             // Если да – запускаем поведение "начать идти"  
    } else if (tt.tag == "aim_turn") {  ??????????  
        ai_start_turn(tt);             // Или – запускаем поведение "начать поворот"  
    }  
    ai_unit_base::on_event(tt);        // Передаём событие дальше по иерархии (базовому классу)  
}
```

Как пользоваться. Заметили проблему?

Допустим мы делаем некоторую реакцию на события в игре

```
struct time_tag {  
    float12 time;  
    xstring tag;  
};  
  
void ai_unit::on_event(const time_tag &tt) {  
    // Проверяем, соответствует ли тег событию "aim_walk"  
    if (tt.tag == "aim_walk") { <<<<< вот тут будет конвертация из строки и поиск по таблице  
        ai_start_walk(tt); // Если да – запускаем поведение "начать идти"  
    } else if (tt.tag == "aim_turn") { <<<<< и тут будет тоже  
        ai_start_turn(tt); // Или – запускаем поведение "начать поворот"  
    }  
    ai_unit_base::on_event(tt); // Передаём событие дальше по иерархии (базовому классу)  
}
```

Как пользоваться правильно

```
struct time_tag {  
    float12 time;  
    xstring tag;  
};  
  
struct events {  
    static const xstring aim_walk; <<<<< фактически int64  
    static const xstring aim_turn;  
};  
  
void ai_unit::on_event(const time_tag &tt) {  
    if (tt.tag == events::aim_walk) { <<<< только сравнение int64  
        ai_start_walk(tt);  
    } else if (tt.tag == events::aim_turn) { <<<< только сравнение int64  
        ai_start_turn(tt);  
    }  
    ai_unit_base::on_event(tt);  
}
```

Как пользоваться правильно

```
struct events {  
    static const xstring aim_walk; <<<<< фактически int64  
    static const xstring aim_turn;  
};  
void ai_unit::on_event(const time_tag &tt) {  
    if (tt.tag == events::aim_walk) { <<<< только сравнение int64  
        ai_start_walk(tt);  
    } else if (tt.tag == events::aim_turn) { <<<< только сравнение int64  
        ai_start_turn(tt);  
    }  
    ai_unit_base::on_event(tt);  
}
```

- + очень мало индирекций и слома кэша
- + очень простые и прогнозируемые переходы
- + “короткая” логика и простые сравнения

Известные мне реализации

foonathan/string_id

https://github.com/foonathan/string_id

libstringintern

<https://github.com/RipcordSoftware/libstringintern>

stb

https://graemephi.github.io/posts/stb_ds-string-interning

akhenaten

<https://github.com/dalerank/Akhenaten/blob/master/src/core/xstring.h>

К чему привели пулы строк?

Очень интересный вопрос, потому что сами пулы строк породили много смежных решений

Хешированные идентификаторы и таблицы строк

Где

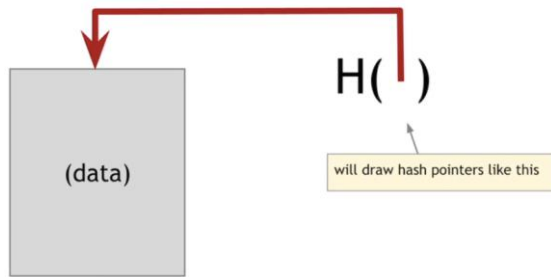
компиляторы Lisp, Smalltalk, позже C/C++ и Java.

Как

храним каждую уникальную строку (имя переменной, метода, тега) один раз
работаем с её идентификатором (указателем или хешем)

Влияние

ускоряет сравнение идентификаторов (== вместо strcmp)
основа для таблиц символов (symbol tables) в компиляторах
породило типы вроде `Symbol` в Ruby, `atom()` в Erlang, `intern()` в Java



Resource ID и StringID

Где

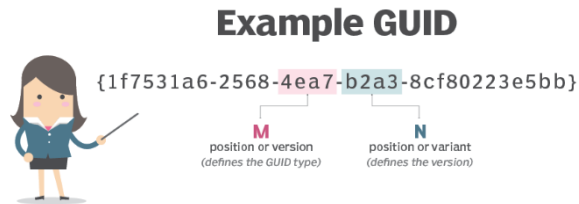
движки вроде Unreal, CryEngine, Dagor, Frostbite

Как

текстовые имена (материалы, анимации, звуки) заменяются на числовые идентификаторы
часто CRC32 или хеши

Зачем

устранение затрат на строковые сравнения в рантайме
возможность сериализации ресурсов по ID, без хранения строк
использование предвычисленных хешей в пайплайне сборки контента



String Pools и Flyweight паттерн

Где

паттерны проектирования (GoF, 1990-е).

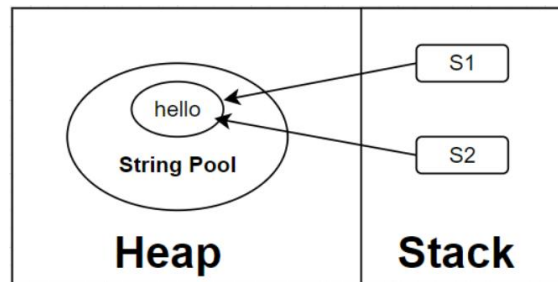
Как

разделять неизменяемые данные (например, одинаковые строки), чтобы не дублировать память

Зачем

база для оптимизации постоянных таблиц (constant pools) в Java и .NET

экономим гигабайты в системах, где строки повторяются много раз (базы данных)



Immutable/Interned объекты

Где

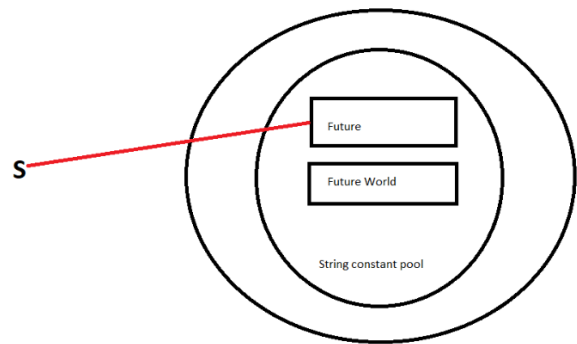
Java, .NET, Python.

Как

интернированные строки неизменяемы
можно безопасно кэшировать и делить между потоками.

Зачем

все эти механизмы экономят память и ускоряют сравнение ключей (при парсинге JSON или XML).



ECS системы

Где

современные игровые движки (Unity DOTS, Frostbite, Snowdrop).

Как

строки (или их ID) стали частью **компонентных систем**

строки заранее заменяются ID

ID могут индексировать массивы данных

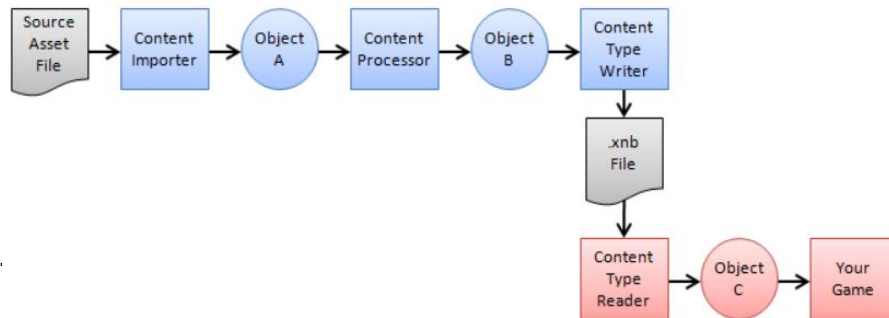
сравнение → одно число, без доступа к heap или таблице символов

Зачем

развитие идеи "interning" в сторону **data locality** и **cache-friendly** дизайна

	Entity 1	Entity 2	Entity 3
Component 1	[data]	[data]	[data]
Component 2	[data]	[data]	[data]
Component 3	[data]	[data]	[data]

Content pipeline



Где

инструменты Unreal/Unity, пайплайны AAA студий.

Как

строки (теги, имена свойств, имена костей, звуки) заранее "запекаются" в таблицы ID

→ текст вообще не нужен в рантайме.

Зачем

меньше памяти и строковых сравнений

стабильные бинарные интерфейсы между данными и кодом

возможность детерминированных билдов и контентных диффов

Reflection и RTTI через StringID

Где

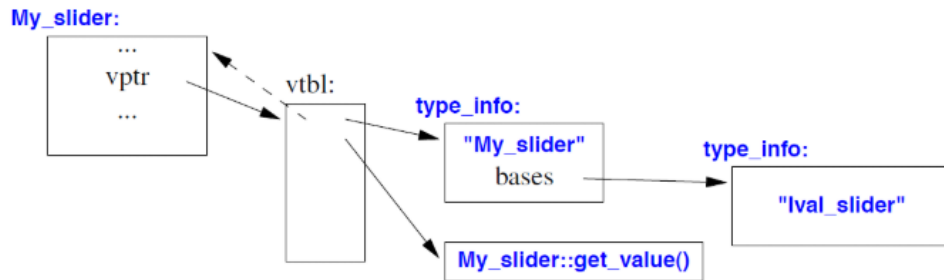
Современные движки и фреймворки (Unreal, Entt, Frostbite):

Как

типы, поля и функции имеют интернированные имена/StringID

Зачем

доступ к метаданным и свойства через ID



Сериализация и Репликация данных

Системы сериализации и network replication

Интернирование строк позволяет передавать по сети не строки, а их ID

Уменьшает размер пакетов и синхронизацию строковых данных

Используется в репликации свойств, RPC и UI системах.

