ОПТИМИЗАЦИЯ ПРОТОКОЛА СЕРИАЛИЗАЦИИ ОБЪЕКТОВ В ТРЕЙДИНГОВЫХ СИСТЕМАХ НА ОСНОВЕ СОБИРАЕМОЙ В RUNTIME СТАТИСТИКИ

Раков И. С., магистрант кафедры КТ, Университет ИТМО
ivan.glukos@gmail.com

Коковцев Д. Г., инженер-программист, ООО «Эксперт-Система»
dmkokovtsev@devexperts.com

Аннотация

В трейдинговых системах между узлами проходит огромное количество сетевого трафика. При определенных событиях физический канал может быть использован на 100%, что ведет к просадке производительности.
В работе предлагается универсальное решение, позволяющее уменьшить количество байт, передаваемое по сети при сериализации объектов. Получен результат лучше, чем у стандартных энтропийных/словарных алгоритмов сжатия.

Введение

Серверная часть современных трейдинговых систем представляет собой множество модулей, каждый из которых работает на отдельной машине в кластере. Для такой системы характерны очень большие объемы данных, передаваемые как в сетевых соединениях с внешним миром (ежесекундно меняющиеся котировки на все торгуемые финансовые инструменты, сообщения от биржи об изменении статуса заявок), так и в соединениях между узлами (распространение полученных «внешних» данных по всем заинтересованным узлам, изменение статуса аккаунтов, пересчет внутренних метрик, аудит).

Если проанализировать трафик (в качестве объекта анализа была использована трейдинговая система Thinkorswim), можно выделить его две основные составляющие:

- Рыночные данные (котировки на инструменты, волатильность и другие рассчитываемые метрики). Этот тип данных очень чувствителен к задержкам — поступившая на несколько секунд позже котировка может заставить трейдера принять ошибочное решение. В количественном соотношении рыночные данные составляют небольшую долю байт (<20%).
- DTO (Data Transfer Objects, объекты передачи данных),

сохраненные в поток байт сущности (ордера, счета, балансы и т.п.) Передаваемые по сети DTO составляют основную часть трафика (>80%) и менее чувствительны к задержкам (короткие паузы до секунды нежелательны, но допустимы). Кроме того, активность передачи DTO в системе имеет всплесковый характер. Пример: администратор меняет глобальные параметры расчета маржи для фьючерсов, после чего свежие параметры должны как можно быстрее попасть на все сервера (несколько десятков серверов в кластере), а каждый сервер должен разослать параметры каждому подключенному к нему клиенту (балансировщик нагрузки работает так, что каждый сервер Thinkorswim держит 3000-4000 подключений к клиентам). В результате, сетевые соединения в кластере будут работать на физическом пределе (1 gbps) в течение нескольких секунд. В этот период система перестает отвечать возложенным на неё требованиям по производительности и времени отклика.

Если уменьшить количество байт, которое занимает DTO, то можно добиться существенного уменьшения просадки системы в критические моменты. Использование алгоритмов сжатия неизбежно приведет к увеличению задержки (latency), но увеличение пропускной способности (throughput) в данном случае приоритетнее.

Целью работы является поиск технологического решения, которое позволит наиболее эффективно увеличить пропускную способность передачи DTO по сети.

Стандартные алгоритмы сжатия

Существующие словарные (RLE, LZ) и энтропийные (Huffman, Shannon-Fano, PPMA) алгоритмы сжатия успешно работают на DTO (лучший средний множитель сжатия 6 для алгоритма DEFLATE[7]).

Однако, все перечисленные выше алгоритмы уменьшают непредсказуемость (энтропию) информации, используя только предыдущую часть потока байт (на самом деле не обычно всю, а последнюю его часть фиксированной длины, так называемый «контекст»).

В случае с DTO, у нас есть доступ не только к предыдущему куску данных. Информация становится «размеченной» — при передаче очередного объекта мы точно знаем, сколько у него полей, какой у них тип данных, и каких именно данных следует ожидать. Все эти знания можно использовать, чтобы уменьшить энтропию еще сильнее, а следовательно, уменьшить итоговое количество байт в сжатом виде.

В работах[3-5] были показаны подходы к увеличению пропускной способности сериализации объектов. Однако, оптимизации были больше направлены не на уменьшение непредсказуемости значений полей объектов, а на избавление от лишних метаданных в протоколах и устранение изъянов конкретных языков программирования и технологий.

Pack 200

Pack 200[1] — алгоритм сжатия от Oracle, разработанный специально для сжатия байт-кода JVM. Процесс упаковки байт-кода в .jar выглядит следующим образом:

.class files → Pack200 → Tar → Gzip → .jar file

Pack 200 как уменьшает чистое количество байт, так и увеличивает «сжимаемость» данных стандартными архиваторами (как это делает, например, BW-преобразование).

Pack 200 успешно работает за счет того, что подаваемые ему на вход данные специфичны — это всегда Java байт-код. Данные в DTO тоже специфичны, поэтому некоторое подходы из Pack 200 могут оказаться полезными при сжатии DTO.

Помимо специфичных непосредственно для байт-кода техник (banding – группировка похожих данных: названия полей и классов — отдельно, сигнатуры методов — отдельно, константы — отдельно; индексирование структуры java packages; перегруппировка байт-кода для уменьшения расстояния между инструкциями и т. п.), в Pack200 разработаны алгоритмы эмпирического кодирования констант и целых чисел, некоторые из которых будут рассмотрены ниже.

Семейство кодирований целых чисел BHSD

BHSD — семейство алгоритмов для кодирования коррелированных последовательностей целых чисел. У кодирования есть четыре параметра:

- *B* — максимальная длина. В худшем случае запись числа будет занимать *B* байт
- *H* — показатель степени позиционной системы счисления
- *S* — количество бит, отведенное под знак. Если в первых *S* битах '1', то следующее за *S* битами число считается отрицательным. Если хотя бы один бит '0', то считается, что с самого начала записано положительно число
- *D* — равно 1, если дельта-кодирование [6] используется, иначе 0.

Целое число всегда закодировано последовательностью $\{b_1, b_2, \ldots, b_n\}$.

$L = 256 - H$, $b_i \geq L$, $b_n < L$

Пример при $H = 64$:

- число 191 имеет вид {191},
- число 192 имеет вид {192, 128},
- число 193 имеет вид {192, 129}.

Для примера, в Pack 200 для кодирования небольших по модулю целых чисел используется алгоритм BHSD(5,64,1,0), а для кодирования монотонных последовательностей — BHSD(5,64,0,1). Всего в Pack 200 используется 115 различных вариантов BHSD.

В DTO так же, как и в байт-коде, присутствует много целочисленных данных, на которых использование BHSD оказывается эффективным.

FTU-преобразование

Для кодирования последовательностей значений, где часто встречаются повторы, в Pack200 используется FTU-преобразование: выделяется множество «популярных» значений *F*, все остальные значения автоматически принадлежат к множеству «непопуярных» значений *U*. Итоговая последовательность *{a}* после преобразования выглядит следующим образом:

$\{a\} \rightarrow \{F\} + \{Tokens\} + \{U\}$

- $Tokens[i] = j$, *if* $a[i] \in F$, $Tokens[i] = F[j]$
- $Tokens[i] = -1$, *if* $a[i]$ *in* U

Использованный подход

В системе Thinkorswim используется стек технологий Java + RMI, поэтому DTO — сериализованные в соответствии с Java Serialization API объекты.

Java Serialization API предлагает интерфейс для написания пользовательского протокола сериализации. Для этого необходимо реализовать два метода в классе:

```
private void writeObject(..) {
    IOUtil.writeCompactInt(out, year);
    …
}
```

```
private void readObject(...) {
    year = IOUtil.readCompactInt(in);
    …
}
```

В методе writeObject есть доступ к байтовому потоку вывода `out`. Разработчик может записать в этот поток всю информацию об объекте оптимальным способом.

В методе readObject, соответственно, есть доступ к потоку ввода `in`, используя данные из которого нужно восстановить состояние объекта.

EGEN – утилита для оптимизации во время компиляции

EGEN [2] — процессор аннотаций, генерирующий методы writeObject() и readObject().

Вместо ручного написания методов, можно расставить аннотации у полей класса, и интегрировать EGEN в процесс компиляции — утилита сама сгенерирует методы writeObject/readObject().

Предлагаемые аннотации: @Compact (небольшое число), @PresenceBit (число со значениеми по умолчанию), @Inline, @Delta (разность с константой/другим полем вместо числа), @Decimal (если в double хранится десятичная дробь, то она передается как целое число).

Массив сериализованных инструментов (объектов класса Instrument, торгуемых финансовых инструментов) с ручной расстановкой аннотаций занимает в 2 раза меньше байт, чем массив, сериализованный стандартным способом. Чистое относительное улучшение составило 1.3.

Для расчета чистого относительного улучшения сравнивался размер сжатого с помощью DEFLATE массива инструментов с размером массива инструментов, сериализованного через EGEN и после этого сжатого с помощью DEFLATE. Это более справедливая метрика, чем просто сравнение количества байт, так как она учитывает только те оптимизации, которые не нивелируются стандартными алгоритмами сжатия.

Sprut – Stream Profiling Util

С расставлением аннотаций EGEN есть одна проблема — надо понимать структуру класса и то, какие значение обычно бывают в каждом поле. Если в трейдинговой системе по сети передаётся несколько сотен классов, то собрать эту информацию вручную — слишком трудоемкий процесс.

Чтобы автоматизировать этот процесс, был разработан профилировщик сериализации Sprut. Он представляет собой Java-агент, добавляющий в классы JDK, отвечающие за стандартную сериализацию (java.io.ObjectOutput[Input]Stream), код сбора статистики. Код добавляется на этапе запуска Java-машины.

Если sprut включен на работающем приложении, то для каждого целевого класса собирается статистическая информация (мат. ожидание значения, дисперсия значения, k самых популярных значений) о всех его полях. Пример:

```
class OrderLeg { int x1; int x2; String desc … }


x1: expected = 124, variance = 20.5,
top = 124(5), 134(3), 133(2), 130(1), ...
x2: expected = 100, variance = 70.8,
top = 100(10), 90(2), 95(1), 99(1), ...
desc: top = 'accepted'(50), 'pending'(20), ...
(x1 - x2): expected = 20, variance = 10.5, …
```

Рекомендация аннотаций на основе статистики Sprut

На основе собранной в production статистики можно сформулировать рекомендации, какие аннотации лучше подходят к конкретному полю класса.

На данный момент выделены следующие жадные рекомендации по аналогии с Pack 200:

- объект встречается один раз за сессию – @Inline,
- double-поле всегда хранит десятичную дробь — @Decimal,
- среднее значение меньше, чем $H^{B/2}$ — BHSD(5,64,1,0),
- поле почти всегда близко по значению к другому полю — BHSD(5,64,1,1),
- в последовательности значений много повторов — FTU.

При расставленных на основе этих рекомендаций аннотациях массив инструментов сжимается с чистым относительным улучшением 1.1. Стоит заметить, что это хуже, чем при ручной расстановке аннотаций.

Заключение

Предложено техническое решение по облегчению сериализованного

DTO на основе собираемой в runtime статистики. По сравнению с подходом (DTO → Gzip) чистое относительное улучшение составляет 1.1. Предложенный подход применим в любых Java-приложениях, где для передачи данных активно используется сериализация объектов. В дальнейшем планируется улучшить алгоритм рекомендации аннотаций, чтобы приблизить его по эффективности к ручному выбору кодирования.

Литература

1. Orcale — Java SE Documentation, JSR-200 Public Draft Specification http://docs.oracle.com/javase/6/docs/technotes/guides/pack200/pack-spec.html
2. EGEN - Tool for compile-time optimization of Java classes serialization https://github.com/Devexperts/egen
3. Barış Aktemur, Joel Jones, Samuel Kamin, Lars Clausen Optimizing Marshalling by Run-Time Program Generation, Springer, 2005
4. Daniel Tejera, Predictable Serialization in Java, 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07), 2007
5. Zachary DeVito, First-class Runtime Generation of High-performance Types using Exotypes http://theory.stanford.edu/~aiken/publications/papers/pldi14b.pdf
6. F Douglis, A Iyengar, Application-specific Delta-encoding via Resemblance Detection, USENIX 2003 Annual Technical Conference, General Track — Paper
7. DEFLATE Compressed Data Format Specification version 1.3 http://tools.ietf.org/html/rfc1951