Блог Go

Карты Faster Go со швейцарскими таблицами

Майкл Пратт
26 февраля 2025 г.

Хэш-таблица является центральной структурой данных в информатике и обеспечивает реализацию типа карты во многих языках, включая Go.

Концепция хеш-таблицы была впервые описана Хансом Петером Луном в 1953 году во внутреннем меморандуме IBM. В нём предлагалось ускорить поиск, помещая элементы в «корзины» и используя связанный список для переполнения, когда корзины уже содержат элемент. Сегодня мы бы назвали это хеш-таблицей с цепочками .

В 1954 году Джин М. Амдал, Элейн М. Макгроу и Артур Л. Сэмюэл впервые применили схему «открытой адресации» при программировании IBM 701. Если в контейнере уже есть элемент, новый элемент помещается в следующий пустой контейнер. Эта идея была формализована и опубликована в 1957 году У. Уэсли Петерсоном в работе «Адресация для хранилищ с произвольным доступом» . Сегодня мы бы назвали это хеш-таблицей с открытой адресацией и линейным зондированием .

Учитывая, что структуры данных существуют уже так долго, легко подумать, что они уже «готовы к использованию», что мы знаем о них всё, и их больше нельзя улучшить. Это неправда! Исследования в области информатики продолжают совершенствовать фундаментальные алгоритмы, как с точки зрения алгоритмической сложности, так и использования возможностей современных процессоров. Например, в версии Go 1.19 пакет перешёл`sort` с традиционной быстрой сортировки на быструю сортировку, опровергающую шаблоны , — новый алгоритм сортировки, разработанный Орсоном Р. Л. Питерсом, впервые описанный в 2015 году.

Как и алгоритмы сортировки, структуры данных хэш-таблиц продолжают совершенствоваться. В 2017 году Сэм Бензакен, Алкис Эвлогименос, Мэтт Кулукундис и Роман Перепелица из Google представили новую разработку хэш-таблиц на C++ , получившую название «Швейцарские таблицы». В 2018 году их реализация была открыта в библиотеке Abseil C++ .

В Go 1.24 реализована совершенно новая реализация встроенного типа карт, основанная на швейцарской таблице. В этой статье мы рассмотрим, как швейцарские таблицы превосходят традиционные хеш-таблицы, а также некоторые уникальные проблемы, связанные с внедрением швейцарской таблицы в карты Go.

Открытая адресная хэш-таблица

Швейцарские таблицы представляют собой разновидность хэш-таблиц с открытой адресацией, поэтому давайте сделаем краткий обзор того, как работает простая хэш-таблица с открытой адресацией.

В хэш-таблице с открытой адресацией все элементы хранятся в одном резервном массиве. Мы будем называть каждое местоположение в массиве *слотом* . Слот, к которому принадлежит ключ, в первую очередь определяется *хэш-функцией* , `hash(key)`. Хэш-функция сопоставляет каждый ключ целому числу, где один и тот же ключ всегда сопоставляется одному и тому же целому числу, а различные ключи в идеале следуют равномерному случайному распределению целых чисел. Определяющей особенностью хэш-таблиц с открытой адресацией является то, что они разрешают коллизии, сохраняя ключ в другом месте резервного массива. Таким образом, если слот уже заполнен (коллизия) , то используется *последовательность зонда* для рассмотрения других слотов, пока не будет найден пустой слот. Давайте рассмотрим пример хэш-таблицы, чтобы увидеть, как это работает.

Пример

Ниже представлен 16-слотовый резервный массив для хэш-таблицы и ключ (если есть), хранящийся в каждом слоте. Значения не показаны, так как они не имеют отношения к данному примеру.

Слот	0	1	2	3	4	5	6	7	8	9	10
Ключ				56	32	21					

Чтобы вставить новый ключ, мы используем хеш-функцию для выбора слота. Поскольку слотов всего 16, нам нужно ограничиться этим диапазоном, поэтому мы используем `hash(key) % 16`в качестве целевого слота. Предположим, мы хотим вставить ключ 98и `hash(98) % 16 = 7`. Слот 7 пуст, поэтому мы просто вставляем туда 98. С другой стороны, предположим, что мы хотим вставить ключ 25и `hash(25) % 16 = 3`. Слот 3 является коллизией, поскольку он уже содержит ключ 56. Поэтому мы не можем выполнить вставку сюда.

Мы используем последовательность зондов для поиска следующего слота. Существует множество известных последовательностей зондов. Первоначальная и самая простая последовательность зондов — это *линейное зондирование* , которое просто проверяет последовательные слоты по порядку.

Итак, в этом `hash(25) % 16 = 3`примере, поскольку слот 3 занят, следующим мы рассмотрим слот 4, который также занят. Также занят и слот 5. Наконец, мы доберёмся до пустого слота 6, где сохраним ключ 25.

Поиск выполняется по тому же принципу. Поиск ключа 25 начинается со слота 3, проверяется, содержит ли он ключ 25 (его там нет), а затем продолжается линейное зондирование, пока ключ 25 не будет найден в слоте 6.

В этом примере используется резервный массив с 16 ячейками. Что произойдёт, если мы вставим больше 16 элементов? Если в хеш-таблице закончится место, она увеличится, обычно удваивая размер резервного массива. Все существующие записи будут перевставлены в новый резервный массив.

Открытоадресные хэш-таблицы фактически не ждут полного заполнения резервного массива, чтобы начать расти, поскольку по мере заполнения массива средняя длина каждой пробной последовательности увеличивается. В приведенном выше примере с использованием ключа 25 нам необходимо проверить 4 различных слота, чтобы найти пустой. Если бы в массиве был только один пустой слот, в худшем случае длина пробы была бы O(n). То есть, вам может потребоваться просканировать весь массив. Доля используемых слотов называется *коэффициентом загрузки* , и большинство хэш-таблиц определяют *максимальный коэффициент загрузки* (обычно 70–90%), при достижении которого они начинают расти, чтобы избежать чрезвычайно длинных пробных последовательностей, характерных для очень полных хэш-таблиц.

Швейцарский стол

Конструкция Swiss Table также является разновидностью хеш-таблицы с открытой адресацией. Давайте посмотрим, чем она превосходит традиционную хеш-таблицу с открытой адресацией. У нас по-прежнему есть один резервный массив для хранения данных, но мы разобьем его на логические *группы* по 8 ячеек в каждой. (Возможны и более крупные группы. Подробнее об этом ниже.)

Кроме того, каждая группа имеет 64-битное *управляющее слово* для метаданных. Каждый из 8 байтов управляющего слова соответствует одному из слотов в группе. Значение каждого байта указывает, пуст ли этот слот, удалён или используется. Если слот используется, байт содержит младшие 7 бит хэша ключа этого слота (называемого h2).

	Группа 0										
Слот	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**
Ключ	56	32	21						78		

	64-битное управляющее слово 0								**64-би**		
Слот	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**0**	**1**	**2**
h2	23	89	50						47		

Вставка работает следующим образом:

1. Вычислите `hash(key)`и разбейте хеш на две части: старшие 57 бит (называемые h1) и младшие 7 бит (называемые h2).
2. Верхние биты (h1) используются для выбора первой группы для рассмотрения: h1 % 2в данном случае, поскольку имеется только 2 группы.
3. Внутри группы все слоты имеют равное право на хранение ключа. Сначала необходимо определить, содержит ли уже какой-либо слот этот ключ. В этом случае это обновление, а не новая вставка.
4. Если ни один слот не содержит ключа, то мы ищем пустой слот для размещения этого ключа.
5. Если ни один слот не пуст, то мы продолжаем последовательность зондирования, выполняя поиск в следующей группе.

Поиск выполняется по тому же базовому алгоритму. Если на шаге 4 мы находим пустую ячейку, то мы знаем, что эта ячейка была бы занята вставкой, и можем прекратить поиск.

На третьем этапе происходит магия швейцарской таблицы. Нам нужно проверить, содержит ли какой-либо слот в группе искомый ключ. Наивно было бы просто выполнить линейное сканирование и сравнить все 8 ключей. Однако контрольное слово позволяет сделать это более эффективно. Каждый байт содержит младшие 7 бит хэша (h2) для этого слота. Если мы определим, какие байты контрольного слова содержат h2искомый ключ, мы получим набор потенциальных совпадений.

Другими словами, мы хотим выполнить побайтовое сравнение на равенство в управляющем слове. Например, если мы ищем ключ 32, где h2 = 89, то искомая операция будет выглядеть так.

Тестовое слово	89	89	89	89	89	89	89	89
Сравнение	==	==	==	==	==	==	==	==
Контрольное слово	23	89	50	-	-	-	-	-
Результат	0	1	0	0	0	0	0	0

Это операция, поддерживаемая аппаратным обеспечением SIMD , при которой одна инструкция выполняет параллельные операции над независимыми значениями внутри большего значения (*вектора*). В этом случае, если специальное аппаратное обеспечение SIMD недоступно, эту операцию можно реализовать с помощью набора стандартных арифметических и побитовых операций.

Результатом является набор слотов-кандидатов. Слоты, где хэш h2не совпадает, не имеют соответствующего ключа, поэтому их можно пропустить. Слоты, где хэш h2совпадает, являются потенциальными совпадениями, но мы всё равно должны проверить весь ключ, так как существует вероятность коллизий (вероятность коллизии с 7-битным хешем составляет 1/128, поэтому она всё ещё довольно низкая).

Эта операция очень мощная, поскольку мы фактически выполнили 8 шагов последовательности зондирования одновременно, параллельно. Это ускоряет поиск и вставку за счёт сокращения среднего количества необходимых сравнений. Это улучшение поведения зондирования позволило реализациям Abseil и Go увеличить максимальный коэффициент загрузки карт Swiss Table по сравнению с предыдущими картами, что снижает средний объём занимаемой памяти.

Проблемы в игре Go

Встроенный тип карт Go обладает некоторыми необычными свойствами, которые создают дополнительные сложности при адаптации нового дизайна карт. Два из них оказались особенно сложными.

Постепенный рост

Когда хеш-таблица достигает максимального коэффициента загрузки, ей необходимо увеличить размер резервного массива. Обычно это означает, что следующая вставка удваивает размер массива и копирует все записи в новый массив. Представьте себе вставку в карту с 1 ГБ записей. Большинство вставок выполняются очень быстро, но та, которая должна увеличить размер карты с 1 ГБ до 2 ГБ, потребует копирования 1 ГБ записей, что займёт много времени.

Go часто используется для серверов, чувствительных к задержкам, поэтому нам не нужны операции со встроенными типами, которые могут оказывать произвольно большое влияние на задержку в хвосте. Вместо этого карты Go растут инкрементально, так что каждая вставка имеет верхнюю границу объёма работы по росту, которую она должна выполнить. Это ограничивает влияние на задержку одной вставки карты.

К сожалению, конструкция швейцарской таблицы Abseil (C++) предполагает одновременный рост всех элементов, а последовательность зондов зависит от общего количества групп, что затрудняет ее разделение.

Встроенная в Go карта решает эту проблему, используя дополнительный уровень косвенности, разделяя каждую карту на несколько швейцарских таблиц. Вместо одной швейцарской таблицы, реализующей всю карту, каждая карта состоит из одной или нескольких независимых таблиц, охватывающих подмножество пространства ключей. Отдельная таблица хранит максимум 1024 записи. Для выбора таблицы, к которой принадлежит ключ, используется переменное количество старших битов в хэше. Это форма *расширяемого хеширования* , где количество используемых битов увеличивается по мере необходимости для дифференциации общего количества таблиц.

Если во время вставки требуется увеличить размер отдельной таблицы, это произойдет сразу, при этом другие таблицы не будут затронуты. Таким образом, верхняя граница для одной вставки равна задержке увеличения таблицы из 1024 записей в две таблицы по 1024 записи, копируя 1024 записи.

Модификация во время итерации

Многие конструкции хеш-таблиц, включая швейцарские таблицы Абсейла, запрещают изменение карты во время итерации. Спецификация языка Go явно разрешает изменения во время итерации со следующей семантикой:

- Если запись будет удалена до того, как она будет достигнута, она не будет создана.
- Если запись обновлена до того, как она будет достигнута, будет создано обновленное значение.
- Если добавляется новая запись, она может быть создана, а может и нет.

Типичный подход к итерации хеш-таблицы заключается в простом проходе по резервному массиву и получении значений в порядке их расположения в памяти. Этот подход противоречит описанной выше

семантике, в частности, потому что вставки могут привести к увеличению размера карты, что приведет к перетасовке структуры памяти.

Мы можем избежать влияния перемешивания во время роста, сохраняя в итераторе ссылку на таблицу, которую он в данный момент итерирует. Если эта таблица увеличивается во время итерации, мы продолжаем использовать старую версию таблицы и, таким образом, продолжаем предоставлять ключи в порядке, соответствующем старой структуре памяти.

Работает ли это с указанной выше семантикой? Новые записи, добавленные после роста, будут полностью пропущены, поскольку они добавляются только в расширенную таблицу, а не в старую. Это нормально, поскольку семантика позволяет не создавать новые записи. Однако обновления и удаления представляют собой проблему: использование старой таблицы может привести к появлению устаревших или удалённых записей.

Этот пограничный случай решается использованием только старой таблицы для определения порядка итераций. Перед тем, как фактически вернуть запись, мы обращаемся к расширенной таблице, чтобы определить, существует ли ещё запись, и получить последнее значение.

Это охватывает всю основную семантику, хотя есть ещё больше мелких пограничных случаев, которые здесь не рассматриваются. В конечном счёте, разрешённость карт Go с итерацией приводит к тому, что итерация становится самой сложной частью реализации карт Go.

Будущая работа

В микробенчмарках операции с картами выполняются на 60% быстрее, чем в Go 1.23. Точное увеличение производительности может существенно варьироваться из-за широкого спектра операций и способов использования карт, а в некоторых крайних случаях наблюдается регресс по сравнению с Go 1.23. В целом, в полнофункциональных тестах приложений мы обнаружили среднегеометрическое улучшение времени работы процессора примерно на 1,5%.

Мы хотим изучить и другие улучшения карт для будущих версий Go. Например, мы можем увеличить локальность операций с картами, не хранящимися в кэше процессора.

Мы также могли бы дополнительно улучшить сравнение управляющих слов. Как описано выше, у нас есть переносимая реализация, использующая стандартные арифметические и побитовые операции. Однако в некоторых архитектурах есть инструкции SIMD, которые выполняют подобные сравнения напрямую. В версии Go 1.24 уже используются 8-байтовые инструкции SIMD для amd64, но мы могли бы расширить поддержку и на другие архитектуры. Что ещё важнее, в то время как стандартные инструкции работают со словами длиной до 8 байт, инструкции SIMD почти всегда поддерживают как минимум 16-байтовые слова. Это означает, что мы могли бы увеличить размер группы до 16 слотов и выполнять 16 сравнений хэшей параллельно вместо 8. Это ещё больше уменьшило бы среднее количество запросов, необходимых для поиска.

Благодарности

Реализация карты для go на основе швейцарской таблицы (Swiss Table) разрабатывалась долго и включала множество участников. Я хочу поблагодарить Юньхао Чжана (@zhangyunhao116), П. Дж. Маллоя (@thepudds) и @andy-wm-arthur за создание первых версий реализации швейцарской таблицы для го. Питер Мэттис (@petermattis) объединил эти идеи с решениями перечисленных выше задач для го `github.com/cockroachdb/swiss`, чтобы создать реализацию швейцарской таблицы, соответствующую спецификации Go. Встроенная реализация карты для Go 1.24 во многом основана на работе Питера. Спасибо всем участникам сообщества, которые внесли свой вклад!

Следующая статья: От уникальности к очистке и слабости: новые низкоуровневые инструменты для повышения эффективности
Предыдущая статья: Тестирование параллельного кода с помощью testing/synctest
Индекс блога