GeekBrains

Урок 7. Кэширование. Redis

На этом уроке

1. Узнаем, что такое кэширование, зачем оно нужно и рассмотрим какие виды кэширований существуют.
2. Рассмотрим основные key-value хранилища/databases при кэшировании.
3. Рассмотрим примеры на go для работы с redis.

Оглавление

Дополнительные материалы

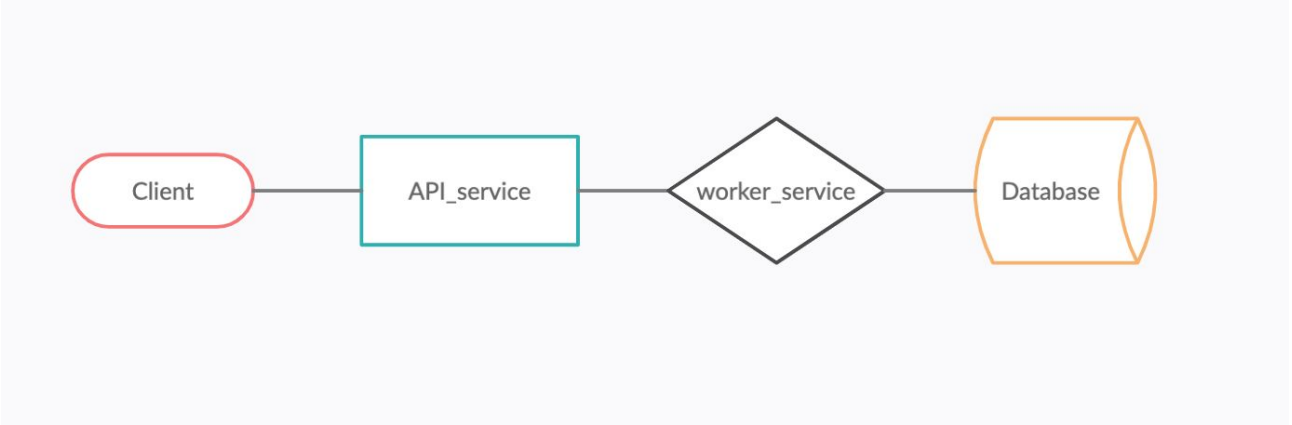
Используемые источники

Теория урока

Кэширование/Кэш

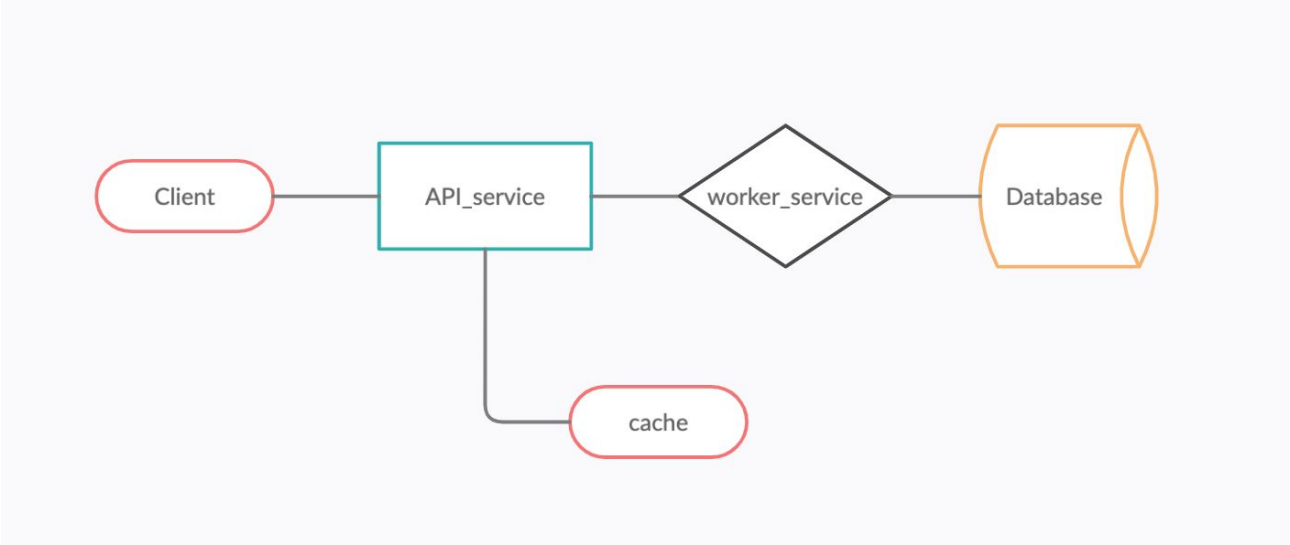
Кэш - промежуточное место хранения временных данных с быстрым доступом. ***Кэширование*** - процесс, при котором данные сохраняются в **Кэш**. Объем данных в кэше значительно меньше, чем в основном хранилище, но доступ осуществляется быстрее. В кэш обычно сохраняется тот набор набор данных, который будет запрошен в дальнейшем с наибольшей вероятностью.

Попробуем выяснить для чего необходимо кэширование. Виды кэширования мы рассмотрим ниже, но попробуем рассмотреть следующую ситуацию: у вас есть база данных, которая хранит определенную информацию, есть сервис-воркер (worker_service), который обращается за информацией в данную базу данных и есть апи-сервис (api_service), который предоставляет API для клиента и отправляет запрос воркеру:



Время на на весь запрос для клиента (от момента отправки до получения ответа от сервиса) состоит из трех частей. И если к нам клиент в определенный момент времени приходит к запросом, то мы идем по всей цепочки до базы данных, достаем оттуда информацию и возвращаемся по цепочке обратно. Представим теперь, что клиент спустя какое-то время отправляет аналогичный запрос, но за промежуток между старым и новым запросам у нас ничего не изменилось в базе данных. Нам придется опять же пройти по всей цепочке до базы данных, забрать оттуда туже самую информацию и вернуть по цепочке клиенту обратно. В итоге мы выполнили два одинаковых запроса с каким-то промежутком времени. Если нагрузка на сервис небольшая - кажется, что вполне терпимо, но если нагрузка на сервис увеличивается и каждый клиент запрашивает одинаковую информацию, то время ответа для клиента может увеличиваться за счет долгих походов в базу данных. В результате клиент приложения может быть недоволен долгими запросами в приложении. Здесь нас и могут спасти кэши.

Перед тем как пойти в базу данных, мы можем сходить во временное хранилище с быстрым доступом и проверить, есть ли там нужная нам информацию. Выглядеть это может примерно так:



В данном случае мы первым делом сходим в cache и если найдем там подходящую под наш запрос информацию, то сразу же отдаем клиенту эту информацию. Если же информацию в кэше не нашли - идем в базу данных как и в первоначальном варианте. Поход в кэш может сильно сократить время ожидания для клиента. Кэш, конечно же, нужно периодически обновлять после похода в базу данных. Варианты обновления кэшей рассмотрены ниже.

Кэш может быть пустым, может хранить релевантную информацию и может хранить не релевантную информацию. Пустой кэш и кэш с не релевантной информацией называют **холодным (cold),** так как необходимо дополнительно сходить в основное хранилище, чтобы получить нужные данные для запроса, поэтому запрос будет выполнен дольше. Кэш с релевантными данными называют **горячим (warm)**.

В примере выше мы рассмотрели только определенный тип кэширования, теперь пройдемся немного по каждому типу кэширования.

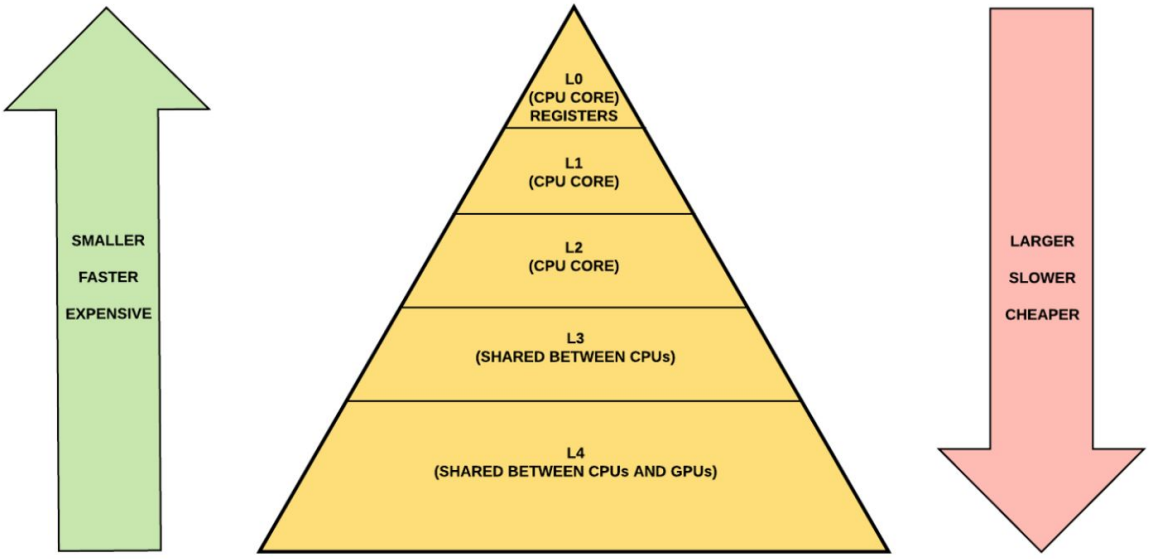
Виды кэшированиий

Кэширование может быть использовано на разных технологических уровнях - ОС, сети, база данных, на уровне приложения. Перейдем к некоторым конкретным видам. Будут затронуты не все виды и многие из них достаточно поверхностно, так как они не относятся напрямую к теме урока.

Процессорный кэш

Кэш-память процессора - это буфер между CPU и RAM. Данная память хранит данные и инструкции, которые используются чаще всего. За счет этого доступ к ним может происходить практически мгновенно. У кэша существует несколько уровней:

1. L0 - так называемые регистры, расположены ближе всего к CPU.
2. L1 - объем памяти в 10кб, интегрированный в CPU. При помощи кэша L1 мы можем скопировать содержимое адресов памяти, причем скорость доступа с высокой вероятностью будет близка к скорости регистров ЦП. Данные очень быстро загружаются в регистры ЦП
3. L2 - объем памяти до 200кб, расположен также как и L1 на одном кристалле с процессором.
4. L3 - объем памяти до нескольких мбайт.
5. Существует и L4, использование которого оправдано только в многопроцессорных высокопроизводительных серверов.



Кэш жесткого диска

Жесткие диски (HDD, Hard Disk Drive), предназначенные для постоянного хранения данных, являются медленными устройствами. В системах долговременного хранения информации кэш диска (или буфер диска) — это встроенная в жесткий диск память, которая играет роль буфера между CPU и физическим жестким диском.

Кэш базы данных

Обычно базы данных в дефолтных конфигурациях тоже имеют определенные уровни кэширования, который позволяют использовать базу данных эффективно. Помимо дефолтной конфигурации базы данных поддерживают набор настроек, который может задать пользователь из своих соображений.

Кэш на уровне браузера

В каждом браузере имеется реализация веб-кэша (или иначе - HTTP-кэша), который предназначен для временного хранения материалов, полученных из интернета, таких, как изображения, HTML-страницы или JavaScript-файлы. Данный вид кэша работает тогда, когда в ответе от сервера явно указывается когда браузер может кэшировать данные с сервера. В ответе передается статус код 304 и данные подгружаются с устройства пользователя. Это очень удобно, если сервис содержит много картинок. Но есть и нюанс: если на сервере картинка поменялась, то браузер не сразу об этом узнает.

Кэш уровня приложений

Кэш между хранилищем/базой данных и приложением. Наиболее популярным представления кэша являются key-value хранилища такие как redis и memcached. Данные кэши называются in-memory хранилищами. Данные хранятся в RAM, доступ к которому осуществляется быстрее, чем к базе данных, которая хранит данные на диске.

Кэш на уровне запросов к базе данных

На уровне приложения можно кэшировать запросы в базу данных. Но данных подход может со временем привести к сложностям:

- Трудно удалить кэш с результатом на сложный запрос
- Если что-то изменится в базе данных (e.g. какая-то таблица), то придется обновлять весь кэш

Кэш на уровне объектов

На уровне приложения можно также кэшировать объекты, которые позволяют отдавать результат пользователю без похода в базу данных.

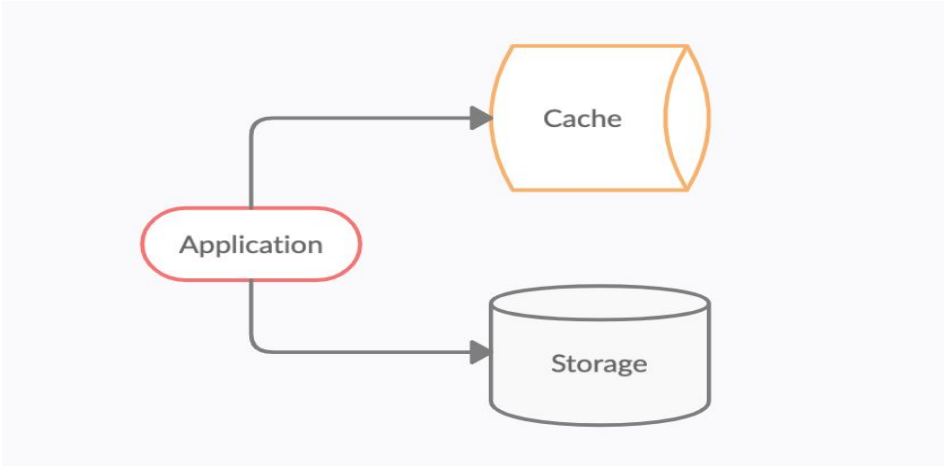
На данном уроке мы подробно рассмотрим только кэш на уровне приложения с использованием redis/memcached.

Способы обновления кэшей

Из-за того, что кэш хранит ограниченный объем данных, то нам нужно разобраться какие стратегии для обновления кэша существуют и рассмотреть их плюсы и минусы.

cache-aside

В данном подходе приложение (application) читает данные из хранилища (storage). Кэш (cache) не взаимодействует с хранилищем напрямую.



Приложение выполняет следующее:

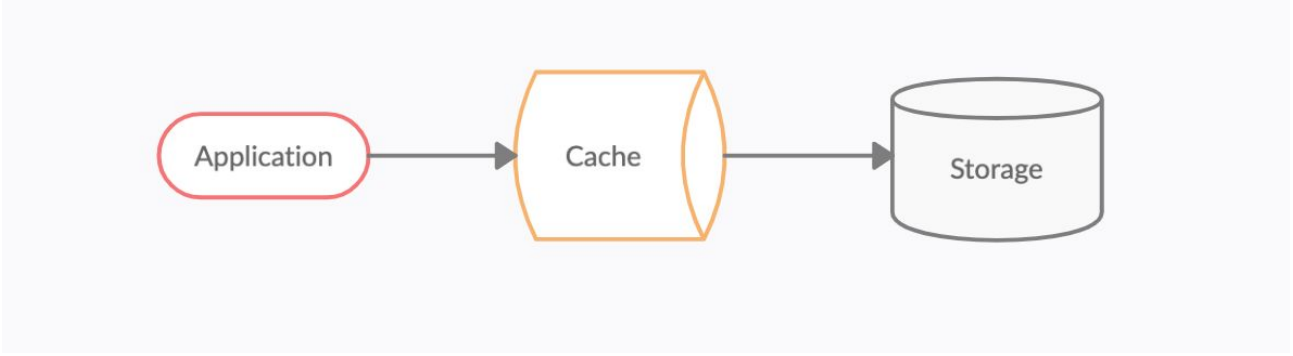
1. Ищет запись в кэше
2. Если запись есть - отдает клиенту информацию из кэша
3. Если записи нет - идет в хранилище
4. Обновляет кэш на основе данных из хранилища
5. Отдает результат клиенту

Недостатки:

- Данные в кэше могут устаревать при обновлении данных в storage. Может быть решено установкой TTL (time to live, время жизни) для данных или использованию другого подхода использования кэша.
- Если вдруг нода с кэшом упадет, то при замене ноды время ответа клиенту будет увеличено.

write-through

В данном подходе приложение использует кэш как основное хранилище данных, в то время как на кэш ставится задача взаимодействия с базой данных.



Выполняются следующие действия:

1. Приложение добавляет или обновляет данные кэша
2. Кэш пишет данные в базу данных

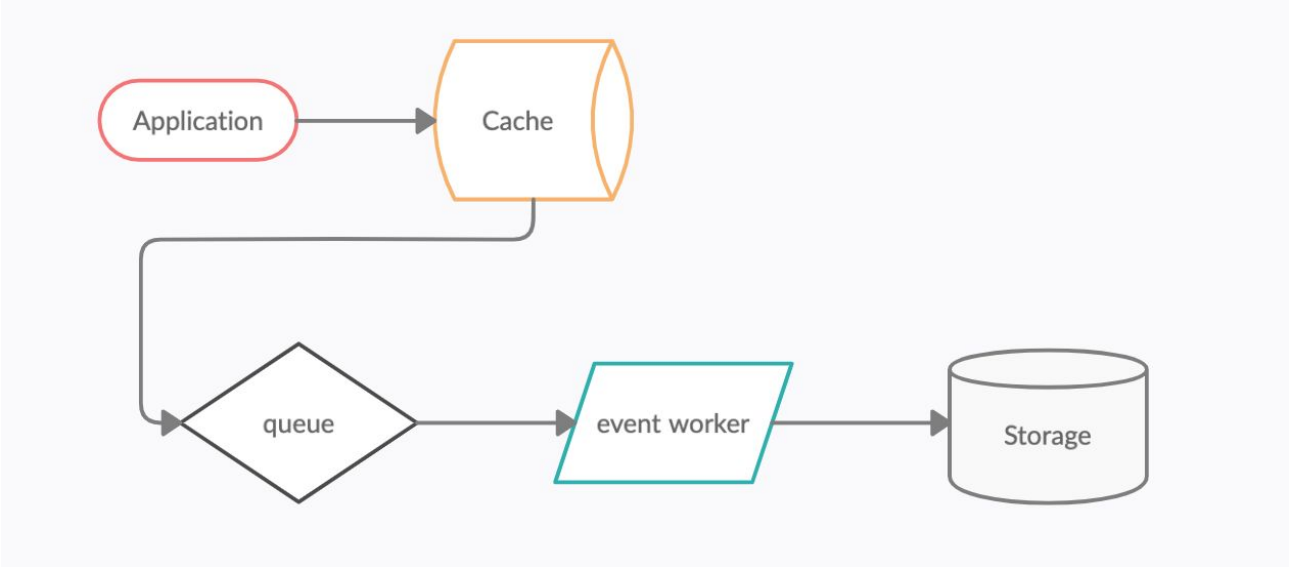
В данном подходе на операцию записи тратится много времени, а чтение данных может происходить очень быстро. Но в повседневной жизни пользователь проще/лояльнее относится к медленному обновлению или добавлению данных, чем к медленному чтению данных.

Недостатки:

- Многие записанные данные могут быть никогда и не прочитаны.
- Когда новый узел создается из-за сбоя или масштабирования, новый узел не будет кэшировать записи до тех пор, пока запись не будет обновлена в базе данных.

write-behind (write-back)

В данном подходе происходит асинхронная запись в хранилище с помощью очереди (queue) и воркера (event worker), который читает данные из очереди и пишет их в хранилище. Очереди будут рассмотрены на следующем уроке.



Выполняются следующие действия:

1. Приложение добавляет или обновляет данные кэша
2. Кэш добавляет в очередь, что необходимо сделать данные и сами данные
3. Воркер (worker) читает очередь и пишет в базу данных асинхронно

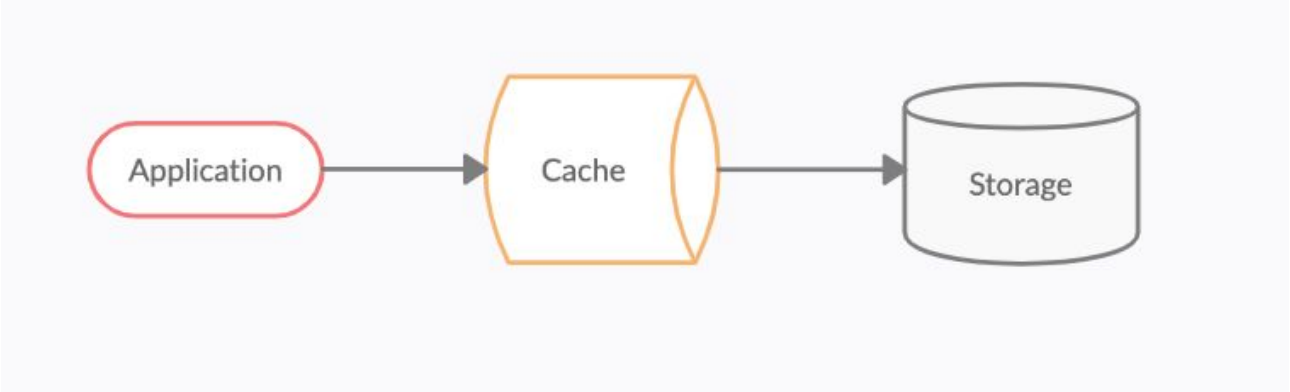
Недостатки:

- Если кэш выйдет из строя до того, как его содержимое запишется в хранилище, может произойти потеря данных

- Реализация сложнее, чем предыдущие

refresh-ahead

Можно настроить кэш так, чтобы он автоматически обновлял любую недавно доступную запись кэша до истечения TTL. Опережающее обновление может привести к снижению задержки по сравнению с чтением, если кэш может точно предсказать, какие элементы, вероятно, понадобятся в будущем.



Выполняемые действия:

1. Кэш периодически обновляет данные до истечения TTL
2. Все остальное происходит также как и в **write-through** подходе

Недостаток:

- Неточное предсказание того, какие элементы, вероятно, понадобятся в будущем, может привести к снижению производительности, чем без предварительного обновления.

Недостатки использования кэшей

1. Необходимо поддерживать согласованность данных между кэшем и основным хранилищем данных.
2. Валидация данных кэша не тривиальная задача. Также есть дополнительная сложность в том, по какой стратегии данные обновлять.
3. Необходимо внедрять в приложение дополнительные сущности, такие как redis/memcached.

Метрики для кэшей

При работе с кэшами часто строят метрики, чтобы проверять эффективно ли используются кэши. Есть метрика показывающая попадания (**hits**) в кэш и метрика подсчитывающая промахи (**misses**). На основе этих данных анализируется кэш.

Redis vs. Memcached

Существует две популярные key-value базы данных, которые могут быть использованы как кэш.

Redis

Redis (от англ. remote dictionary server) — опенсорсная, in-memory хранилище структур данных, используемая как база данных, кэш и брокер сообщений. Поддерживает такие структуры данных как strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs.

- Имеет много возможностей и применений: кэши, очереди, транзакции
- Позволяет хранить до 512Мб
- Поддерживает master-slave репликацию
- Производительность сравнима с memcached
- Может быть использовано как постоянное хранилище данных

Memcached

Memcached - это универсальная распределенная система кэширования. Часто используется для ускорения динамических веб-сайтов, управляемых базами данных, путем кэширования данных и объектов в оперативной памяти, чтобы уменьшить количество раз, когда внешний источник данных должен быть прочитан. Является бесплатной и опенсорсной.

- Быстрее чем Redis, но на практике это почти незаметно
- Отлично подходит для использования в качестве кэша, но в некоторых задач может проигрывать Redis

Практическая часть

Целью практической части является знакомство с таким инструментом кэширования как redis. Мы напишем несколько go-приложений, которые будут использовать взаимодействие с redis.

Подготовка нужны инструментов

Для использования redis поднимем локально с ним контейнер с помощью команды:

```
docker run -p 6379:6379  -d redis
```

Для работы с redis с помощью go нам понадобится следующая библиотека.

Базовое взаимодействие с redis

Рассмотрим несколько базовых функций библиотеки для работы с redis. Для начала научимся подключаться к redis из go-приложения. Создадим файл **simple_redis/client.go**:

```
package main

import (
  "context"
  "fmt"

  "github.com/go-redis/redis/v8"
)

type RedisClient struct {
  *redis.Client
}

func NewRedisClient(host, port string) (*RedisClient, error) {
  client := redis.NewClient(&redis.Options{
     Addr:     fmt.Sprintf("%s:%s", host, port),
     Password: "", // no password set
     DB:       0,  // use default DB
  })

  err := client.Ping(context.Background()).Err()
  if err != nil {
     return nil, fmt.Errorf("try to ping to redis: %w", err)
  }

  c := &RedisClient{
     Client: client,
  }

  return c, nil
}

func (c *RedisClient) Close() error {
  return c.Client.Close()
}

func (c *RedisClient) GetRecord(mkey string) ([]byte, error) {
  data, err := c.Get(context.Background(), mkey).Bytes()
  if err == redis.Nil {
     // we got empty result, it's not an error
     return nil, nil
  } else if err != nil {
     return nil, err
  }

  return data, nil
}
```

1. Мы создали структуру **RedisClient** и встроили в нее ***redis.Client**, с помощью которого и происходит взаимодействие с redis**.**
2. Создали функцию **NewRedisClient**, которая на вход получает **host**, **port**. Внутри функции мы создаем клиента для взаимодействия с redis, а с помощью команды **Ping** проверяем подключение с redis.
3. **GetRecord** на вход получает ключ для поиска в redis, а возвращает слайс байт и ошибку. С помощью команды Get мы по ключу обращаемся к redis. Базовый результат это структура, но с помощью функции пакета **Bytes** результат сразу возвращается в виде слайса байт. Это нам пригодится в дальнейшем.
4. В функции **GetRecord** мы обрабатываем ошибку, когда возвращается пустой результат. Здесь мы считаем пустой результат ошибкой и возвращаем **nil, nil.** Возможно, стоило вернуть наверх кастомную ошибку, что получили результат и на вызывающей стороне проверять пустой результат или нет, но на данный нам удобнее оставить в текущем варианте.

Теперь напишем в **simple_redis/main.go** функцию, в которой реализуем вызов базовых функций для работы с redis:

```
func basicWork(client *RedisClient) error {
  const (
     mKey         = "basic_key"
     notExistKey  = "basic_key" + "_not_exist"
     notExistKey2 = "sure_not_exist"
  )

  keys := []string{mKey, notExistKey, notExistKey2}

  // comment it if you want data from previous launch
  /**/
  err := client.Del(context.Background(), keys...).Err()
  if err != nil {
     return err
  }
  /**/

  item, err := client.GetRecord(mKey)
  if err != nil {
     return err
  }
  fmt.Printf("FIRST GetRecord for key %q `%s`\n", mKey, item)

  ttl := 5 * time.Second
  // добавляет запись, https://redis.io/commands/set
  err = client.Set(context.Background(), mKey, 1, ttl).Err()
  if err != nil {
     return err
  }
```

```
// just try to uncomment
// time.Sleep(ttl)

item, err = client.GetRecord(mKey)
if err != nil {
   return err
}
fmt.Printf("SECOND GetRecord for key %q `%s`\n", mKey, item)

// https://redis.io/commands/incrby
var count int64 = 2
err = client.IncrBy(context.Background(), mKey, count).Err()
if err != nil {
   return err
}
fmt.Printf("INCR for key %q on value %v\n", mKey, count)

item, err = client.GetRecord(mKey)
if err != nil {
   return err
}
fmt.Printf("THIRD GetRecord for key %q `%s`\n", mKey, item)

// https://redis.io/commands/decrby
err = client.Decr(context.Background(), mKey).Err()
if err != nil {
   return err
}
fmt.Printf("DECR for key %q\n", mKey)

item, err = client.GetRecord(mKey)
if err != nil {
   return err
}
fmt.Printf("FOURS GetRecord for key %q `%s`\n", mKey, item)

err = client.Incr(context.Background(), notExistKey).Err()
if err != nil {
   return err
}
fmt.Printf("INCR for key %q\n", notExistKey)

item, err = client.GetRecord(notExistKey)
if err != nil {
   return err
}
fmt.Printf("THIRD GetRecord for key %q `%s`\n", notExistKey, item)

// https://redis.io/commands/mget
result, err := client.MGet(context.Background(), keys...).Result()
if err != nil {
   return err
```

```
  }
  log.Printf("MGET for keys %v, result: %v", keys, result)

  return nil
}
```

1. В начале функции мы определяем список ключей для нашего примера, а с помощью функции библиотеки **Del** удаляем все нынешние значения значения для этих ключей в целях примера. Кусок кода с удалением можно закомментить, если удалять не хотите. **Del** команда возвращает уже другой объект, у которого мы вызываем функцию **Err** для проверки ошибки.
2. Функцию **GetRecord** мы определили ранее в другом файле, на ней останавливаться не будем.
3. Функция **Set** на вход принимает несколько параметров. Контекст в нашем случае нам не интересен (он может пригодится при более сложных запросах, на которые можно устанавливать таймауты и т.д.), затем идет пара ключ-значение, а последним элементом передается элемент ttl - time to live или время жизни данного значения. Мы выставляем значение для ttl 5 секунд. Значит после 5 секунд данное значение удалится.
4. **IncrBy** - увеличивает значение по ключу на заданное значение. Мы передаем в аргумент **count**, которое определили выше.
5. **Decr** - уменьшает значение по ключу на единицу.
6. **Incr** - уменьшает значение по ключу на единицу.
7. **MGet** - получение списка значения по списку ключей. Мы передаем в аргумент функции слайс ключей.

В **main.go** добавим еще функцию main:

```
func main() {
  const (
     host = "localhost"
     port = "6379"
  )

  client, err := NewRedisClient(host, port)
  if err != nil {
     log.Fatal(err)
  }
  defer client.Close()

  if err := basicWork(client); err != nil {
     log.Fatal(err)
  }
}
```

После выполнения команды:

```
 go run *.go
```

Получим **output**:

```
FIRST GetRecord for key "basic_key" ``
SECOND GetRecord for key "basic_key" `1`
INCR for key "basic_key" on value 2
THIRD GetRecord for key "basic_key" `3`
DECR for key "basic_key"
FOURS GetRecord for key "basic_key" `2`
INCR for key "basic_key_not_exist"
THIRD GetRecord for key "basic_key_not_exist" `1`
2020/12/06 21:09:16 MGET for keys [basic_key basic_key_not_exist sure_not_exist],
result: [2 1 <nil>]
```

Можно заметить, что если до вызова функции **Incr** не было значения с ключом в redis, то он создастся там автоматически. А в случае функции **MGET** мы получили **nil** для ключа, которого не нашли в redis.

В примере выше мы хранили в redis простые элементы: числа. Попробуем хранить там структуру. Для этого напишем следующую функцию:

```
func withStructWork(client *RedisClient) error {
  jsonKey := "json_key"
  // comment it if you want data from previous launch
  /**/
  err := client.Del(context.Background(), []string{jsonKey}...).Err()
  if err != nil {
     return err
  }
  /**/

  type exampleStruct struct {
     FieldOne string `json:"field_one"`
     FieldTwo string `json:"field_two"`
  }

  s := exampleStruct{
     FieldOne: "one",
     FieldTwo: "two",
  }

  data, err := json.Marshal(s)
  if err != nil {
     return err
  }

  ttl := 5 * time.Second
  err = client.Set(context.Background(), jsonKey, data, ttl).Err()
  if err != nil {
     return err
  }
```

```
  item, err := client.GetRecord(jsonKey)
  if err != nil {
     return err
  }
  fmt.Printf("GetRecord for key %q `%s`\n", jsonKey, item)

  e := new(exampleStruct)
  if err := json.Unmarshal(item, e); err != nil {
     return err
  }
  fmt.Printf("example struct: %+v\n", e)

  return nil
}
```

Теперь в **main** функции вместо **basicWork** вызовем эту **withStructWork.** В **output** должны получить:

```
GetRecord for key "json_key" `{"field_one":"one","field_two":"two"}`
example struct: &{FieldOne:one FieldTwo:two}
```

Чтобы хранить структуру мы сериализовали ее и redis сохранил ее как строку.

Может возникнуть вопрос: что произойдет, если выполнить функцию для числовых значений на строковое значение в redis? Например **Incr**.
Ответ: возникнет ошибка **2020/12/06 21:17:04 ERR value is not an integer or out of range**

Сервис с сессиями пользователя

Рассмотрим теперь чуть более сложный пример - простенькую авторизацию с сохранением сессий пользователя в redis. Суть заключается в том, чтобы генерировать key-value пару, хранить ее в redis, на основе key генерировать клиенту cookie, чтобы была возможность оставаться авторизованным на протяжении какого-то времени.

Первым делом возьмем весь код из файла **simple_redis/client.go** и скопируем его в **sessions/client.go.** Но структуру **RedisClient** нужно дополнить одним полем:

```
type RedisClient struct {
  *redis.Client
  TTL time.Duration
}
```

TTL будет определять время жизни куки и время жизни key-value пары в redis.

Теперь напишем **sessions/sessions.go** файл с использованием библиотеки uuid:

```go
type Session struct {
  Login     string
  Useragent string
}

type SessionID struct {
  ID string
}

func (c *RedisClient) Create(in Session) (*SessionID, error) {
  data, err := json.Marshal(in)
  if err != nil {
     return nil, fmt.Errorf("marshal session ID: %w", err)
  }

  id := SessionID{
     ID: uuid.NewV4().String(),
  }

  mkey := newRedisKey(id.ID)
  err = c.Set(context.Background(), mkey, data, c.TTL).Err()
  if err != nil {
     return nil, fmt.Errorf("redis: set key %q: %w", mkey, err)
  }

  return &id, nil
}

func (c *RedisClient) Check(in SessionID) (*Session, error) {
  mkey := newRedisKey(in.ID)
  data, err := c.GetRecord(mkey)
  if err != nil {
     return nil, fmt.Errorf("redis: get record by key %q: %w", mkey, err)
  }else if data == nil {
      // add here custom err handling
      return nil, nil
  }


  sess := new(Session)
  err = json.Unmarshal(data, sess)
  if err != nil {
     return nil, fmt.Errorf("unmarshal to session info: %w", err)
  }

  return sess, nil
}

func (c *RedisClient) Delete(in SessionID) error {
  mkey := newRedisKey(in.ID)

  err := c.Del(context.Background(), mkey).Err()
  if err != nil {
```

```
    return fmt.Errorf("redis: trying to delete value by key %q: %w", mkey,err)
  }

  return nil
}

func newRedisKey(sessionID string) string {
  return fmt.Sprintf("sessions: %s", sessionID)
}
```

Здесь определены три функции: **Create** - для создания объекта **SessionID**, **Check** - для создания структуры **Session** по **SessionID**, **Delete** - удаление по **SessionID** значения из redis. Как можно заметить - код достаточно простой, большую часть представленных операций вы узнали на ранних курсах, а операции для работы с redis были рассмотрены в предыдущем примере.

Теперь определим файл **sessions/handlers.go**:

```
var loginFormTmpl = []byte(`
  <html>
     <body>
     <form action="/login" method="post">
        Login: <input type="text" name="login">
        Password: <input type="password" name="password">
        <input type="submit" value="Login">
     </form>
     </body>
  </html>
  `)

const (
  loginValue    = "login"
  passwordValue = "password"
)

var welcome = "Welcome, %s <br />\nSession User-Agent: %s <br />\n<a
href=\"/logout\">logout</a>"

func (c RedisClient) RootHandler(w http.ResponseWriter, r *http.Request) {
  sess, err := c.checkSession(r)
  if err != nil {
     err = fmt.Errorf("check session: %w", err)
     log.Printf("[ERR] %v", err)
     w.WriteHeader(http.StatusInternalServerError)
     return
  }

  if sess == nil {
     _, _ = w.Write(loginFormTmpl)
     return
  }
```

```
  w.Header().Set("Content-Type", "text/html")
  _, _ = fmt.Fprintln(w, fmt.Sprintf(welcome, sess.Login, sess.Useragent))
}

var users = map[string]string{
  "geek": "brains",
}

const cookieName = "session_id"

func (c RedisClient) LoginHandler(w http.ResponseWriter, r *http.Request) {
  inputLogin := r.FormValue(loginValue)
  inputPass := r.FormValue(passwordValue)

  // common map!!! dont make the same in production
  pass, exist := users[inputLogin]
  if !exist || pass != inputPass {
     w.WriteHeader(http.StatusUnauthorized)
     return
  }

  sess, err := c.Create(Session{
     Login:     inputLogin,
     Useragent: r.UserAgent(),
  })

  if err != nil {
     err = fmt.Errorf("create session: %w", err)
     log.Printf("[ERR] %v", err)
     w.WriteHeader(http.StatusInternalServerError)
     return
  }

  cookie := http.Cookie{
     Name:    cookieName,
     Value:   sess.ID,
     Expires: time.Now().Add(c.TTL),
  }

  http.SetCookie(w, &cookie)
  http.Redirect(w, r, "/", http.StatusFound)
}

func (c RedisClient) LogoutHandler(w http.ResponseWriter, r *http.Request) {
  session, err := r.Cookie(cookieName)
  if err == http.ErrNoCookie {
     http.Redirect(w, r, "/", http.StatusFound)
     return
  } else if err != nil {
     err = fmt.Errorf("read cookie %q: %w", cookieName, err)
     log.Printf("[ERR] %v", err)
     w.WriteHeader(http.StatusInternalServerError)
```

```
        return
    }

    err = c.Delete(SessionID{ID: session.Value})
    if err != nil {
        err = fmt.Errorf("delete session value %q: %w", session.Value, err)
        log.Printf("[ERR] %v", err)
        w.WriteHeader(http.StatusInternalServerError)
        return
    }

    session.Expires = time.Now().AddDate(0, 0, -1)

    http.SetCookie(w, session)
    http.Redirect(w, r, "/", http.StatusFound)
}

func (c RedisClient) checkSession(r *http.Request) (*Session, error) {
    cookieSessionID, err := r.Cookie(cookieName)
    if err == http.ErrNoCookie {
        return nil, nil
    } else if err != nil {
        return nil, err
    }

    sess, err := c.Check(SessionID{ID: cookieSessionID.Value})
    if err != nil {
        return nil, fmt.Errorf("check session value %q: %w", cookieSessionID.Value,
err)
    }

    return sess, nil
}
```

Данный файл тоже достаточно простой. В нем определены 3 хэндлера и вспомогательные объекты: **RootHandler** - хэндлер с проверкой сессии по куке, **LoginHandler** - хэндлер, который проверяет введенный логин и пароль пользователя в объекте **users, LogoutHandler** - хэндлер для удаления сесси пользователя: очищается кука пользователя и значение из redis.

Пример достаточно учебный и поэтому для простоты и экономии времени здесь используется захардкоженная мапа с пользователями и без необходимых проверок.

И заключительный файл **sessions/main.go:**

```
func main() {
    const (
        host        = "localhost"
        redisPort   = "6379"
        servicePort = "8080"
```

```
  )

  ttl := 1 * time.Hour
  client, err := NewRedisClient(host, redisPort, ttl)
  if err != nil {
     log.Fatal(err)
  }
  defer client.Close()

  http.HandleFunc("/", client.RootHandler)
  http.HandleFunc("/login", client.LoginHandler)
  http.HandleFunc("/logout", client.LogoutHandler)

  log.Printf("starting server at :%s", servicePort)
  log.Fatal(http.ListenAndServe(":"+servicePort, nil))
}
```

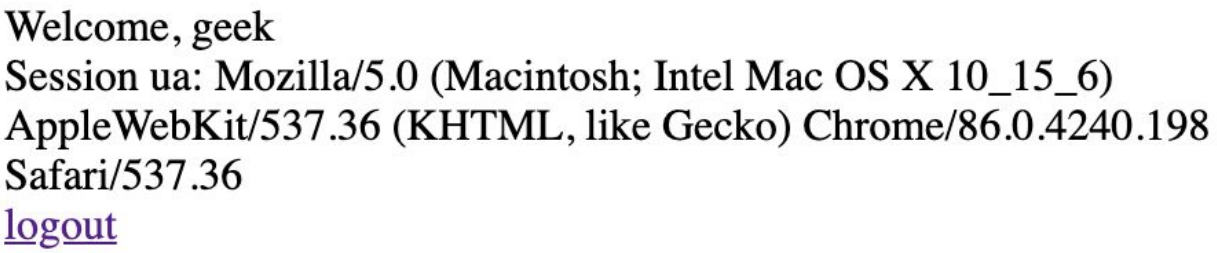
Здесь также ничего нового - все уже проходили.

После запуска в **sessions:**

```
go run *.go
```

И при переходе на **localhost:8080** можно увидеть:

А при вводе **geek** в **Login**, а **brains** в **Password** получим:

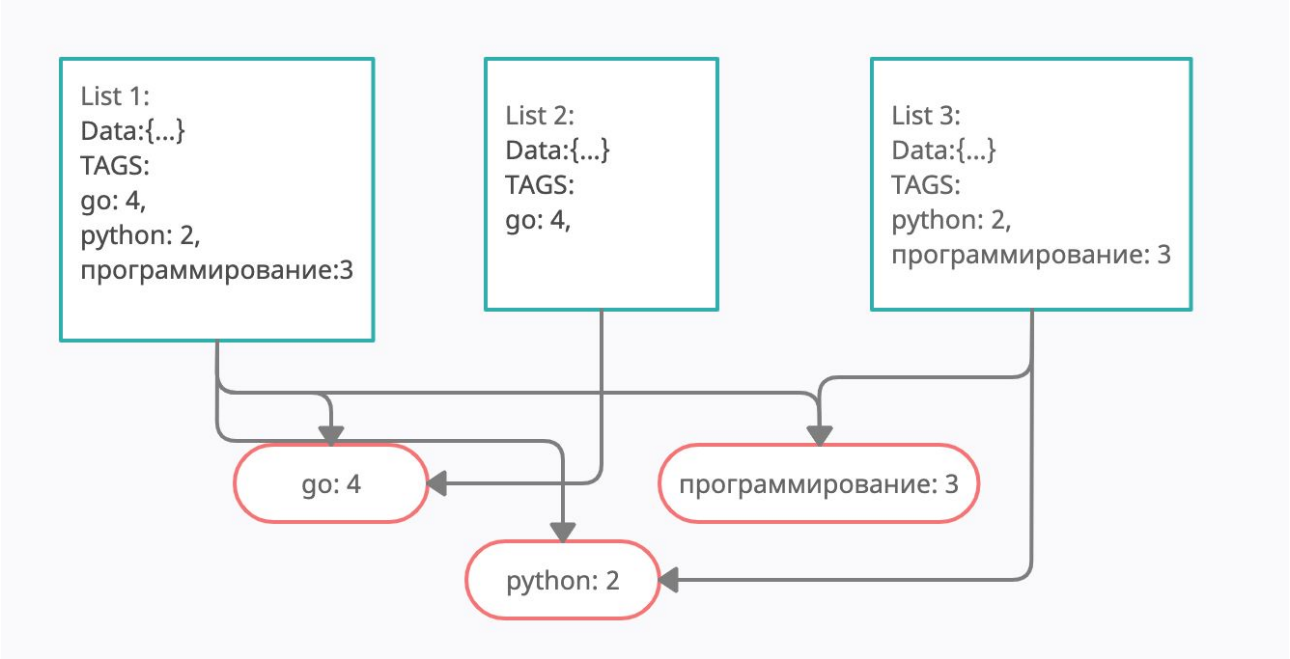


Чтобы посмотреть **cookie**, можно выполнить следующее: **Посмотреть код страницы>Application**.

Приложения для обновления кэша

В заключении рассмотрим следующую общую систему. Допустим, у нас есть ресурс (база данных, API ручка другого сервиса и т.д.), с которого мы получаем нужную нам информацию. У нас есть кэш, который хранит эту информацию. У нас есть сервис-worker, который обновляет кэш и есть сервис-апи, который читает значения из кэша, а затем отдает результат клиенту, который к нему обратился. Попробуем реализовать часть этой системы в упрощенном формате, а именно - обновление кэша. В качестве ресурса мы возьмем habr. Мы хотим реализовать приложение так, чтобы всю базовую инфу о постах с различных хабов (go-hub, python-hub, programming-hub и т.д.) бралась из кэша. Перед нами встает проблема, как нам доверять кэшу, или как мы можем убедиться что он валиден. Например, у нас есть два хаба: go и программирование. Если мы добавим в hub по go пост с тэгами Go и программирование, то нам нужно обновить кэш так, чтобы в хабе по программированию мы смогли показать этот же пост, но используя кэш.

Посмотрим на следующую картинку:



Каждый List содержит информацию по каждому хабу в Data (какую-то базовую инфу по типу URL, Title, Tags) и информацию в TAGS с общим счетчиком всех постов на эту тему между этими листами.

Идея заключается в том, чтобы при походе в кэш проверять для каждого List TAGS, сравнивать значения в кэше для каждого тэга и, если, значение не совпадает - перестраивать общий кэш. Все объекты на картинке хранятся в кэше, но под различными ключами. Например List 1 хранится под ключом **list_programming**, счетчики по постам под ключами **go**, **python**, **programming** и т.д.

Мы будем реализовывать обновление кэша только по одному листу, например по go с помощью представления RSS.

Сразу напишем необходимый код на go в файле **rebuild_cache/posts.go**:

```
type RSS struct {
  Items []Item `xml:"channel>item"`
}

type Item struct {
  URL      string   `xml:"guid"`
  Title    string   `xml:"title"`
  Category []string `xml:"category"`
}

func FetchContent(url string) (*RSS, error) {
  fmt.Printf("fetching: %s\n", url)
  resp, err := http.Get(url)
  if err != nil {
     return nil, fmt.Errorf("HTTP GET for URL %s: %w", url, err)
  }

  defer resp.Body.Close()
  body, err := ioutil.ReadAll(resp.Body)
  if err != nil {
     return nil, fmt.Errorf("read response body: %w", err)
  }

  rss := new(RSS)
  err = xml.Unmarshal(body, rss)
  if err != nil {
     return nil, fmt.Errorf("unmarshal body: %w", err)
  }

  return rss, nil
}
```

Здесь все очень тривиально - функция, которая отправляет GET запрос по URL и анмарашлить ответ в готовую структуру. В кэше мы храним тэги (категории, к которому относится пост) и url с названием.

Теперь возьмем с предыдущего примера кусок кода для создания клиента для redis и запишем его в **rebuild_cache/redis_client.go**:

```
type RedisClient struct {
  Client *redis.Client
  TTL    time.Duration
}

func NewRedisClient(host, port string, ttl time.Duration) (*RedisClient, error)
{
  client := redis.NewClient(&redis.Options{
     Addr:     fmt.Sprintf("%s:%s", host, port),
     Password: "", // no password set
```

```
        DB:       0,  // use default DB
    })

    err := client.Ping(context.Background()).Err()
    if err != nil {
        return nil, fmt.Errorf("try to ping to redis: %w", err)
    }

    c := &RedisClient{
        Client: client,
        TTL:    ttl,
    }

    return c, nil
}

func (rc *RedisClient) Close() error {
    return rc.Client.Close()
}
```

Данный код тоже нам знаком - рассматривали ранее на вебинаре. Переходим к более сложной части - работа с кэшом.

С этого момента мы будем писать в файлик **rebuild_cache/cache.go**:

```
func (rc *RedisClient) getCurrentTags(tags []string) (map[string]int, error) {
    currTags, err := rc.Client.MGet(context.Background(), tags...).Result()
    if err != nil {
        return nil, fmt.Errorf("MGET redis for tags %v: %w", tags, err)
    }

    resultTags := make(map[string]int, len(tags))
    now := int(time.Now().Unix())

    for i, tagKey := range tags {
        tagItem := currTags[i]
        if tagItem == nil {
            err := rc.Client.Set(context.Background(), tagKey, now, rc.TTL).Err()
            if err != nil {
                return nil, fmt.Errorf("set to redis key-value: %v-%v", tagKey, now)
            }

            resultTags[tagKey] = now
            continue
        }

        data, ok := tagItem.(string)
        if !ok {
            log.Printf("current tags assertion err for %v with type %T", tagItem,
tagItem)
```

```
            continue
        }

        number, err := strconv.Atoi(data)
        if err != nil {
            return nil, err
        }

        resultTags[tagKey] = number
    }

    return resultTags, nil
}
```

Здесь мы реализовали функцию **getCurrentTags.** На вход мы принимаем список тэгов и вызываем функцию для работы с redis **MGET.** Данная функция нам вернет слайс такого же размера, но если по ключу не было найдено значение - элемент будет равен **nil.** Это мы можем использовать дальше в работе этой функции. В качестве ключей для полученных тэгов мы используем **time.Now()**. Если мы получили на вход тэг, которого раньше не было, то мы его устанавливаем его с ключем **now**. В противном случае, мы просто собираем результат для функции.

Теперь рассмотрим функцию **rebuild**:

```
func (rc *RedisClient) rebuild(mkey string, in interface{}, rebuildCb
RebuildFunc) error {
    result, tags, err := rebuildCb()
    if err != nil {
        return fmt.Errorf("rebuild cb: %w", err)
    }

    if reflect.TypeOf(result) != reflect.TypeOf(in) {
        return fmt.Errorf("data type mismatch, expected %s, got %s",
reflect.TypeOf(in), reflect.TypeOf(result))
    }

    currTags, err := rc.getCurrentTags(tags)
    if err != nil {
        return fmt.Errorf("get current item tags: %w", err)
    }

    cacheData := CacheItemStore{
        Data: result,
        Tags: currTags,
    }

    rawData, err := json.Marshal(cacheData)
    if err != nil {
        return fmt.Errorf("marshal cache item store: %w", err)
    }
```

```
  err = rc.Client.Set(context.Background(), mkey, rawData, rc.TTL).Err()
  if err != nil {
     return fmt.Errorf("set raw data: %w", err)
  }

  inVal := reflect.ValueOf(in)
  resultVal := reflect.ValueOf(result)
  rv := reflect.Indirect(inVal)
  rvpresult := reflect.Indirect(resultVal)
  rv.Set(rvpresult

  return nil
}
```

Данная функция поинтереснее: на вход она принимает значение **mkey** - это как раз ключ для redis, в котором хранятся посты, например, для хаба **programming**, **in** - это элемент который хочет получить пользователь. Мы проверяем с помощью рефлектов, что callback функция возвращает элемент такого же типа что и **in**. В самом конфе **in** изменяется с помощью рефлектов свежими значениями. **rebuildCb** - это как раз функция, которая ходит в “хранилище” (в нашем случае на habr) и получает свежую информацию. Затем мы создаем объект, который храним по ключу **mkey**.

Следущая функция проверяет валидность данных с помощью тэгов.

```
func (rc *RedisClient) validateTags(itemTags map[string]int) (bool, error) {
  tags := make([]string, 0, len(itemTags))
  for tagKey := range itemTags {
     tags = append(tags, tagKey)
  }

  curr, err := rc.Client.MGet(context.Background(), tags...).Result()
  if err != nil {
     return false, fmt.Errorf("MGET redis for tags %v: %w", tags, err)
  }

  currentTagsMap := make(map[string]int, len(curr))
  for i, tagItem := range curr {
     data, ok := tagItem.(string)
     if !ok {
        log.Printf("validate tags: type assertion err for value %v with type
%T", tagItem, tagItem)
        continue
     }

     number, err := strconv.Atoi(data)
     if err != nil {
        return false, err
     }
     currentTagsMap[tags[i]] = number
```

```
  }

  return reflect.DeepEqual(itemTags, currentTagsMap), nil
}
```

И в заключении - определяем нужные структуры и функцию, которая работает с кэшом. Суть в том, чтобы обновить входящий параметр in, есть информацию в кэша валидна или обновить с п мощью свежего похода в ресурс.

```
type CacheItem struct {
  Data json.RawMessage
  Tags map[string]int
}

type CacheItemStore struct {
  Data interface{}
  Tags map[string]int
}

type RebuildFunc func() (interface{}, []string, error)

func (rc *RedisClient) GetCache(mkey string, in interface{}, rebuildCb
RebuildFunc) (err error) {
  inKind := reflect.ValueOf(in).Kind()
  if inKind != reflect.Ptr {
     return fmt.Errorf("expect pointer, got %s", inKind)
  }

  itemRaw, err := rc.Client.Get(context.Background(), mkey).Bytes()
  if err == redis.Nil {
     fmt.Println("record not found in cache")
     return rc.rebuild(mkey, in, rebuildCb)
  } else if err != nil {
     return fmt.Errorf("redis: get info for key %v: %w", mkey, err)
  }

  item := new(CacheItem)
  err = json.Unmarshal(itemRaw, item)
  if err != nil {
     return fmt.Errorf("unmarshal to cache item: %w", err)
  }

  tagsValid, err := rc.validateTags(item.Tags)
  if err != nil {
     return fmt.Errorf("validate item tags error %w", err)
  }

  if tagsValid {
     return json.Unmarshal(item.Data, &in)
  }
```

```
  return rc.rebuild(mkey, in, rebuildCb)
}
```

И теперь напишем файлик **rebuild_cache/main.go**:

```
func main() {
  const (
     host = "localhost"
     port = "6379"

     url = "https://habr.com/ru/rss/hub/go"
     ttl = 30 * time.Second
  )

  client, err := NewRedisClient(host, port, ttl)
  if err != nil {
     log.Fatal(err)
  }
  defer client.Close()

  const (
     mkey = "rebuild_cache_key"

     customTagOne      = "python"
     customTagTwo = "go"
  )
  tags := []string{customTagHabr, customTagGeekBrains}

  /**/
  // comment it if you dont want delete tags before work
  for _, v := range append(tags, mkey) {
     client.Client.Del(context.Background(), v)
  }
  /**/

  rebuild := func() (interface{}, []string, error) {
     posts, err := FetchContent(url)
     if err != nil {
        return nil, nil, err
     }

     // for lesson example we use here hardcode tags
     return posts, tags, nil
  }

  fmt.Println("FIRST call")
  posts := RSS{}
  err = client.GetCache(mkey, &posts, rebuild)
  log.Printf("FIRST result: posts: %v, error: %v\n\n", len(posts.Items), err)
```

```
  fmt.Println("SECOND call")
  posts = RSS{}
  err = client.GetCache(mkey, &posts, rebuild)
  log.Printf("SECOND result: posts: %v, error: %v\n\n", len(posts.Items), err)

  fmt.Printf("increment tag: %v\n", customTagOne)
  client.Client.Incr(context.Background(), customTagHabr)

  fmt.Println("THIRD call")
  posts = RSS{}
  err = client.GetCache(mkey, &posts, rebuild)
  log.Printf("THIRD result: posts: %v, error: %v\n\n", len(posts.Items), err)
}
```

Здесь мы проверяем работоспособность нашей функции. Создание RedisClient затрагивалось в прошлый пример, удаление данных нужно, чтобы пример работал каждый раз с чистым кэшом. **rebuild** функция просто вызывает функцию для похода на ресурс (в нашем случае хабр) и возвращает результат. В качестве тестирования мы здесь хардкодим тэги, чтобы проверить работоспособонсть. В первый вызов функции мы ожидаем, что кэш пуст и мы сходим в ресурс за обновлением кэша, второй вызов должен нам просто сходить в кэш, а перед тертьим вызовом мы искусственно изменяет в редисе значение по ключу одного из тэгов, чтобы посмотреть будет ли обновляться кэш или нет.

После запуска должно быть следующее:

```
# go run *.go
FIRST call
record not found in cache
fetching: https://habr.com/ru/rss/hub/go/
2020/12/08 19:46:33 FIRST result: posts: 20, error: <nil>

SECOND call
2020/12/08 19:46:33 SECOND result: posts: 20, error: <nil>

increment tag: python
THIRD call
fetching: https://habr.com/ru/rss/hub/go/
2020/12/08 19:46:33 THIRD result: posts: 20, error: <nil>
```

Программа ведет себя таким образом, как мы описали выше. По поводу реализации есть важное замечание: если несколько воркеров полезут обновлять кэш - могут случиться с проблемы с нагрузкой. Здесь нужно добавить блокировку на использование кэша, чтобы все вдруг не стали ломиться обновлять кэш.

Практическое задание

1. Реализовать сервис, который подтверждает аккаунт пользователя по смс-коду/email. То есть пользователь регистрируется, ему приходит уведомление на телефон/почту с кодом, он переходит по ссылке и вводит полученный код. Не обязательно реализовывать отправку sms/email - можно просто выводить в консоль “отправленный код”. Фронтенд тоже не обязательно. Важно использование redis.
2. *(доп). Реализовать систему worker-service, api-service, cache. Где worker читает базу данных (любую на ваш вкус) и обновляет кэш при необходимости, а api-service получает запросы от клиента на получение данных из базы, но запросы должны идти в cache.
3. *(доп) Взять последний пример с вебинара (про обновление кэша) и добавить блокировку на кэш, который уже перестраивается.

Дополнительные материалы

1. Статья Scalable System Design Patterns
2. Статья Introduction to architecting systems for scale
3. Материал про Amazon ElastiCache
4. Статья с habr “Как работает реляционная БД”
5. Статья Golang’s Superior Cache Solution to Memcached and Redis
6. Библиотека для работы с memcached
7. Статья Redis vs. Memcached: In-Memory Data Storage Systems

Используемые источники

1. Обзор кэширования на AWS.
2. Кэши с википедии.
3. Кэширование и производительность веб-приложений с habr.
4. 11 видов кэширования для современного сайта с habr.
5. Документация к redis.
6. Документация к memcached.
7. Раздел про кэширования с репозитория github про построения сервисов.
8. Статья с habr Использование memcached и Redis в высоконагруженных проектах