

Golang으로 로그 적재 시스템 개발기

golang-korea Meetup - Nov.2018

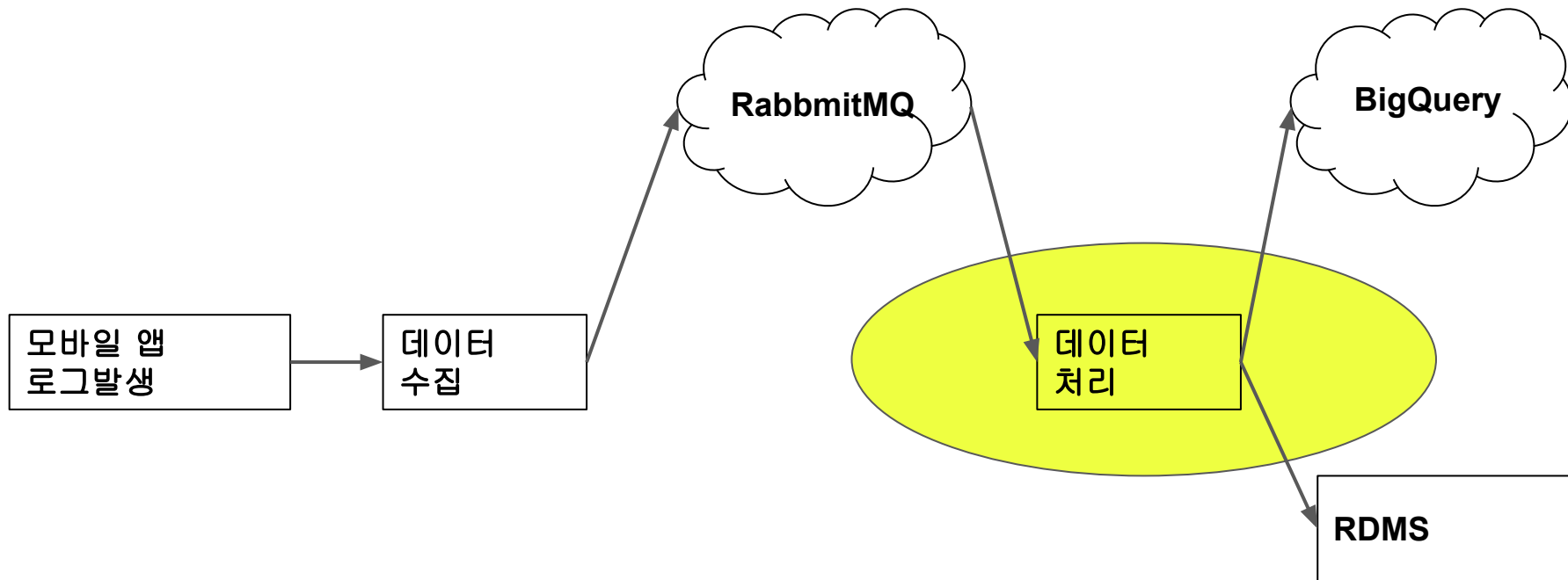
온수영

로그 적재 시스템 개요



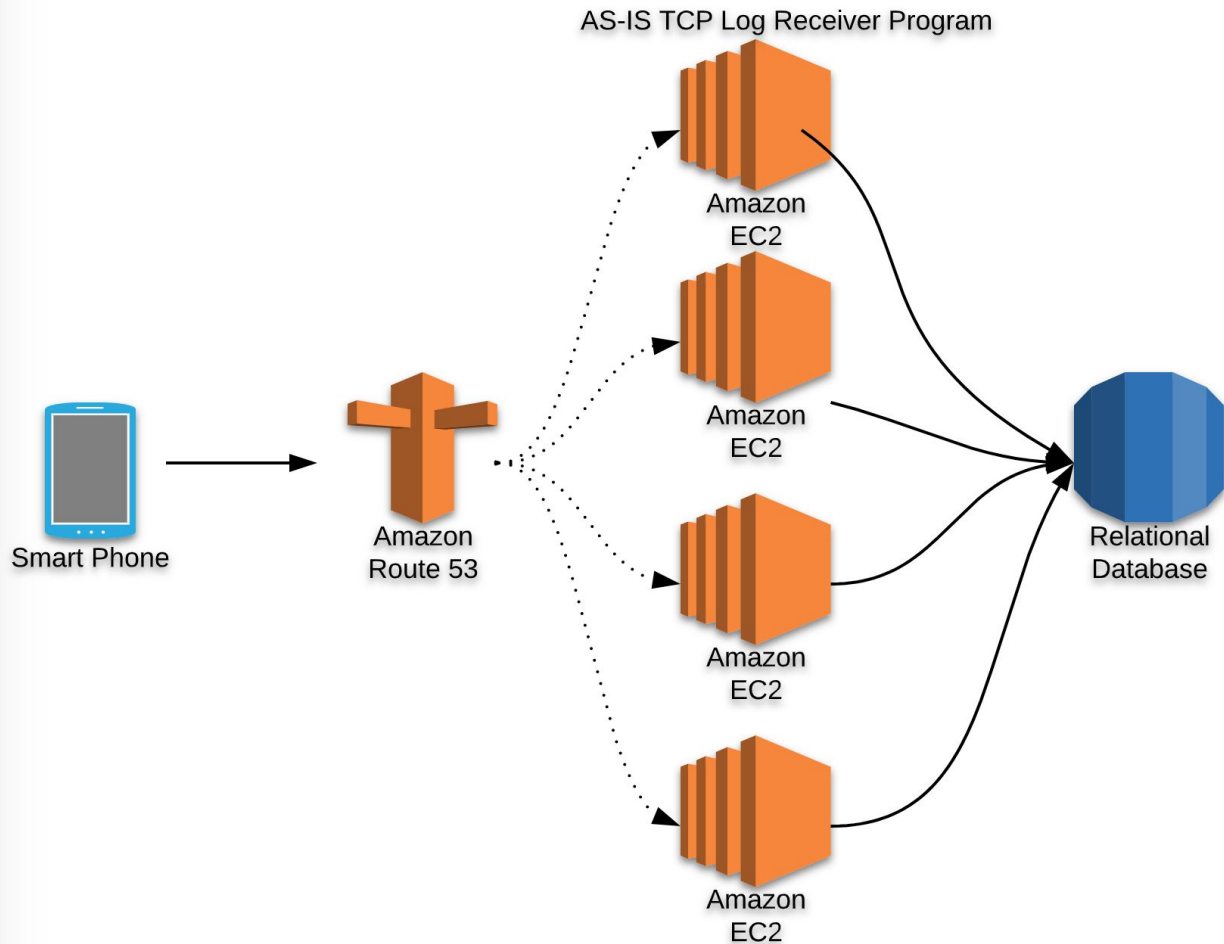
모바일 단말상의 로그를
수집 서버에서 받아
RabbitMQ를 브로커로
활용하여 RDMS와
BigQuery에 적재하는 시스템

로그 적재 시스템 데이터 흐름



기존 시스템 구조

- Netty 기반 개발
- 로그를 수신도 받고 가공 처리(DB DML 작업)까지 수행.
- 사용자 증가에 비례하여 **Scale-Out** 하는 구조로 운영 비용증가.
- TPS가 1대당 60을 넘지 못함.

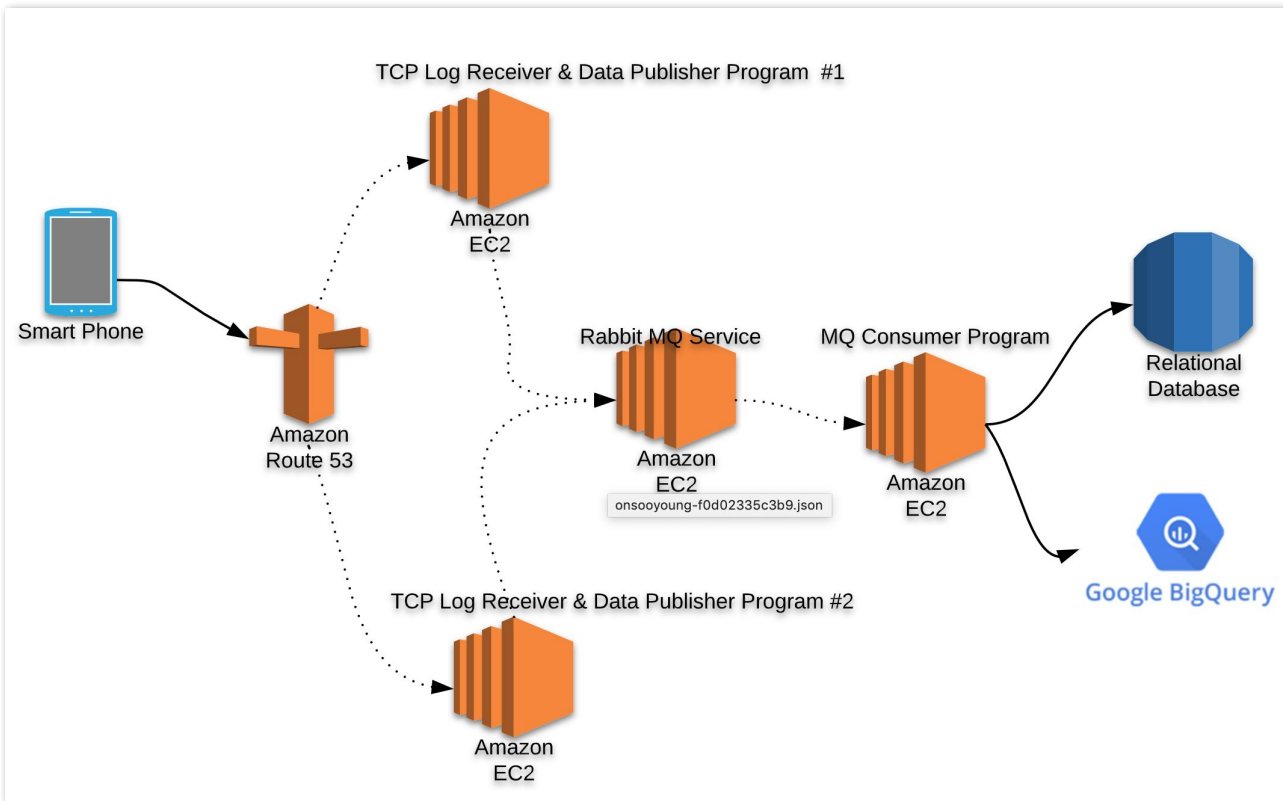


Golang을 활용한 계기

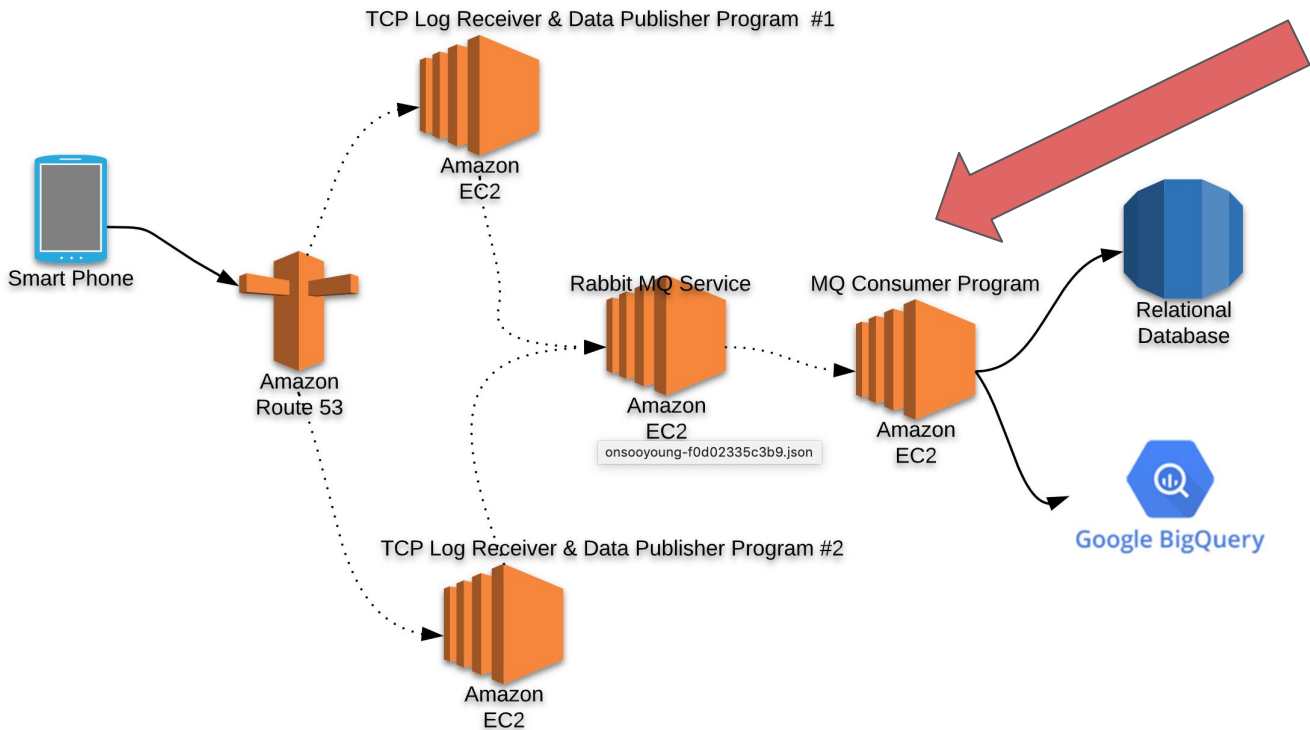
- 수집은 **Netty**기반의 수집서버에서 처리
- 데이터 가공 처리하여 **RDMS**에 데이터를 넣을 프로그램이 필요하다.
- 반드시 자바일 필요가 있을까?
- 고속 처리가 가능해야한다.
- 시간이 없으니, 개발 생산성이 높아야 한다.
- **RabbitMQ**와 **DB** 핸드링에 용이해야 한다.

개선된 시스템 구조

- 수집과 처리를 분리
- Netty 기반 유지
- 로그를 수신하여 RabbitMQ를 이용하여 메세지 발행(publish)만
- 그뒤에서 가공처리(DB DML 작업, BigQuery)작업
- 사용자가 증가시 worker 프로그램만 추가 Startup
- TPS가 1대당 211까지 상승
- 드!디!어! Golang를 활용



Golang을 활용한 부분



RabbitMQ Consumer 역할

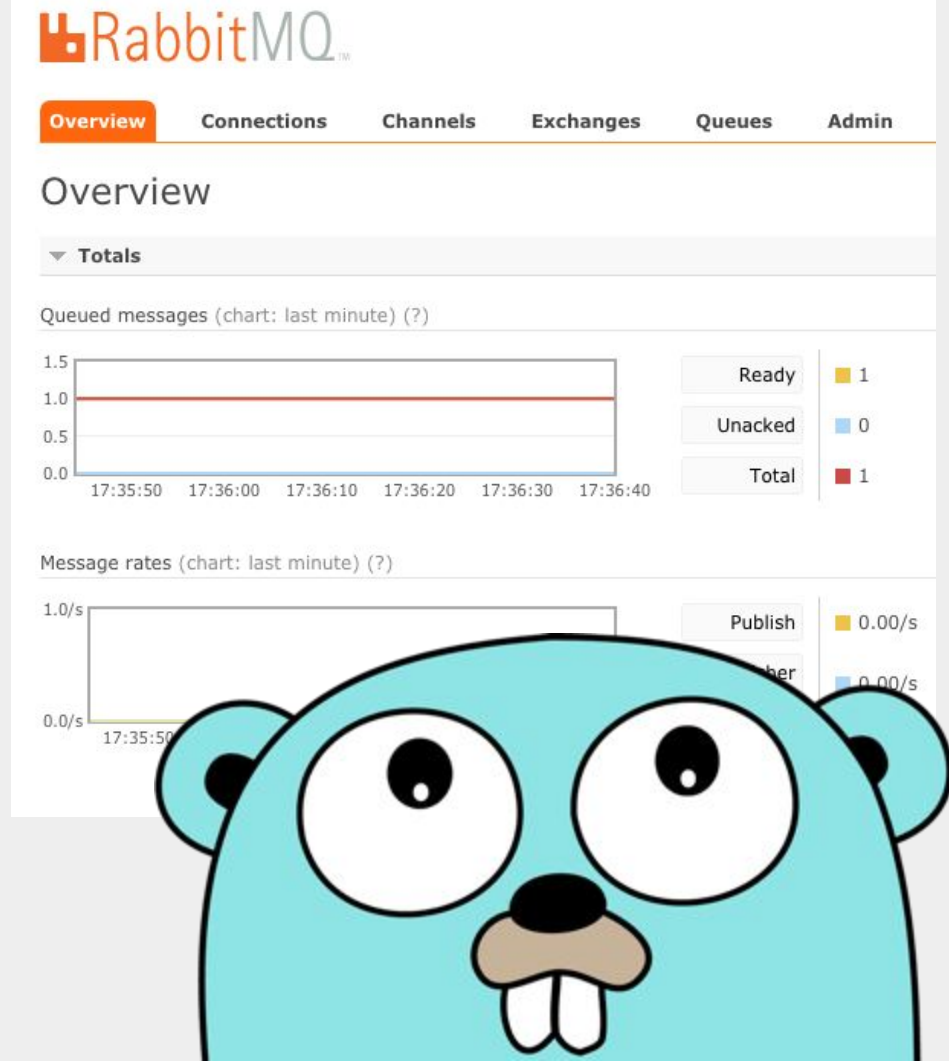
- RabbitMQ 연동
- string 문자열 가공
- RDMS DML 처리
- BigQuery Steaming Insert

오늘 나눌 부분은
Golang 자체보다
Golang으로 타 시스템
연동간의 경험이 중점

Contents

1. RabbitMQ 에서 데이터 가져오기
2. RabbitMQ 미들웨어 연동간 예외/에러 처리
3. 고속처리를 위한 벤치마킹 활용하기
4. 단위테스트를 통하여 단단한 프로그램 만들기
5. BigQuery와 연동하기

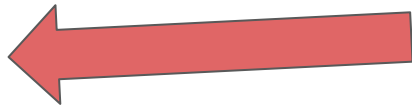
RabbitMQ 에서 데이터 가져오기



RabbitMQ 간략설명

- open source message broker.
- Asynchronous Messaging.
 - multiple messaging protocols

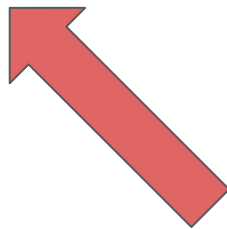
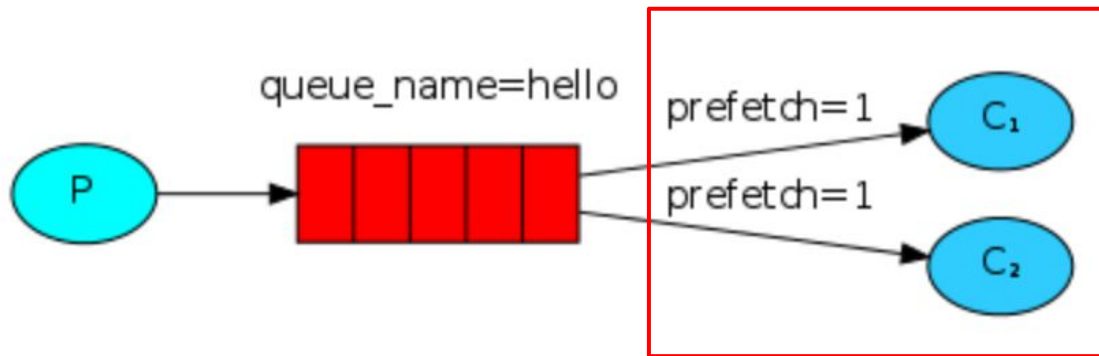
*** AMQP 0-9-1**



사용할 프로토콜

- message queuing
- delivery acknowledgement
- flexible routing to queues
- multiple exchange type

다수의 소비자가 분배하여 처리가능한 구조



Golang으로 개발한 부분

잘 정리된 예제와 문서

<https://www.rabbitmq.com/getstarted.html>

에 Golang도 있다!!

<https://www.cloudamqp.com/>

RabbitMQ Cloud Service도 있다.

(영어지만 번역기로 극복 가능한 정도)

RabbitMQ 연결하기

```
func main() {  
    conn, err := amqp.Dial( url: "amqp://spdconin:SmE00SdoMd1GRp  
  
    //Distributing tasks among workers  
    failOnError(err, msg: "Failed to connect to RabbitMQ")  
    defer conn.Close()  
  
    ch, err := conn.Channel()  
    failOnError(err, msg: "Failed to open a channel")  
    defer ch.Close()  
}
```

- Database 연결하듯이 연결정보를 나열하고 connection을 얻어옴.
- error 처리부분이 failOnError 공통함수로 처리됨.
- defer로 해제를 예약 함.
- connection에서 Channel도 얻어옴.

RabbitMQ 에 메시지 발행 Tutorial

- Connection 을 생성
- Channel 을 생성
- basicPublish 함수를

```
ConnectionFactory factory = new ConnectionFactory();  
factory.setHost("localhost");  
Connection connection = factory.newConnection();  
Channel channel = connection.createChannel();
```

이용하여 Send

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
String message = "Hello World!";  
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());  
System.out.println(" [x] Sent '" + message + "'");
```


spring-rabbit 을 이용한 RabbitMQ 발행

- application.yml
에 환경설정
- Autowired
- Queue 설정 및
메시지 보내기

```
@Autowired
private RabbitTemplate rabbitTemplate;

private void doSendMQ(String mqProtocol, String message) {
    rabbitTemplate.setQueue(mqProtocol);
    rabbitTemplate.convertAndSend(mqProtocol, message);
}
```

Queue 데이터 소비하기

```
msgs, err := ch.Consume(
    q.Name, // queue
    "",     // consumer
    false,  // auto-ack
    false,  // exclusive
    false,  // no-local
    false,  // no-wait
    nil,    // args
)
failOnError(err, msg: "Failed to register a consumer")
forever := make(chan bool)

go func() {
    for d := range msgs {
        log.Printf("Received a message: %s", d.Body)

        msg := string(d.Body)
        log.Printf("%s", msg)

        log.Printf("Done")
        d.Ack(false)
    }
}()

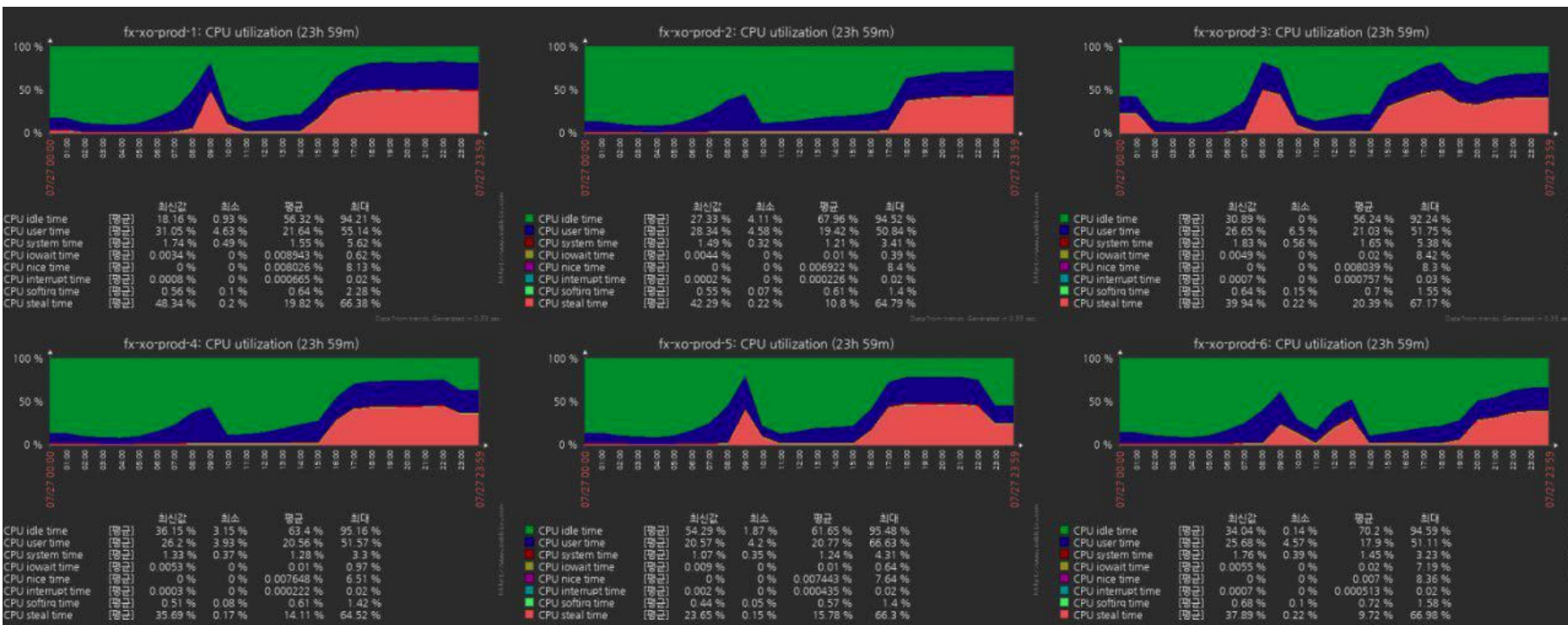
log.Printf(" [*] Waiting for messages. To exit press\n<-forever
```

- Channel에서 Consume 함수에 옵션값을 설정하여 데이터 얻기
- Consume 옵션들
- 데이터를 지속적으로 소비하기 위한 반복문을 Goroutines로 실행

예제와 가이드가 잘
정리되어있어, 코드화 하는
것은 어렵지 않았다..

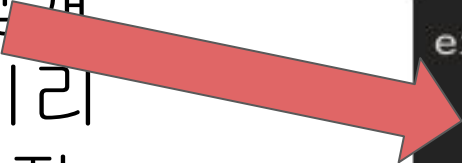
그런데 이런문제가 있었다.

Zabbix (서버 모니터링 툴)로 감지된 높은 CPU steal time



Worker당 너무 많은 prefetchCount가 원인

Worker당 몇개
(Count)씩 미리
분배할 것인지
설정 값.



```
err = ch.Qos(  
    1,      // prefetch count  
    0,      // prefetch size  
    false, // global  
)  
failOnError(err, "Failed to set QoS")
```

CPU steal time은
해결되었으나, 종종
Kill 되는 상황 발생.

RabbitMQ 미들웨어
연동간 예외/에러처리
:

```
if err != nil {  
    //TODO handler error  
}  
...  
log.Fatal("error msg...")
```



Golang에서 익숙한 패턴

```
val, err := myfunc()
```

```
if err != nil {
```

```
    //TODO handler error
```

```
}
```

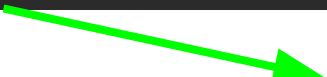
```
//or
```

```
val, err := myfunc()
```


```
failOnError(err, ".....")
```


Copy & Paste의 폐해

```
func main() {  
    err := throwConnectionErr()  
    failOnError(err, msg: "this is connection error")  
}
```



```
func failOnError(err error, msg string) {  
    if err != nil {  
        log.Fatalf(format: "%s: %s", msg, err)  
    }  
}
```



```
// Fatal is equivalent to Print() followed by a call to os.Exit(1).  
func Fatal(v ...interface{}) {  
    std.Output(calldepth: 2, fmt.Sprint(v...))  
    os.Exit( code: 1)  
}
```

log는 로그만 찍는게
아니였어??!! 웬 아니야.

Fatal은 초기구동시만 쓰고,
Runtime간 발생하는 부분에는
금지.

RabbitMQ 연동중 예외사항 발생

- Queue의 데이터가 폭증하여 RabbitMQ Service 마비
- RabbitMQ Connection 끊기는 현상 발생
- Golang으로 개발된 처리시스템도 Kill.. 또 Kill

Connection이 끊기는
예외사항 발생시 Kill
되지 않고 Connection
retry 시키자

예외처리를 위해 추가된 코드

```
go func() {  
    log.Errorf(format: "connection closing: %s",  
        <-conn.NotifyClose(make(chan *amqp.Error)))  
    time.Sleep(5 * time.Second)  
    main()  
}()
```

RabbitMQ connection에서 제공하는 NotifyClose 함수(close 됨을 알려주는 역할)로 감지하여 메인함수 실행

고속처리를 위한
benchmarks
활용하기

```
import "testing"
```

```
func BenchmarkFunc(){  
    //TODO bench define  
}
```



고성능이 요구되는
시스템에서 문자열
비교를 어떻게 하지?

다양한 문자열 비교 방법중 뭐가 빠르지?

`return a == b`

`return strings.Count(a, b) == 1`

`return strings.Compare(a, b) == 0`

`return strings.HasPrefix(a, b)`

`return strings.EqualFold(a, b)`

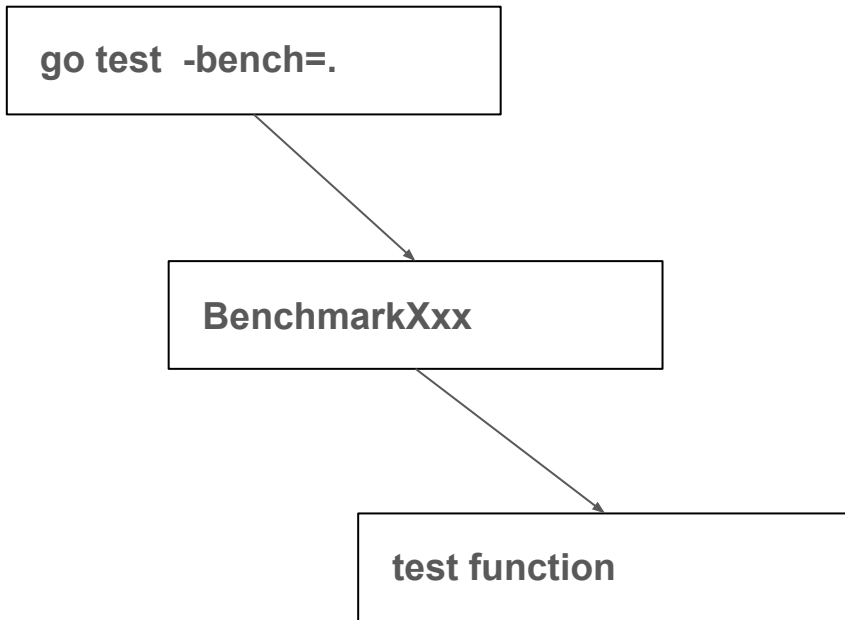
`return strings.Index(a, b) == 0`

`return strings.Contains(a, b)`

Benchmark 기능이
있다던데, 해보자!

간단 사용법

- 파일명이 *_test.go 로 끝날것
- import “testing” 할것
- 함수명이 Benchmark* 로 시작할것
- `$> go test -bench=.` 하면 끝



실제 Benchmark 코드

```
import "testing"

const COMPARE_STR = "ABCD"

func benchmarkCompareStr(
    TestComapare func(string, string) bool, b *testing.B) {
    for n := 0; n < b.N; n++ {
        TestComapare(COMPARE_STR, COMPARE_STR)
    }
}

func BenchmarkCompareStr1(b *testing.B) {
    benchmarkCompareStr( StrCompare, b) }
func BenchmarkCompareStr2(b *testing.B) {
    benchmarkCompareStr( StrEqual, b) }
func BenchmarkCompareStr3(b *testing.B) {
    benchmarkCompareStr( StrHasPrefix, b)}
func BenchmarkCompareStr4(b *testing.B) {
    benchmarkCompareStr( StrIndex, b) }
func BenchmarkCompareStr5(b *testing.B) {
    benchmarkCompareStr( StrContains, b)}
func BenchmarkCompareStr6(b *testing.B) {
    benchmarkCompareStr( StrCount, b) }
func BenchmarkCompareStr7(b *testing.B) {
    benchmarkCompareStr( StrEqualFold, b)}
```

```
func StrEqual(a string, b string) bool{
    return a == b
}

func StrCompare(a string, b string) bool{
    return strings.Compare(a, b) == 0
}

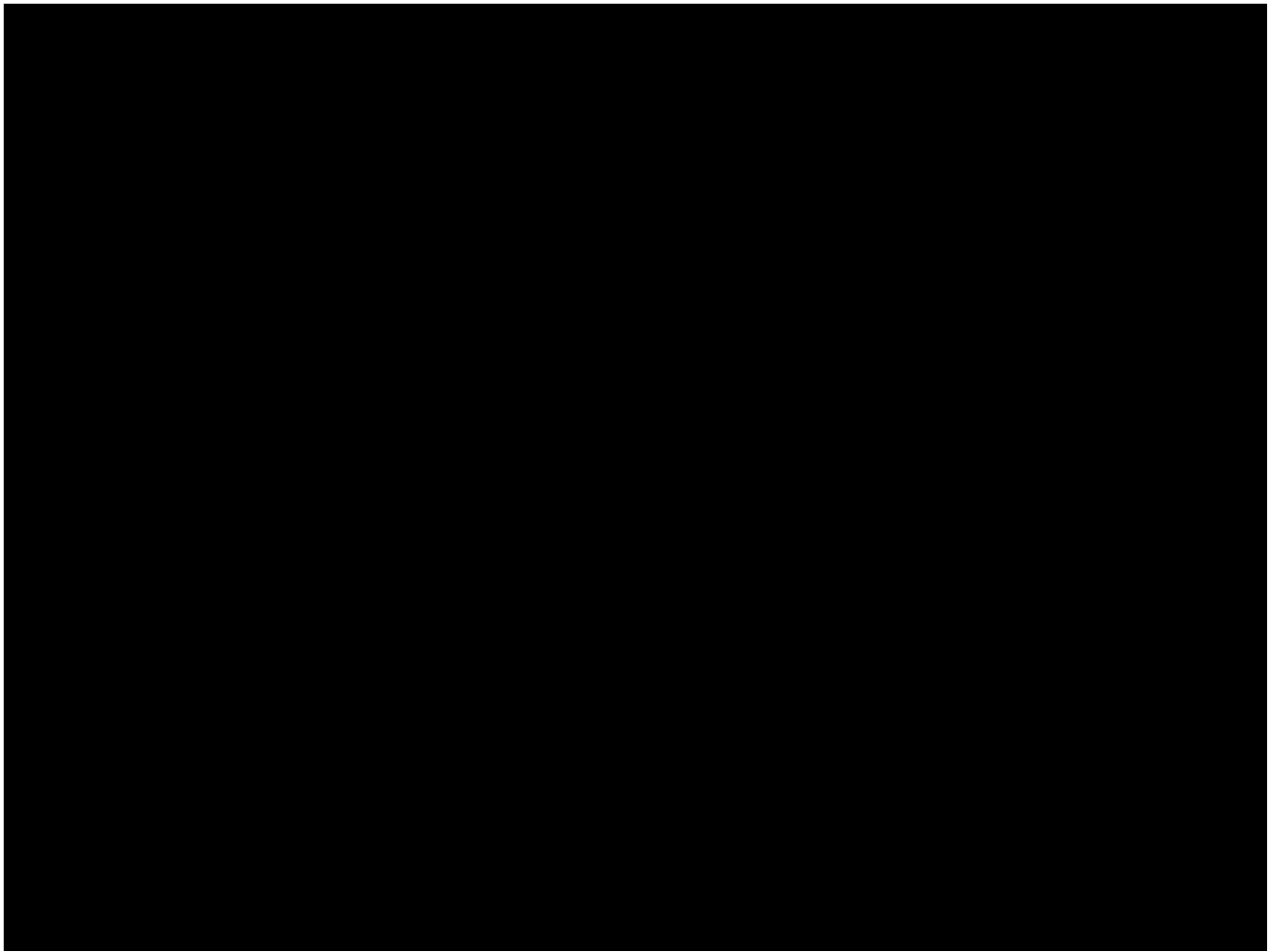
func StrEqualFold(a string, b string) bool{
    return strings.EqualFold(a, b)
}

func StrContains(a string, b string) bool{
    return strings.Contains(a, b)
}

func StrCount(a string, b string) bool{
    return strings.Count(a, b) == 1
}

func StrHasPrefix(a string, b string) bool{
    return strings.HasPrefix(a, b)
}

func StrIndex(a string, b string) bool{
    return strings.Index(a, b) == 0
}
```



결과 보는 방법

BenchmarkNum1-4	300000000	4.46 ns/op
BenchmarkNum2-4	300000000	4.77 ns/op
BenchmarkNum3-4	300000000	5.49 ns/op
BenchmarkNum4-4	200000000	7.64 ns/op
BenchmarkNum5-4	200000000	9.83 ns/op
BenchmarkNum6-4	100000000	16.1 ns/op
BenchmarkNum7-4	100000000	10.1 ns/op

BenchmarkXxx
함수

테스트 횟수

소요시간 ns단위

BenchmarkNum1-4	300000000	4.46 ns/op
BenchmarkNum2-4	300000000	4.77 ns/op
BenchmarkNum3-4	300000000	5.49 ns/op
BenchmarkNum4-4	200000000	7.64 ns/op
BenchmarkNum5-4	200000000	9.83 ns/op
BenchmarkNum6-4	100000000	16.1 ns/op
BenchmarkNum7-4	100000000	10.1 ns/op
BenchmarkOneStr1-4	300000000	4.50 ns/op
BenchmarkOneStr2-4	300000000	4.40 ns/op
BenchmarkOneStr3-4	300000000	5.10 ns/op
BenchmarkOneStr4-4	200000000	6.41 ns/op
BenchmarkOneStr5-4	200000000	8.65 ns/op
BenchmarkOneStr6-4	100000000	12.2 ns/op
BenchmarkOneStr7-4	100000000	18.0 ns/op
BenchmarkProtoStr1-4	300000000	4.53 ns/op
BenchmarkProtoStr2-4	300000000	4.42 ns/op
BenchmarkProtoStr3-4	300000000	5.30 ns/op
BenchmarkProtoStr4-4	200000000	6.38 ns/op
BenchmarkProtoStr5-4	200000000	8.58 ns/op
BenchmarkProtoStr6-4	100000000	12.3 ns/op
BenchmarkProtoStr7-4	200000000	66.2 ns/op
BenchmarkSha256Str1-4	300000000	4.18 ns/op
BenchmarkSha256Str2-4	300000000	4.53 ns/op

```
func StrCompare(a string, b string) bool{  
    return strings.Compare(a, b) == 0  
}
```

```
func StrEqual(a string, b string) bool{  
    return a == b  
}
```

가장 빠른 strings.Compare 함수와 가장 느린 함수

비교

```
func Compare(a, b string) int {  
    // NOTE(rsc): This function  
    // because we do not want to  
    // using strings.Compare. Ba  
    // As the comment above says  
    // If performance is importa  
    // the pattern so that all c  
    // using strings.Compare, ca  
    if a == b {  
        return 0  
    }  
    if a < b {  
        return -1  
    }  
    return +1  
}
```

7라인

```
func EqualFold(s, t string) bool {  
    for s != "" && t != "" {  
        // Extract first rune from each string.  
        var sr, tr rune  
        if s[0] < utf8.RuneSelf {  
            sr, s = rune(s[0]), s[1:]  
        } else {  
            r, size := utf8.DecodeRuneInString(s)  
            sr, s = r, s[size:]  
        }  
        if t[0] < utf8.RuneSelf {  
            tr, t = rune(t[0]), t[1:]  
        } else {  
            r, size := utf8.DecodeRuneInString(t)  
            tr, t = r, t[size:]  
        }  
        // If they match, keep going; if not, return false  
        // Easy case.  
        if tr == sr {  
            continue  
        }  
        // Make sr < tr to simplify what follows.  
        if tr < sr {  
            tr, sr = sr, tr  
        }  
        // Fast check for ASCII.  
        if tr < utf8.RuneSelf && 'A' <= sr && sr <= 'Z' {  
            // ASCII, and sr is upper case. tr must be  
            if tr == sr+'a'-'A' {  
                continue  
            }  
            return false  
        }  
        // General case. SimpleFold(x) returns the next  
        // or wraps around to smaller values.  
        r := unicode.SimpleFold(sr)  
        for r != sr && r < tr {  
            r = unicode.SimpleFold(r)  
        }  
        if r == tr {  
            continue  
        }  
        return false  
    }  
    // One string is empty. Are both?
```

20라인
이상

결론

비교문자	"1"	"ABCD"	"PROT.._GOLANG_MEE TUP"	"8f2af31397ffcb52 73.."
가장 빠른비교방법	strings.Compare(a, b) == 0	a == b	a == b	strings.Compare(a , b) == 0

비교하는 문자열의 길이에 따라 소요시간이 다르다.

“1” 혹은 64자리 문자열은 strings.Compare 함수를 사용
4자리 부터 22자리 정도는 == 비교를 사용함이 적절하다.

단위테스트를 통하여
단단한 프로그램
만들기

```
import (  
    "testing"  
    "log"  
)  
  
func TestFunc(){  
  
    //TODO Handler  
}
```



프로그램이 점점
커지고, 복잡해져서,
하나 수정하고 테스트
하기 힘들어.

단위 테스트 기능이
있었던데, 해보자!

자바보다 쉬웠다!

- 파일명이 *_test.go 로 끝날것
- import “testing” 할것
- 함수명이 Test* 로 시작할것
- \$> go test 하면 끝

```
package main

import (
    "testing"
    "log"
)

func TestDoExecProceder(t *testing.T) {

    log.Printf( format: "%s / %s", MQ_AH01, test2002)
    doExecProceder(MQ_AH01, test2002)

    log.Printf( format: "%s / %s", MQ_AH01, test2002_1)
    doExecProceder(MQ_AH01, test2002_1)

    log.Printf( format: "%s / %s", MQ_AH01, test2003)
    doExecProceder(MQ_AH01, test2003)

    log.Printf( format: "%s / %s", MQ_AH01, test2003_1)
    doExecProceder(MQ_AH01, test2003_1)

    log.Printf( format: "%s / %s", MQ_AH01, test2003_2)
    doExecProceder(MQ_AH01, test2003_2)
}
```

```

import (
    "testing"
)

func Test_iosTools(t *testing.T) {
    //r := iosTools("1111", MQ_ILAF)

    if iosTools(MQ_ILAF, tools: "") != "App tampering detected in iOS local environment." {
        t.Error( args: "Test_iosTools error..")}

    if iosTools(MQ_ILSF, tools: "") != "App tampering detected through iOS server verification." {
        t.Error( args: "Test_iosTools error..")}

    if iosTools(MQ_ISSF, tools: "") != "A GPS manipulated file was detected within the iOS system." {
        t.Error( args: "Test_iosTools error..")}

    if iosTools(MQ_ID01, tools: "") != "iOS An unusual debugging process has been detected." {
        t.Error( args: "Test_iosTools error..")}

    if iosTools( cate: "", tools: "One Tools[:]Two Tools") != "One Tools<br/>Two Tools" {
        t.Error( args: "Test_iosTools error..")}
}

```

```

NSHC-M-SYONui-MacBook-Pro-2:LogPiper nshc_m_syon$ go test
PASS
ok      _/Users/nshc_m_syon/GolandProjects/LogPiper    0.017s

```

BigQuery와 연동하기



Google
BigQuery





Google
BigQuery

Google BigQuery는 대용량 Dataset(최대 몇 십억 개의 행)를 대화식으로 분석하는 데 사용할 수 있는 웹 서비스입니다. 확장 가능하고 사용이 간편한 **BigQuery**를 통해 개발자와 기업은 필요할 때 강력한 데이터 분석을 수행할 수 있습니다.

BigQuery 사용 계기

- MAU/ DAU 측정을 위한 Raw Data를 기반으로 집계 필요
- RDMS에 Raw Data를 쌓기엔 부담. 쌓아도 조회시 부담.
- NoSQL 인프라 구축도, 구축이후 운영도 부담.
- BigQuery의 경우, 1년 무상 사용가능
- 대량 데이터 적재 부담 없고, 조회, 구축, 운영 부담 없음.
- 기존 SQL 거의 변화없이 그대로 사용 가능.

역시 예제와 가이드가
잘 되어있어 코드화
하는 데는 어려움은
없다.

BigQuery에 데이터 스트리밍하는 예제코드

[C#](#)[GO](#)[자바](#)[NODE.JS](#)[PHP](#)[PYTHON](#)[RUBY](#)

이 샘플을 시도해 보기 전에 [클라이언트 라이브러리](#)를 사용하는 [BigQuery 빠른 시작](#)의 Go 설정 안내를 따르세요. 자세한 내용은 [BigQuery Go API 참조 문서](#) [🔗](#)를 참조하세요.

[GITHUB에서 보기](#)[의견 보내기](#)

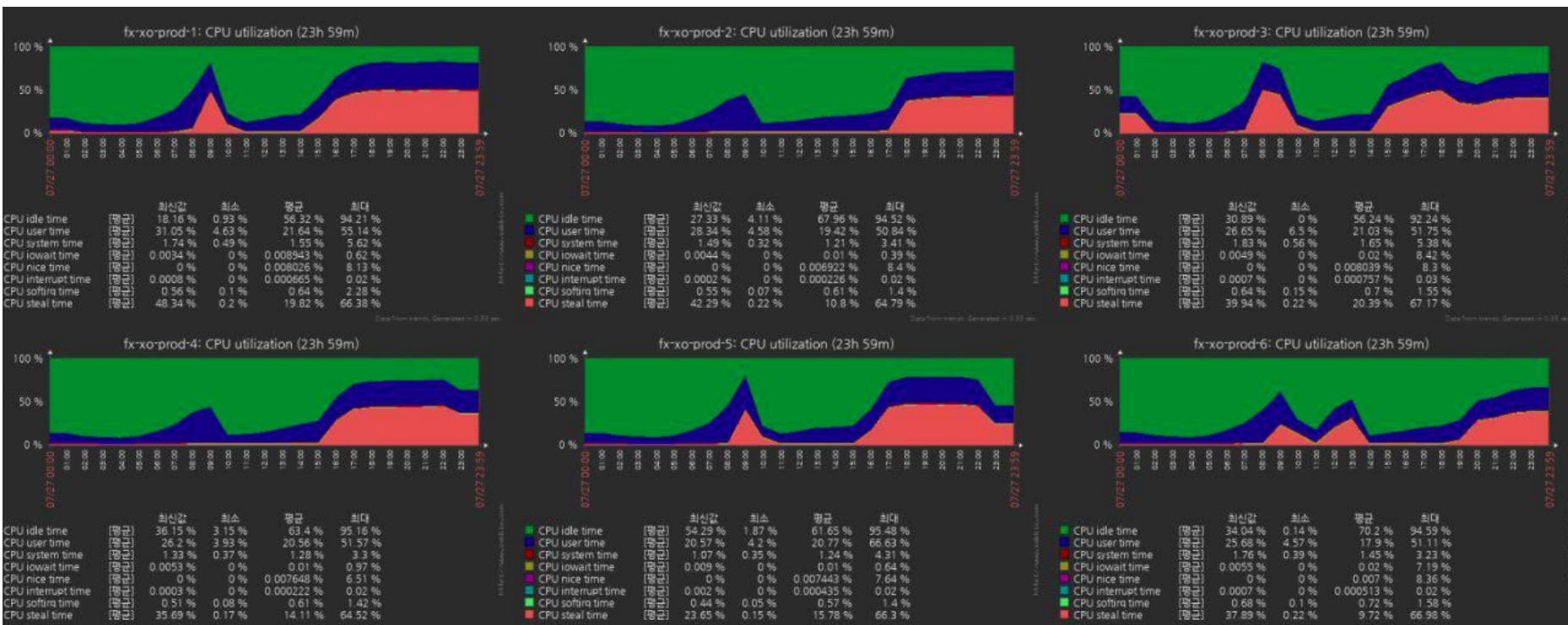
```
// To run this sample, you will need to create (or reuse) a context and
// an instance of the bigquery client. For example:
// import "cloud.google.com/go/bigquery"
// ctx := context.Background()
// client, err := bigquery.NewClient(ctx, "your-project-id")
u := client.Dataset(datasetID).Table(tableID).Uploader()
items := []*Item{
    // Item implements the ValueSaver interface.
    {Name: "Phred Phlyntstone", Age: 32},
    {Name: "Wylma Phlyntstone", Age: 29},
}
if err := u.Put(ctx, items); err != nil {
    return err
}
```

Database 처럼, 1건씩
Insert 를 하면 굉장히
느리다. 그래서 data를
append 해두었다가,
100건씩(변경가능)
Insert... 그런데..

문제시 코드

```
if connCnt >= bqAddCondition {  
  
    tmp := bqdata  
    bqdata = nil  
    connCnt = 0  
  
    log.Infof( format: "DoBQConnLog start")  
    DoBQConnLog(tmp)  
    //DoBQConnLog(tmp)  
}
```

또! 높은 CPU steal time 발생



개선된 코드

```
if connCnt >= bqAddCondition {  
  
    tmp := bqdata  
    bqdata = nil  
    connCnt = 0  
  
    log.Infof(format: "DoBQConnLog start")  
    go DoBQConnLog(tmp)  
    //DoBQConnLog(tmp)  
}
```

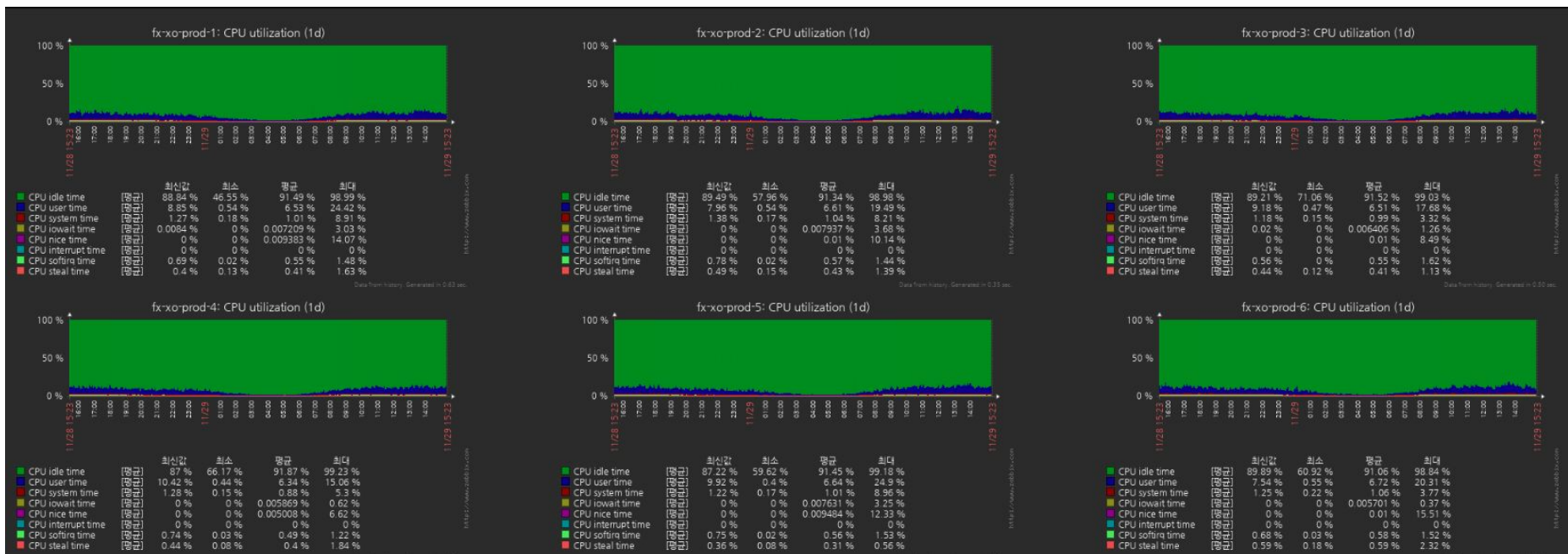
BigQuery에 일괄적재시, goroutine을 사용하여 비동기처리

```
items := []*connData{
    // Item implements the ValueSaver interface.
    {level: "info", date: "2018-11-28 12:11:01",
}

//BQInsert(items)

go BQInsert(items)
```

여유를 찾은 CPU 사용률



데이터는 멈추지 않고 계속
발행되므로 소비도
계속해야 한다. BigQuery에
Insert를 할 때, 지연된다면
CPU 부하발생, 따라서
goroutine

고맙습니다.