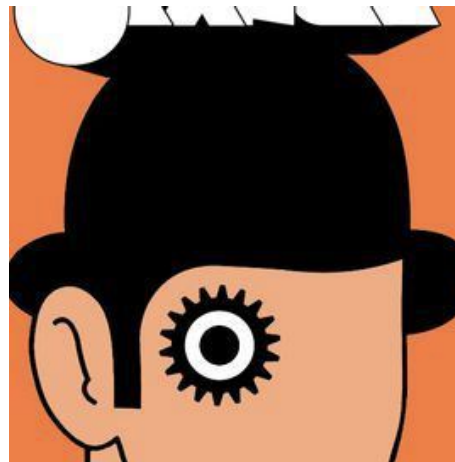
DBのTCPプロトコルとJDBC

JJUG CCC 2017 Fall

2017-11-18

自己紹介

- 山名 洋平
- 株式会社オプト（Opt Technologies）
- @vertical_blank
- Java, Groovy, Scala, PHP
- 初めてのJJUG発表です



おはなしすること

- きっかけ
- MySQL/PostgresのTCPプロトコル/通信内容
  SELECTを送り、ResultSetが戻ってくるまでの部分
- JDBC Types
- JDBC Version
- JDBCのこれから

みなさんJDBC使ってますか？

きっかけ

前職でものすごく暇な時期

ChromeAppsではJSで直接TCP叩けると聞き何か作れないか

Postgresのドライバ書いて遊んでいた(Trust認証のみ、SSLは非対応)

残念ながらソースは紛失

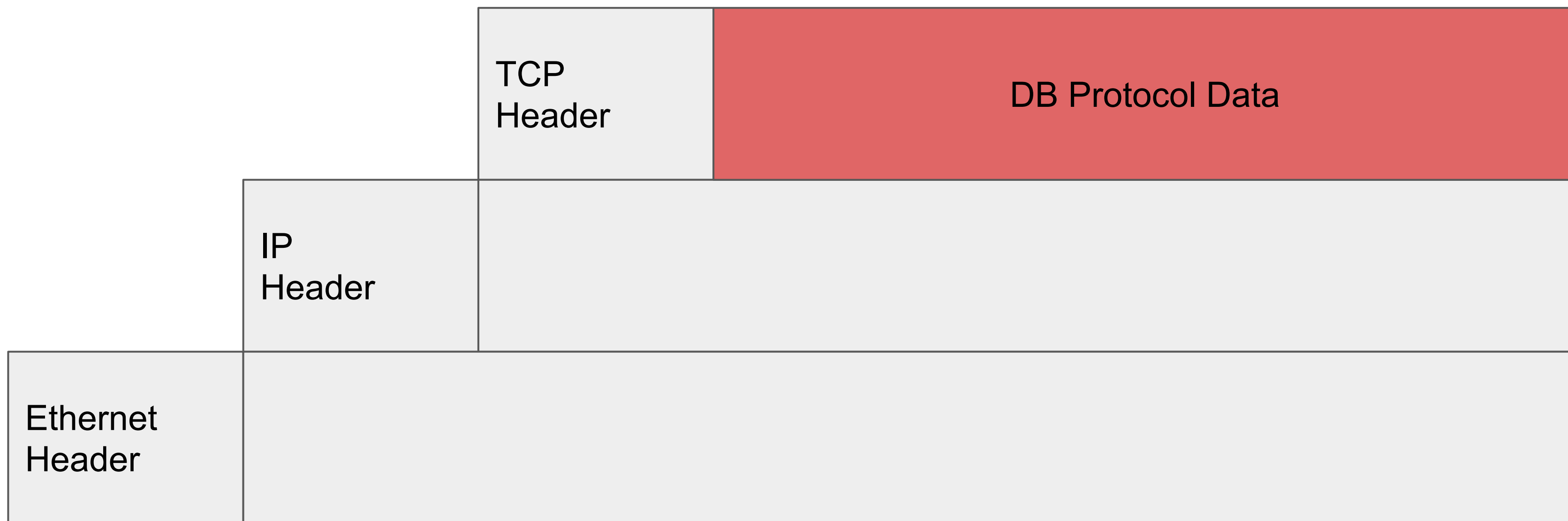
=> MySQLとか、Javaではどうなんだろう？

前提

確認した言語・ツール

- Docker
- PostgreSQL
  - psql
  - jdbc（groovy経由）
- MySQL
  - mysql-client
  - jdbc（groovy経由）
- いずれも非SSL
- WireShark

パケットの構造



文字コード（抜粋）

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	-	

先立って作ったテーブルデータ

- database: test
- table: BOOKS

id	name	author
1	Re-Engineering Legacy Software	Chris Birchall
2	EFFECTIVE JAVA	Joshua Bloch
3	JavaScript	David Flanagan

MySQLにクエリを投げるコード

```
@GrabConfig(systemClassLoader=true)
@Grab(group='mysql', module='mysql-connector-java', version='5.1.6')

import groovy.sql.*

def db = Sql.newInstance(
 url: 'jdbc:mysql://localhost/test',
 user: 'root',
 password: 'root')


def rows = db.rows('SELECT * FROM BOOKS')
rows.each{println it}
```

PostgreSQLにクエリを投げるコード

```
@GrabConfig(systemClassLoader=true)
@Grab(group='org.postgresql', module='postgresql', version='42.1.4')

import groovy.sql.*

def db = Sql.newInstance(
 url: 'jdbc:postgresql://localhost/test',
 user: 'postgres'
 password: 'postgres'
 preferQueryMode: 'simple')

def rows = db.rows('SELECT * FROM BOOKS')
rows.each{println it}
```

実際のパケットを見てみる

PostgreSQL

基本

tag 1Byte	length 4Byte	payload (length - 4) Byte

- tag
  - 送信
    - 'Q' 簡易問い合わせ
    - 'D' Describe
  - 受信
    - 'T' 結果セットのメタ情報
    - 'D' 結果セットのデータ部
- length (big-endian)
  - このフィールドを含めたByte長（つまり最低でも4）
- payload
  - tagに応じてさまざま

大まかな流れ

1. 送信
    - 簡易問い合わせ('Q')
2. 受信
    - メタデータ(カラム名、型など)('T')
    - 結果データ('D')
      行数分
    - 完了('C')
    - 待機('Z')

簡易問い合わせ送信

```
0000    51 00 00 00 18 53 45 4c 45 43 54 20 2a 20 46 52
0010    4f 4d 20 42 4f 4f 4b 53 00
```

tag	length	query
51	00 00 00 18	53 45 4c 45 43 54 20 2a 20 46 52 4f 4d 20 42 4f 4f 4b 53 00
'Q'	24	'SELECT * FROM BOOKS\0'

sqlの文字列はNULL終端含め20文字 ＝ 24 - 4(length自身)

大まかな流れ

1. 送信
    - 簡易問い合わせ('Q')
2. 受信
    - メタデータ(カラム名、型など)('T')
    - 結果データ('D')
      行数分
    - 完了('C')
    - 待機('Z')

メタデータ受信(1)

```
0000    54 00 00 00 4b 00 03 69 64 00 00 00 40 0e 00 01
0010    00 00 00 17 00 04 ff ff ff ff 00 00 6e 61 6d 65
0020    00 00 00 40 0e 00 02 00 00 00 19 ff ff ff ff ff
0030    ff 00 00 61 75 74 68 65 72 00 00 00 40 0e 00 03
0040    00 00 00 19 ff ff ff ff ff ff 00 00
```

tag	length	colnum count
54	00 00 00 4b	00 03
'T'	75	3

メタデータ受信(2)

```
0000    54 00 00 00 4b 00 03 69 64 00 00 00 40 0e 00 01
0010    00 00 00 17 00 04 ff ff ff ff 00 00 6e 61 6d 65
0020    00 00 00 40 0e 00 02 00 00 00 19 ff ff ff ff ff
0030    ff 00 00 61 75 74 68 65 72 00 00 00 40 0e 00 03
0040    00 00 00 19 ff ff ff ff ff ff 00 00
```

name	table id	column id	column type	type length	modifier	format (0:text,1:binary)
69 64 00	00 00 40 0e	00 01	00 00 00 17	00 04	FF FF FF FF	00 00
'id\0'		1	23(INT4)	4	-1	text
6e 61 6d 65 00	00 00 40 0e	00 02	00 00 00 19	FF FF	FF FF FF FF	00 00
'name\0'		2	25(TEXT)	-1	-1	text

大まかな流れ

1. 送信
    - 簡易問い合わせ('Q')
2. 受信
    - メタデータ(カラム名、型など)('T')
    - 結果データ('D')
行数分
    - 完了('C')
    - 待機('Z')

行データ受信(行数分繰り返し)

```
0000    44 00 00 00 3f 00 03 00 00 00 01 31 00 00 00 1e
0010    52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c
0020    65 67 61 63 79 20 53 6f 66 74 77 61 72 65 00 00
0030    00 0e 43 68 72 69 73 20 42 69 72 63 68 61 6c 6c
```

tag	length	colnum count	length	data	length	data	length	data
44	00 00 00 3f	00 03	00 00 00 01	31	00 00 00 1e	52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66 74 77 61 72 65	00 00 00 0e	43 68 72 69 73 20 42 69 72 63 68 61 6c 6c
'D'	63	3	1	'1'	30	'Re-Engineering Legacy Software'	14	'Chris Birchall'

列数分繰り返し

大まかな流れ

1. 送信
   - 簡易問い合わせ('Q')
2. 受信
   - メタデータ(カラム名、型など)('T')
   - 結果データ('D')
     行数分
   - 完了('C')
   - 待機('Z')

完了パケット受信

```
0000   43 00 00 00 0d 53 45 4c 45 43 54 20 33 00
```

tag	length	command tag
43	00 00 00 0d	53 45 4c 45 43 54 20 33 00
'C'	13	'SELECT 3\0' 'INSERT n', 'UPDATE n', 'DELETE n'だったりする

大まかな流れ

1. 送信
    - 簡易問い合わせ('Q')
2. 受信
    - メタデータ(カラム名、型など)('T')
    - 結果データ('D')
      行数分
    - 完了('C')
    - 待機('Z')

待機パケット受信

```
0000    5a 00 00 00 05 49
```

tag	length	command tag
5a	00 00 00 05	49
'Z'	5	'I' 待機状態(トランザクションブロックにない状態)に　'I' トランザクションブロック内の場合に　'T' 失敗したトランザクションブロック内の場合に　'E'

MySQL

基本

length 3Byte	seq 1Byte	payload (length) Byte

- length (little-endian)
  seqやこのフィールド自身を**含まない長さ**
  0xFFFFFF だった場合、次のパケットのpayloadを結合
- seq
  連番
- payload
  可変（直前が何だったかなど）

Length-encoded Integers

- 0x00 ~ 0xFA (250)
  - その値を採用（0 ~ 250）
- 0xFB (251)
  - NULL
- 0xFC (252)
  - 後続の2バイトを値とする(16bit int)
  - 本来1byteで表現できる 251 ~ 255 はここで表現されることになる
- 0xFD (253)
  - 後続の3バイトを値とする(24bit int)
- 0xFE (254)
  - 後続の8バイトを値とする(64bit int)

Length-encoded Strings

Length-encoded integers (1, 3, 4 ,9) Bytes	String (0 - 2^8) Bytes

今回の例では 1Byteで表現できる物のみ

例）

- **03** 41 42 43
  - "ABC"
- **04** 68 6f 67 6e **04** 46 55 47 41
  - "hoge", "FUGA" という独立した2つの文字列

大まかな流れ

1. 送信
    - 問い合わせ
2. 受信
    - カラム数
    - カラムメタデータ(カラム名、型など)
      カラム数分
    - EOFパケット
    - 行データ
      行数分
    - EOFパケット

問い合わせ

```
0000    14 00 00 00 03 53 45 4c 45 43 54 20 2a 20 46 52
0010    4f 4d 20 42 4f 4f 4b 53
```

length	seq	payload
14 00 00	00	03 53 45 4c 45 43 54 20 2a 20 46 52 4f 4d 20 42 4f 4f 4b 53 .. ..
20	0	03固定 'SELECT * FROM BOOKS'

sqlの文字列はNULL終端含めず19文字 ＝ 20 - 固定1

大まかな流れ

1. 送信
    - 問い合わせ
2. 受信
    - カラム数
    - カラムメタデータ（カラム名、型など）
      カラム数分
    - EOFパケット
    - 行データ
      行数分
    - EOFパケット

カラム数受信

```
0000   01 00 00 01 03      .....
```

length	seq	column count
01 00 00	01	03
1	1	3

カラム数は Length-encoded Integers
その場合lengthフィールドにも反映

大まかな流れ

1. 送信
    - 問い合わせ
2. 受信
    - カラム数
    - カラムメタデータ（カラム名、型など）
      カラム数分
    - EOFパケット
    - 行データ
      行数分
    - EOFパケット

カラムメタデータ受信（カラム数分繰り返し）

```
0000    28 00 00 02 03 64 65 66 04 74 65 73 74 05 42 4f
0010    4f 4b 53 05 42 4f 4f 4b 53 02 69 64 02 69 64 0c
0020    3f 00 0b 00 00 00 03 03 42 00 00 00
```

length	seq	column information
28 00 00	02	03 64 65 66 04 74 65 73 74 05 42 4f 4f 4b 53 05 42 4f 4f 4b 53 02 69 64 02 69 64 0c 3f 00 0b 00 00 00 03 03 42 00 00 00 .. ..
40	2	------ (3,'def') 固定 schema (4,'test') table alias (5,'BOOKS') table (5,'BOOKS') column alias (2,'id') column (2,'id') ------ 0C 固定 character set 3f 00 = 63(binary) max. column size 0b 00 00 00 = 11 Field types 3(LONG) Field detail flag 03 42 = (NOT NULL, PrimaryKey, AutoIncrement) decimals 0 ------ 00 00 固定

大まかな流れ

1. 送信
    - 問い合わせ
2. 受信
    - カラム数
    - カラムメタデータ（カラム名、型など）
      カラム数分
    - EOFパケット
    - 行データ
      行数分
    - EOFパケット

EOFパケット受信

```
0000   05 00 00 05 fe 00 00 22 00
```

length	seq	EOF
05 00 00	05	fe 00 00 22 00
5	5	----- 0xfe 固定 warning count 00 00 server status 22 00 (SERVER_QUERY_NO_INDEX_USED, SERVER_STATUS_AUTOCOMMIT)

大まかな流れ

1. 送信
    - 問い合わせ
2. 受信
    - カラム数
    - カラムメタデータ（カラム名、型など）
カラム数分
    - EOFパケット
    - 行データ
行数分
    - EOFパケット

行データ受信（行数分繰り返し）

```
0000    30 00 00 06 01 31 1e 52 65 2d 45 6e 67 69 6e 65
0010    65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66
0020    74 77 61 72 65 0e 43 68 72 69 73 20 42 69 72 63
0030    68 61 6c 6c
```

length	seq	row data
30 00 00	06	01 31 1e 52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66 74 77 61 72 65 0e 43 68 72 69 73 20 42 69 72 63 68 61 6c 6c
42	6	(01, 31) = '1' (1e, 52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66 74 77 61 72 65) = 'Re-Engineering Legacy Software' (0e, 43 68 72 69 73 20 42 69 72 63 68 61 6c 6c) = 'Chris Birchall'

大まかな流れ

1. 送信
    - 問い合わせ
2. 受信
    - カラム数
    - カラムメタデータ（カラム名、型など）
      カラム数分
    - EOFパケット
    - 行データ
      行数分
    - EOFパケット

EOFパケット受信

```
0000   05 00 00 09 fe 00 00 22 00
```

length	seq	EOF
05 00 00	09	fe 00 00 22 00
5	9	----- 0xfe 固定 warning count 00 00 server status 22 00 (SERVER_QUERY_NO_INDEX_USED, SERVER_STATUS_AUTOCOMMIT)

比較

	PostgreSQL	MySQL
数値表現	上位バイトが先頭 固定長	下位バイトが先頭 可変長
基本的な フォーマット	tag length payload	length seq payload
文字列表現	NULL終端	Length-encoded Strings
結果メタデータ	カラム数、カラム情報を まとめて	カラム数、カラムごとの 情報をそれぞれ
特徴	単体で解釈可能	文脈依存

MySQLにクエリを投げるコード

```
@GrabConfig(systemClassLoader=true)
@Grab(group='mysql', module='mysql-connector-java', version='5.1.6')

import groovy.sql.*

def db = Sql.newInstance(
 url: 'jdbc:mysql://localhost/test',
 user: 'root',
 password: 'root')


def rows = db.rows('SELECT * FROM BOOKS')
rows.each{println it}
```

PostgreSQLにクエリを投げるコード

```
@GrabConfig(systemClassLoader=true)
@Grab(group='org.postgresql', module='postgresql', version='42.1.4')

import groovy.sql.*

def db = Sql.newInstance(
 url: 'jdbc:postgresql://localhost/test',
 user: 'postgres'
 password: 'postgres'
 preferQueryMode: 'simple')

def rows = db.rows('SELECT * FROM BOOKS')
rows.each{println it}
```

どちらもやりたいことはほぼ同じ

- SQLの文字列を送信
- 表形式でデータを返してもらう

それを統一してくれるのがJDBC

共通する概念をInterfaceに抽出

- Driver
- Connection
- Statement / PreparedStatement / CallableStatement
- ResultSet
- DatabaseMetaData / ResultSetMetaData

ドライバのコードをチラ見してみる

PostgreSQL

簡易問い合わせ送信

```
0000    51 00 00 00 18 53 45 4c 45 43 54 20 2a 20 46 52
0010    4f 4d 20 42 4f 4f 4b 53 00
```

tag	length	query
51	00 00 00 18	53 45 4c 45 43 54 20 2a 20 46 52 4f 4d 20 42 4f 4f 4b 53 00
'Q'	24	'SELECT * FROM BOOKS\0'

sqlの文字列はNULL終端含め20文字 ＝ 24 - 4

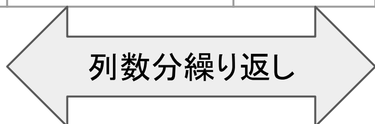
QueryExecutorImpl.java

```
private void sendSimpleQuery(SimpleQuery query, SimpleParameterList params)
throws IOException {
 String nativeSql = query.toString(params);

 LOGGER.log(Level.FINEST, " FE=> SimpleQuery(query=\"{0}\")", nativeSql);
 Encoding encoding = pgStream.getEncoding();

 byte[] encoded = encoding.encode(nativeSql);
 pgStream.sendChar('Q');
 pgStream.sendInteger4(encoded.length + 4 + 1);
 pgStream.send(encoded);
 pgStream.sendChar(0);
 pgStream.flush();
 pendingExecuteQueue.add(new ExecuteRequest(query, null, true));
 pendingDescribePortalQueue.add(query);
}
```

行データ受信(行数分繰り返し)

```
0000    44 00 00 00 3f 00 03 00 00 00 01 31 00 00 00 1e
0010    52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c
0020    65 67 61 63 79 20 53 6f 66 74 77 61 72 65 00 00
0030    00 0e 43 68 72 69 73 20 42 69 72 63 68 61 6c 6c
```

tag	length	colnum count	length	data	length	data	length	data
44	00 00 00 3f	00 03	00 00 00 01	31	00 00 00 1e	52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66 74 77 61 72 65	00 00 00 0e	43 68 72 69 73 20 42 69 72 63 68 61 6c 6c
'D'	63	3	1	'1'	30	'Re-Engineering Legacy Software'	14	'Chris Birchall'



QueryExecutorImpl.java

```
case 'D': // Data Transfer (ongoing Execute response)
  byte[][] tuple = null;
  try {
    tuple = pgStream.receiveTupleV3();
  } catch (OutOfMemoryError oome) {
    if (!noResults) {
    handler.handleError(
        new PSQLException(GT.tr("Ran out of memory retrieving query results."),
              PSQLState.OUT_OF_MEMORY, oome));
    }
  }
```

PGStream.java

```java
int l_msgSize = receiveInteger4();
int l_nf = receiveInteger2();
byte[][] answer = new byte[l_nf][];

OutOfMemoryError oom = null;
for (int i = 0; i < l_nf; ++i) {
  int l_size = receiveInteger4();
  if (l_size != -1) {
    try {
      answer[i] = new byte[l_size];
      receive(answer[i], 0, l_size);
    } catch (OutOfMemoryError oome) {
      oom = oome;
      skip(l_size);
    }
  }
}
```

MySQL

Length-encoded Integers

- 0xFB (251) 未満
  - その値を採用（0 ~ 250）
- 0xFB (251)
  - NULL
- 0xFC (252)
  - 後続の2バイトを値とする(16bit int)
  - 本来1byteで表現できる 251 ~ 255 はここで表現されることになる
- 0xFD (253)
  - 後続の3バイトを値とする(24bit int)
- 0xFE (254)
  - 後続の8バイトを値とする(64bit int)

Buffer.java（readFieldLength）

```
switch (sw) {
    case 251:
        return NULL_LENGTH;

    case 252:
        return readInt();

    case 253:
        return readLongInt();

    case 254:
        return readLongLong();

    default:
        return sw;
}
```

MysqlIO.java

```
switch (sw) {
    case 251:
        len = NULL_LENGTH;
        break;

    case 252:
        len = (this.mysqlInput.read() & 0xff) | ((this.mysqlInput.read() & 0xff) << 8);
        remaining -= 2;
        break;

    case 253:
        len = (this.mysqlInput.read() & 0xff) | ((this.mysqlInput.read() & 0xff) << 8) | ((this.mysqlInput.read() & 0xff)
<< 16);

        remaining -= 3;
        break;

    case 254:
        len = (int) ((this.mysqlInput.read() & 0xff) | ((long) (this.mysqlInput.read() & 0xff) << 8)
                | ((long) (this.mysqlInput.read() & 0xff) << 16) | ((long) (this.mysqlInput.read() & 0xff)<< 24)
                | ((long) (this.mysqlInput.read() & 0xff) << 32) | ((long) (this.mysqlInput.read() & 0xff)<< 40)
                | ((long) (this.mysqlInput.read() & 0xff) << 48) | ((long) (this.mysqlInput.read() & 0xff)<< 56));
        remaining -= 8;
        break;

    default:
        len = sw;
}
```

問い合わせ

```
0000    14 00 00 00 03 53 45 4c 45 43 54 20 2a 20 46 52
0010    4f 4d 20 42 4f 4f 4b 53
```

length	seq	payload
14 00 00	00	03 53 45 4c 45 43 54 20 2a 20 46 52 4f 4d 20 42 4f 4f 4b 53 .. ..
20	0	03固定 'SELECT * FROM BOOKS'

sqlの文字列はNULL終端含めず19文字 ＝ 20

MysqlIO.java

```
final ResultSetInternalMethods sqlQueryDirect(StatementImpl callingStatement, String query,
String characterEncoding, Buffer queryPacket, int maxRows, int resultSetType, int
resultSetConcurrency, boolean streamResults, String catalog, Field[] cachedMetadata) {
/* 中略 */
this.sendPacket.writeByte((byte) MysqlDefs.QUERY);
/* 中略 */
this.sendPacket.writeStringNoNull(query);
/* 中略 */
queryPacket = this.sendPacket;
/* 中略 */
Buffer resultPacket = sendCommand(MysqlDefs.QUERY, null, queryPacket, false, null, 0);
```

```
public final class MysqlDefs
    static final int QUERY = 3;
```

カラム数受信

```
0000   01 00 00 01 03     .....
```

length	seq	column count
01 00 00	01	03
1	1	3

カラム数は Length-encoded Integers
その場合lengthフィールドにも反映

行データ受信（行数分繰り返し）

```
0000    30 00 00 06 01 31 1e 52 65 2d 45 6e 67 69 6e 65
0010    65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66
0020    74 77 61 72 65 0e 43 68 72 69 73 20 42 69 72 63
0030    68 61 6c 6c
```

length	seq	row data
30 00 00	06	01 31 1e 52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66 74 77 61 72 65 0e 43 68 72 69 73 20 42 69 72 63 68 61 6c 6c
42	6	(01, 31) = '1' (1e, 52 65 2d 45 6e 67 69 6e 65 65 72 69 6e 67 20 4c 65 67 61 63 79 20 53 6f 66 74 77 61 72 65) = 'Re-Engineering Legacy Software' (0e, 43 68 72 69 73 20 42 69 72 63 68 61 6c 6c) = 'Chris Birchall'

MysqlIO.java

readResultsForQueryOrUpdate

```
long columnCount = resultPacket.readFieldLength();
```

getResultSet

```
for (int i = 0; i < columnCount; i++) {
    Buffer fieldPacket = null;

    fieldPacket = readPacket();
    fields[i] = unpackField(fieldPacket, false);
}
```

MysqlIO.java (versionMeetsMinimum)

```
if (getServerMajorVersion() >= major) {
    if (getServerMajorVersion() == major) {
        if (getServerMinorVersion() >= minor) {
            if (getServerMinorVersion() == minor) {
                return (getServerSubMinorVersion() >= subminor);
            }

            // newer than major.minor
            return true;
        }

        // older than major.minor
        return false;
    }

    // newer than major
    return true;
}

return false;
```

JDBC Driver Types

DBとの通信方法の違いによる分類

JDBC Driver Types

- Type 1

  ODBCドライバをJNI経由で呼び出すもの
  sun.jdbc.odbc.jdbcodbcdriver
  各種DBの違いはODBCで吸収されるため、事実上これが唯一の実装
  JDK8から同梱されなくなった

- Type 2

  DB固有ネイティブドライバをJNI経由で呼び出す

JDBC Driver Types

- Type 3

  DBにアクセスする中間サーバがあり、そのサーバにより変換された(TCP)プロトコルを解釈するPureJava実装されたドライバ

  com.ibm.db2.jdbc.net.DB2Driver
  Java アプレット用に設計された物らしい

JDBC Driver Types

- Type 4

  java.net.SocketによりPureJavaでDBに直接アクセス

  - 送信パケット構築
  - 受信パケット解析

  チラ見した MySQL, PostgreSQLドライバなどはこれにあたる

いろいろなJDBC Driver

いろいろなJDBC Driver

- H2 Database
  組み込みDB
  PostgreSQL互換サーバとしても動作する（TCPプロトコルが同一）
- https://github.com/opt-tech/redshift-fake-driver
  COPY/UNLOADの向き先を(fake-)S3に接続出来る
- https://github.com/mockrunner/mockrunner
  mockrunner-jdbc / MockDriver
  JDBC経由の処理をまるごとモック化する
- https://github.com/jprante/jdbc-driver-csv
  CSVをSQLで検索できる

JDBC Versions

JDBC Versions

- 1.0 (JDK 1.1)
- 2.0 (JDK 1.2)
  - Statement#~~~Batch 系が追加
- 3.0 (JDK 1.4)
  - getGeneratedKeys 追加（auto_increment で生成された値を取得）
- 4.0 (JDK 6)
  - Class.forName(DriverName) 不要化 (META-INF/services/java.sql.Driver)
    ※ドライバと利用クラスがシステムクラスローダでロードされた場合に限る

JDBC Versions

- 4.1 (JDK 7)
  - try-with-resources 対応
  - Connection,Statement,ResultSetなどがAutoCloseableに
- 4.2 (JDK 8)
  - いくつかのinterfaceにdefaultメソッド追加
  - JDBCType列挙体追加（カラム型はint定数だった）

JDBCのこれから

Asynchronous JDBC API

JavaOne(10/3)で発表された模様

OOW and Java One 2017 Presentations

CON1491 - JDBC Next: A New Asynchronous API for Connecting to a Database

開発中コードスナップショット
http://hg.openjdk.java.net/jdk10/sandbox/jdk/file/a31057bda7c5/src/java.sql/share/classes/java/sql2

Asynchronous JDBC API

- 既存の型とは互換性無し
  - java.sql2パッケージ
- Java 9で追加になったFlow APIとの連携
  - Connectionに以下の様なメソッドが存在

    `Flow.Publisher publisherOperation(String sql)`
- ドライバ側は基本的に書き直し
  - Oracle用ドライバがプロトタイプ作成中（残念ながら未公開）
    - nio Selectorにより完全非同期で書かれているらしい

コード例

```java
public Future<List<String>> selectAllUserNames(DataSource ds) {
  try (Connection conn = ds.getConnection()) {
    return conn.rowOperation("SELECT id, name FROM BOOKS")
      .initialValue(() -> new ArrayList<>())
      .rowAggregator((list, row) -> {
         list.add(row.get("name", String.class));
         return list;
      })
      .submit()
      .toCompletableFuture();
  }
}
```

何がうれしいの？

- 現在のJDBCはブロッキングIO
- スレッドをブロックしない
  - 外部への通信の結果を待っている間もマシンは処理を実行出来る
- 具体的な例としては…
  - 某社の某play frameworkアプリ
  - デモ中におよそ30人もの大量アクセスによりスレッドプールが枯渇
  - playのスレッドプール数はデフォルトだとCPUのコア数
  - 重いクエリを発行する部分だけ、専用のスレッドプールで実行するよう設定し解消

さいごに

- DB間でSQL方言の違いだけでなく、通信している内容も相当違う
- 接続文字列とSQL文法の違いだけ気を使えばDBを使える
  - 各JDBCドライバが共通のinterfaceを実装(した型に通信内容を変換)してくれているから
  - DBごとに共通のAPIでアクセスできる
- Type4ならJavaだけでDBアクセスを完結できる
  - Jarさえあれば繋がる
  - ネイティブクライアントをaptとか叩いて入れなくても良い
- 自分でもドライバ作れるかもって気持ちになってきますね！

ご清聴ありがとうございました