Blockchain Core Camp

RPCアプリの作成

@DG Lab - Karl-Johan Alm

（追加書類）

目次

- Arrow関数
- 非同期プログラミング

Arrow関数

Arrow関数

Arrow関数はECMAScript6以降で使えるようになった。
大きく二つの特徴がある：

① `function` を用いた無名関数よりも簡潔に書ける

② `this`が変わらない（関数定義時のスコープで決まる）

Arrow関数

普通の関数

```
function x(y) {
  return y + 1;
}
```

Arrow関数

```
const x = (y) => {
  return y + 1;
}
```

```
x(1) // 2
```

Arrow関数

一行で終わる場合、Arrow関数は更に短く記述できる。

普通のArrow関数

```
const x = (y) => {
  return y + 1;
};
```

{ }無しのArrow関数

```
const x = (y) => y + 1;
```

```
x(1) // 2
```

Arrow関数

関数を返す場合、入れ子にすることもできる。

```
function x(y) {
  function z(u) {
    return y + u;
  }
  return z;
}
```

```
const x =
    (y) =>
        (u) => y + u;
```

```
x(1)(2) // 3
```

非同期プログラミング

非同期プログラミング

node.jsでは非同期処理の記述が非常に多くなる。

普通の言語:

```
try {
    x = fun(y);
    // xを使う
} catch (err) {
    handle(err);
}
```

node.jsでよくあるパターン:

```
fun(y, (err, x) => {
    if (err) return handle(err);
    // xを使う
});
```

非同期プログラミング

node.jsでは非同期処理の記述が非常に多くなる。

普通の言語:

```
try {
    x = fun(y);
    // xを使う
} catch (err) {
    handle(err);
}
```

node.jsでよくあるパターン:

```
fun(y, (err, x) => {
    if (err) return handle(err);
    // xを使う
});
```

これを
```
if (err) {
  handle(err);
  return;
}
```
としても良いが、短くするために一緒にする事が多い。

非同期プログラミング

普通の言語:

```
console.log('start');
try {
    x = fun(y);
    console.log('got x');
    // xを使う
} catch (err) {
    handle(err);
}
console.log(end');
```

node.jsでよくあるパターン:

```
console.log('start');
fun(y, (err, x) => {
    if (err) return handle(err);
    console.log('got x');
    // xを使う
});
console.log('end');
```

非同期プログラミング

普通の言語：

```
start

got x

end
```

node.jsでよくあるパターン：

```
start

end

got x
```

普通の言語（同期処理）と比べると、処理順序が違う。

callbackの呼び方

callbackの呼び出しを別の非同期関数に任せるために引き渡すときは、そのcallbackオブジェクトに括弧をつけない:

```
function fun(callback) {
  if (foo === bar)
    fun2(callback); // fun2に関数型の変数を引数として渡す
  // ...
}
```

callbackの呼び方

callbackを関数として呼び出す時は、そのcallbackオブジェクトに括弧をつける:

```
function fun2(callback) {
  // …
  callback(result);
}
```

callbackの呼び方

極端な話をすれば、関数が関数を返すこともできる:

```
function makeAddFunStr(a) {
  return (b) => `${a} + ${b} = ${a + b}`;
}
const plus1 = makeAddFunStr(1);
console.log(plus1(2)); // 1 + 2 = 3
console.log(makeAddFunStr(3)(4)); // 3 + 4 = 7
```

非同期プログラミング、再び

callbackを使って、複数の非同期の関数に何段階も潜り込んでいくように記述する事も多い。

```
function initGame(gameid, callback) {
  getGame(gameid, (err, game) => {
    if (err) return callback(err);
    getPlayers(game, (err2, players) => {
        if (err2) return callback(err2);
        callback(null, { game, players });
    });
}
```

非同期プログラミング、再び

callbackが多いと分かりづらくなるのでasyncライブラリの利用などで対策する。例えば前スライドと同じ順次処理は以下のように書ける:

```
function initGame(gameid, callback) {
  let game;
  async.waterfall([
    (c) => getGame(gameid, c),
    (gameIn, c) => { game = gameIn; getPlayers(game, c); },
  ], (err, players) => callback(err, { game, players }));
}
```

@DG Lab - Karl-Johan Alm